

ПУСТЬ ЭТА КНИГА ПРИНЕСЕТ ВАМ УДАЧУ!

КРИС КАСПЕРСКИ



ФУНДАМЕНТАЛЬНЫЕ ОСНОВЫ ХАКЕРСТВА

ИСКУССТВО ДИЗАССЕМБЛИРОВАНИЯ

информация защищена информация защищена информация защищена



Код
серия
Копатель

Книга
5

Крис Касперски

ФУНДАМЕНТАЛЬНЫЕ ОСНОВЫ ХАКЕРСТВА

ИСКУССТВО ДИЗАССЕМБЛИРОВАНИЯ

**Москва
СОЛОН-Р
2007**

Крис Касперски

Фундаментальные основы хакерства. Искусство дизассемблирования / Крис Касперски — М.: СОЛОН-Р, 2007. 448 с. — (Серия «Кодокопатель»)

ISBN 5-93455-175-2

Книга, которую вы сейчас держите в руках, открывает двери в удивительный мир защитных механизмов — здесь рассказывается о том, как создаются и вскрываются защиты. Она адресована всем, кто любит захватывающие дух головоломки, всем, кто проводит свободное и несвободное время за копанием в недрах программ и операционной системы. Наконец, эта книга предназначена для тех, кто по роду своей деятельности занимается (постоянно и/или эпизодически) написанием защит и хочет узнать, как грамотно и уверенно противостоять вездесущим хакерам.

Настоящий том посвящен базовым основам хакерства — технике работы с отладчиком и дизассемблером. Здесь подробно описаны приемы идентификации и реконструкции ключевых структур исходного языка — функций (в т. ч. виртуальных), локальных и глобальных переменных, ветвлений, циклов, объектов и их иерархий, математических операторов и т. д.

Эту книгу можно заказать по почте (наложенным платежом — стоимость 281 руб.)

двумя способами:

- 1) выслать почтовую открытку или письмо по адресу: 123001, Москва, а/я 82;
- 2) передать заказ по электронной почте (e-mail) по адресу:

magazin@solon-r.ru.

Необходимо написать полный адрес, по которому выслать книги.

Обязательно указывать индекс и Ф. И. О. получателя!

При наличии — указать телефон, по которому с вами можно связаться, и адрес электронной почты (E-mail).

Цены действительны до 1 сентября 2002 г.

Вы можете в любое время получить свежий каталог издательства «СОЛОН-Р» по **Интернету**, послав пустое письмо на робот-автоответчик по адресу katalog@solon-r.ru, а также подписаться на **рассылку новостей** о новых книгах издательства, послав письмо по адресу news@solon-r.ru с текстом «SUBSCRIBE» (без кавычек) в теле письма.

*Светлой памяти Сергея Иванова, главного редактора
издательства «СОЛОН-Р», посвящается эта книга*

Предисловие редактора

The only secure computer is one that's unplugged, locked in a safe, and buried 20 feet under the ground in a secret location... and I'm not even too sure about that one...

Дэннис Хьюжз (Dennis Huges), ФБР США

Этот эпиграф выбран не случайно. Информационная безопасность сегодня представляет одну из весьма горячих тем. Ее актуальность сложно преувеличить, и каждое связанное с этой темой пособие неизменно подвергается анализу со стороны, как правило, весьма скептически настроенных специалистов. Исследование программ напрямую связано с вопросами информационной безопасности. Когда автор этой книги пригласил меня, как специалиста, стать ее научным редактором, я весьма заинтересовался этим предложением.

Сама мысль о возможности опубликования подобных материалов звучит для многих несколько крамольно, как некогда было, к примеру, с криптографией и некоторыми областями теории чисел. Более того, тематика данной книги до некоторого времени расценивалась как близкая к широко обсуждаемым криминальным темам и лишь в последнее время вернулась в свое естественное научное русло.

На мой взгляд, эта книга будет интересна для весьма широкого круга читателей. Наверняка ею заинтересуются и те, кто лишь начинает свой восход к Олимпу знаний, и уже матерые специалисты в области программирования и исследования программ (или на иностранный манер «reverse engineering»). Хочется особенно отметить, что материалы книги подобраны и скомпонованы таким образом, что будут полезны и обычному программисту, допустим, как пособие по оптимизации программ для современных интеллектуальных компиляторов, и специалистам различных направлений (например, специалистам информационной защиты — в качестве пособия по поиску так называемых «закладок»). Стиль изложения «от простого к сложному» позволяет говорить и о том, что данная книга послужит также и учебным пособием для начинающих исследователей и «кодекопателей».

Книга содержит множество уникального практического материала. Большинство изданных за рубежом поверхностных работ малоинтересны с практической точки зрения для тех, кто изучает прикладную математику, программирование и устройство компьютеров. Да и среди публикаций современного российского научного сообщества читатель, как мне кажется, не найдет лучшего пособия по изучению техники исследования программ.

Однако, я все-таки рекомендую читателю подвергнуть сомнению все вышесказанное и убедиться во всем самостоятельно, прочитав данную книгу.

С уважением, Хади Р. А.

Что нового во втором издании

Как бы плохо вы ни написали вашу повесть, у вас обязательно найдутся читатели, тысячи читателей, которые сочтут ее шедевром... Как бы хорошо вы ни написали свою повесть, обязательно найдутся читатели, и это будут тысячи читателей, которые сочтут ее чистым барахлом.

Б. Н. Стругацкий

Первое издание «Техники и философии хакерских атак» — довольно фри-вольное и хаотичное — по стилю изложения напоминало собой «Путевые заметки охотника» — читается, может быть, и с интересом, но на учебник, увы, не тянет. К огромному удивлению автора, книга имела ощутимый успех и множество одобрительных откликов. Оно, конечно, понятно — на безрыбье и рак рыба — за последние несколько лет ничего путного по данной тематике не выходило.

Когда же тираж книги был полностью распродан, но заявки на нее по-прежнему продолжали поступать, встал вопрос: что делать дальше: выпускать допечатку, скопированную один к одному, или переработанное и дополненное второе издание? Издатель склонялся к последнему, да и автор, в желании утолить свой профессиональный зуд, признаться, тоже. Однако за время, прошедшее с момента первого издания, автор стал писать значительно структурней и «чище». Поэтому после долгих колебаний, сомнений и тягостных раздумий автор решил полностью переписать книгу с нуля, превратив ее в реальную настольную книгу хакера. Написать своеобразный справочник кодокопателя, но вместе с тем и учебник, помогающий начинающим сделать в этом мире свои первые шаги.

Попутно, движимый просьбами читателей, ожидающих поскорее увидеть продолжение трилогии «*Образ мышления — дизассемблер IDA*», автор рискнул включить в настоящее издание двадцать глав из будущей книги «*Искусство дизассемблирования*» (название рабочее).

Объем книги увеличился настолько, что ее пришлось разбить на несколько томов. Этот том, первый из всех, посвящен базовым основам хакерства — технике работы с отладчиком и дизассемблером, также затронуты вопросы защиты программ от изучения и способы нейтрализации этих защит. Подробный рассказ о методике создания и снятия защитных механизмов — тема последующих томов.

Кто такие хакеры

...Назови ты меня вчера быком, я был бы быком. Назвал бы ты меня лошастью — и я был бы лошастью. Если люди дают имя какой-то сущности, то, не приняв этого имени, навлечешь на себя беду.

Приписывается китайскому мудрецу Лао-Цзы

Прежде чем подавать на стол блюда хакерской кухни, неплохо бы разобраться, кто, собственно, такие эти хакеры и что они едят. Заглянув в толковый словарь английского языка, например в «The American Heritage Dictionary», мы убедимся, что глагол **«hack»** возник в английском лексиконе задолго до появления компьютеров и в прямом смысле обозначал «бить, рубить, кромсать» (но не уродовать!) топором, мотыгой или молотом, иными словами, выполнять физически тяжелую, монотонную, нудную, интеллектуально непритязательную работу — что всегда было уделом батраков, неудачников и бездарей. Неудивительно, что производные от глагола «hack» обозначали «бить баклуши», «халтурить», «выполнять работу наспех», — ведь наемные рабочие испокон веков трудились из-под палки! Термин считался пренебрежительным, если не ругательным: «hack» стало даже синонимом нашего слова «кляча». Словом, в докомпьютерную эпоху титулом хакера ни один здравомыслящий человек гордиться бы не стал.

Сегодня же «хакер» звучит практически так же, как «национальный герой», пускай и преступный, но все же крутой малый, которому не грех подражать. Чем же объясняется такая нелепая мутация данного слова?

По одной из гипотез в щелчке, издаваемом реле, американцам слышалось «хак-хак». Динозавры машинной эры состояли из многих тысяч реле и «хакали» вовсю, особенно когда оператор ЭВМ запускал очередную программу на выполнение. Возможно, именно за это операторов и прозвали хакерами. Или, говоря по-русски, клацальщиками. По другой гипотезе звук «хак» приписывается перфоратору, кромсающему перфоленту на мелкие куски, так что щепки (такие аккуратненькие круглые «щепочки») во все стороны летят!

Если копнуть глубже, то неожиданно обнаружится множество ассоциаций, усиливающих законность нового значения слова «хак». И реле, и перфоратор издают повторяющиеся монотонные удары, чем-то напоминающие кашель, а hack именно это и обозначает (выражение «кашлять сухим кашлем» — одно из его значений). К тому же, программировали «динозавров» исключительно в машинных кодах, подчас с помощью переключателей или перетыкивания разъемов, — физически тяжелая, нудная, неблагодарная работа, достающаяся наименее привилегированной части персонала. Какой там романтизм? Какое изящество решений или полет мысли? Халтура сплошная... Редкая программа обходится без ошибок, а программа, составленная в машинных кодах, — тем более. При желании любого оператора можно было назвать халтурщиком — хакером в ругательном смысле этого слова. «Вот, наделал кучу ошибок, хакер ты наш!»

Обыватели же, далекие от вычислительной техники и знакомые с ней исключительно по фантастическим романам, испытывали перед ЭВМ благоговейное уважение, подогреваемое гордостью за научно-технические достижения всего рода *homo sapiens* в целом и американской нации в частности. «Белые воротнички» — цвет нации, управляющие машиной размером с супермаркет и стоящей дороже тысячи таких супермаркетов, вызывали у рядового американца смесь восторга, зависти и стремления к подражанию. Вроде как «я тоже хочу быть космонавтом», не задумываясь о том, что космонавтика — это только с виду романтика, а в действительности каторжная работа.

Но если желание побывать в космосе до сих пор смогли реализовать лишь единицы, то ЭВМ стали широко доступными уже в начале шестидесятых годов. К тому времени их можно было встретить и в подвалах университетов, и в стенах крупных корпораций, и практически во всех исследовательских учреждениях. Сесть за пульт ЭВМ в сознании студента означало практически то же самое, что сесть за штурвал реактивного бомбардировщика. Программирование ассоциировалось отнюдь не с батрачеством, а с интеллектуальной игрой. И старшие наставники студентов — операторы ЭВМ — были не только их руководителями, но и кумирами. Студенты, одержимые вычислительной техникой, стремились во всем подражать персоналу, обслуживающему большие ЭВМ, часто без понимания сути происходящего. Услышав жаргонное прозвище операторов, студенты, не догадываясь о его иронично-оскорбительном оттенке, с достоинством стали называть хакерами и себя, и своих товарищей, и даже свою работу окрестили хакерством. В их устах слово «хакер» звучало отнюдь не насмешкой, а расценивалось как титул. Ты — хакер, значит, ты такой же мастер, как и настоящий оператор ЭВМ, значит, ты крутой парень и перед тобой другим не стыдно снять шляпу.

Так хакеры из работяг превратились в программистов-энтузиастов, помешанных на компьютерах и выделяющих на них такое... такое, что другим и не снилось. Значение термина «hack» все более изменялось в сторону «крутого трюка», «забавного эффекта», «выполненного со вкусом розыгрыша». Этот дух подхватили и другие факультеты, порой вовсе не связанные ни с электроникой, ни с вычислительной техникой, ни даже с точными науками вообще. Хаком стали называть любой классный розыгрыш или нестандартное решение знакомой задачи — жаргонный термин технического языка превратился в модерновое словечко, употребляемое всеми, кому не лень.

Тем временем мутация понятия «хакер» продолжалась... Чтобы понять ее причины, придется мысленно перенестись в конец шестидесятых — начало семидесятых, а может, даже чуточку позже. В те времена среди западной молодежи витал дух борьбы. Борьбы с кем? Да разве это важно! Протестовали против войны во Вьетнаме (кто не хотел служить в армии — жгли повестки), ломали пуританские устои старого мира, провозглашая свободу любви, презирали деньги (или только делали вид, что презирали, завистливо поглядывая в сторону того, у кого они есть). По большому счету вся борьба сводилась к суеде в песочнице, и власть имущих в общем-то ничуть не беспокоила. Молодежные лидеры не имели в своих руках никакого оружия — ни огнестрельного, ни политического, ни экономиче-

ского, ни идеологического. К тому же через десяток лет дух борьбы в Америке иссяк, весь шум сошел на нет.

Счастливое исключение составили программисты. В те дни компьютерные системы еще не успели обзавестись достойной защитой, но уже управляли важными стратегическими и экономическими объектами. Власть над компьютерами позволяла дать хорошего пинка и правительственным организациям, и финансовым магнатам, и корпорациям, и другим сильным мира сего, причем оставаясь безнаказанным. Не существовало ни соответствующих законов, ни компьютерной полиции, способной вычислить преступника... Словом, Дикий Запад времен разбоя, романтики и беспредела, когда человек с кольтом мог заставить шерифа любого уездного городка «слушать Шопена лежа». У американцев, надо сказать, по поводу освоения Америки очень сильный комплекс — одних вестернов они сняли больше, чем мы фильмов про Великую Отечественную войну. Понятное дело, каждый юный американец в душе мнит себя полноправным ковбоем!

Компьютеры же позволили воплотить эту мечту в жизнь. Освой ЭВМ и носись по электронным сетям как неуловимый Джо, отстреливающий индейцев (банкиров, агентов ЦРУ и т. д.). Да и как не носиться, когда на книжных лотках как грибы появлялись фантастические романы, главными героями которых были компьютерные взломщики — хакеры. Писатели, никогда в жизни не видевшие ЭВМ, плохо разбирались в техническом жаргоне и употребляли его на интуитивно-бессознательном уровне безо всякого понимания. Достаточно перелистать «The Shockware Rider» Джона Бруннера (John Brunner) издания 1975 года, «The Adolescence of P-1» Томаса Риана (Thomas Ryan) издания 1977 года или «Neuromancer» Вильяма Гибсона (William Gibson), опубликованный в 1984 году, чтобы убедиться, насколько бесконечно их авторы были далеки от вычислительной техники. Впрочем, литературных достоинств произведений это ничуть не умаляло, и у читателей сложился устойчивый образ: ЭВМ — это круто, а хак — это вообще круто. «Нейроматик», кстати, был самой любимой книгой Роберта Тапплана Морриса, создавшего свой знаменитый вирус-червь, надо полагать, не без влияния Вильяма Гибсона.

Журналисты, не обремененные ни знаниями ЭВМ, ни лингвистическим образованием, из всего этого поняли только одно: некто, называющие себя хакерам, ломают компьютеры по всей стране, причем ломают весьма круто, с нанесением ущерба в особо крупных размерах.

Слово «хакер» вырвалось на страницы газет, но в широких массах глагол «hack» по-прежнему означал все то же «бить», «кромсать», и американцы, вполне естественно, заключили, что хакер — это тот, кто вламывается в чужие системы и разбивает их в пух и прах.

Вот, собственно, и все... Кольцо замкнулось — термин «хакер» вернул свое историческое значение, но не прекратил эволюцию! Хакерам прошлого поколения (т. е. энтузиастам программирования) очень не понравилось, что их титул смешали, мягко выражаясь, с дерьмом, и при его упоминании все стали шарахаться от них, как от огня. Дабы реабилитироваться в глазах общественности, хакеры предприняли попытку разделить всех своих на «хороших» и «плохих», оставив за хорошими парнями право называться хакерами, а для плохих придумав специаль-

ный термин «кракер» — от слова сгаск — ломать (кстати, почему не «брейкер» — от слова break?), в буквальном смысле обозначающий «ломатель». Затея с треском провалилось — далеко не каждый взломщик был готов нацепить на себя ярлык плохого парня. Называться хакером по-прежнему считалось и модно, и престижно, пускай все «хакерство» ограничилось wappahe (в дословном русском переводе «хочубытькак», т. е. подражательством). Предметы хакерской культуры обожествлялись, становясь предметом поклонения, фетишем, иконой на стене.

Эта ветка генеалогического древа «хакеров» не имеет будущего и обречена на медленное, но неотвратимое вымирание. Уже сегодня, в начале первого десятилетия XXI века, термин «хакер» стал всеобъемлющим и утратил всякий смысл. Кто пишет вирусы? Хакеры! Кто ломает программы? Хакеры! Кто крадет деньги из банков? Хакеры! Кто пакостит в сети? Хакеры! Кто программирует на ассемблере? Хакеры! Кто знает все тонкости операционной системы и «железа»? Хакеры! Сказать собеседнику, что ты хакер, не уточнив, что конкретно ты имеешь под этим в виду, все равно что ничего не сказать.

Термин «хакер» умер, но ведь хакеры остались! Остались и работяги-кодеры, пускай уже не клацающие реле, но зато шумящие пропеллерами вентиляторов, остались и энтузиасты программирования, упоенно программирующие и на древних, и на современных языках, остались и исследователи защит, и умельцы по их взлому... Люди есть, а термина, определяющего их принадлежность, уже нет.

Почему бы не назвать определенную категорию компьютерщиков кодокопателями? Этот термин, впервые употребленный Безруковым, на мой взгляд, очень удачен и интуитивно понятен без дополнительных объяснений. Любой, кто любит копаться в коде (не обязательно машинном), по праву может считать себя кодокопателем.

Таким людям, собственно, и посвящена эта книга...

Чем мы будем заниматься

На протяжении всей книги мы будем заниматься увлекательной интеллектуальной игрой — созданием защитных механизмов и исследованием их стойкости. Скажу сразу — ничего общего со взломом коммерческих программ или кражей денег из банка это занятие не имеет. Автор искренне надеется, что его читатели — граждане в своей массе законопослушные и высоконравственные.

Умение нейтрализовать защиты не дает права применять это умение в преступных целях. Какие же цели являются преступными, а какие нет — вопрос, относящийся уже не к хакерству, а к юриспруденции, в которой автор не силен, и все, что он может порекомендовать — при возникших у вас сомнениях в правомерности совершения некоторых действий обратитесь к юристам.

Однако экспериментировать с вашей личной интеллектуальной собственностью — программами, написанными вами самими, — ни один закон не вправе запретить, да ни один закон этого, собственно, и не запрещает.

А раз так, рюкзак на плечи, охотничий ножик в карман и — в густой таежный лес...

Что нам понадобится

Выбор рабочего инструментария — дело сугубо личное и интимное. Тут на вкус и цвет товарищей нет. Поэтому примите все нижесказанное не как догму, а как рекомендацию к действию. Итак, для чтения книги нам понадобится:

- отладчик **Soft-Ice** версии 3.25 или более старший;
- дизассемблер **IDA** версии 3.7х (рекомендуется 3.8, а еще лучше 4.x);
- *HEX-редактор* **HIEW** любой версии;
- любой C\C++ и Pascal-компилятор по вкусу (в книге подробно описываются особенности компиляторов Microsoft Visual C++, Borland C++, WATCOM C, GNU C, FreePascal, а за основу взят Microsoft Visual C++ 6.0);
- пакеты **SDK** и **DDK** (последний не обязателен, но очень желателен);
- *операционная система* — любая из семейства Windows, но настоятельно рекомендуется **Windows 2000**.

Теперь обо всем этом подробнее.

Soft-Ice. Отладчик Soft-Ice — основное оружие хакера. Хотя с ним конкурируют бесплатные windeb от Microsoft и TRW от LiuTaoTao, Soft-Ice намного лучше и удобнее их всех, вместе взятых. Для наших экспериментов подойдет практически любая версия Айса, например, автор использует давно апробированную и устойчиво работающую версию 3.26, замечательно уживающуюся с Windows 2000. Новомодная версия 4.x не очень-то дружит с видеоадаптером автора (Matrox Millennium G450, для справки) и вообще временами «едет крышей». К тому же из всех новых возможностей четвертой версии полезна лишь поддержка FPO (*Frame point omission* — см. главу «Идентификация локальных стековых переменных») — локальных переменных, напрямую адресуемых через регистр ESP, — бесспорно полезно, но без этого можно и обойтись. Найти Soft-Ice можно на дисках известного происхождения или у российского дистрибьютора: <http://www.quarta.ru/bin/soft/winntutils/softicent.asp?ID=59>. Купите, не пожалеете (хакерство — это ведь не то же самое что пиратство, и честность еще никто не отменял).

IDA Pro. Бесспорно, самый мощный дизассемблер в мире — это IDA. Прожить без нее, конечно, можно, но... нужно ли? IDA обеспечивает удобную навигацию по исследуемому тексту, автоматически распознает библиотечные функции и локальные переменные, в том числе и адресуемые через регистр ESP, поддерживает множество процессоров и форматов файлов. Одним словом, хакер без IDA — не хакер. Впрочем, агитации излишни, единственная проблема: где же эту IDA взять? На пиратских дисках она встречается крайне редко (самая последняя виденная автором версия — 3.74, да и то нестабильно работающая), на сайтах в Интернете — еще реже. Фирма-разработчик жестко пресекает любые попытки несанкционированного распространения своего продукта, и единственный надежный путь его приобретения — покупка в самой фирме или у российского дистрибьютора (*GelioSoft Ltd* <gav@geliosoft.mtu-net.ru>). К сожалению, с дизассемблером не

распространяется никакой документации (не считая встроенного хелпа — очень короткого и бессистемного), поэтому автору ничего не остается, как порекомендовать собственный трехтомник *«Образ мышления — дизассемблер IDA»*, подробно рассказывающий и о самой IDA, и о дизассемблировании вообще.

HEW. «Хьюев» — это не только HEX-редактор, но и дизассемблер, ассемблер и крипт «в одном флаконе». Он не избавит от необходимости приобретения IDA, но с лихвой заменит IDA в ряде случаев (IDA очень медленно работает, и обидно тратить кучу времени, если все, что нам нужно, посмотреть на препарируемый файл одним глазком). Впрочем, основное назначение «хьюева» отнюдь не дизассемблирование, а *bit hack* — небольшое хирургическое вмешательство в двоичный файл — обычное вырезание жизненно важного органа защитного механизма, без которого он перестает работать.

SDK (Software Development Kit — комплект прикладного разработчика). Из пакета SDK нам в первую очередь понадобится документация по Win32 API и утилита для работы с PE-файлами BUMPBIN. Без документации ни хакерам, ни разработчикам никак не обойтись. По крайней мере, необходимо знать прототипы и назначение основных функций системы. Эту информацию в принципе можно почерпнуть и из многочисленных русскоязычных книг по программированию, но ни одна из них не может похвастаться полнотой и глубиной изложения. Поэтому рано или поздно, но все-таки придется обращаться к SDK. Правда, некоторым придется плотно засесть за английский, поскольку вся документация написана именно на этом языке и ждать ее перевода — все равно что караулить у моря погоду (правда, с некоторых пор на сайте Microsoft стало появляться много информации для разработчиков и на русском языке). Где приобрести SDK? Во-первых, SDK входит в состав MSDN, а сам MSDN ежеквартально издается на компакт-дисках и распространяется по подписке (подробнее об условиях его приобретения можно узнать на официальном сайте msdn.microsoft.com). Прилагается MSDN и к компилятору Microsoft Visual C++ 6.0, но, увы, далеко не первой свежести. Впрочем, пользоваться им вполне можно, во всяком случае, для чтения данной книги его будет вполне достаточно.

DDK (Driver Development Kit — комплект разработчика драйверов). Какую пользу может извлечь хакер из пакета DDK? Ну, в первую очередь он поможет разобраться, как устроены, работают и ломаются драйверы. Помимо основополагающей документации и множества примеров в него входит очень ценный файл *NTDDK.h*, содержащий определения большинства недокументированных структур и буквально нашпигованный комментариями, раскрывающими некоторые любопытные подробности функционирования системы. Нелишним будет и инструментарий, прилагающийся к DDK. Среди прочего, сюда входит и отладчик windeb. Весьма неплохой, кстати, отладчик, но все же значительно уступающий Soft-Ice, поэтому и не рассматриваемый в данной книге (но если вы не найдете Айса — сгодится и windeb). Более полезным окажется ассемблер MASM, на котором, собственно, и пишутся драйверы, а также другие полезные программки, облегчающие жизнь хакеру. Последнюю версию DDK можно **бесплатно** скачать

с сайта Microsoft, только имейте в виду, что для NT полный DDK занимает свыше 40 мегабайтов в упакованном виде, и еще больше требует места на диске.

Операционная система. Вовсе не собираясь навязывать читателю собственные вкусы и пристрастия, автор тем не менее настоятельно рекомендует установить именно **Windows 2000**. Мотивация — это действительно стабильная и устойчиво работающая операционная система, мужественно переносящая критические ошибки приложений. Специфика работы хакера такова, что хирургические вмешательства в недра программ часто срывают им «крышу», доводя ломаемое приложение до буйного помешательства с непредсказуемым поведением. Windows 9x, демонстрируя социалистическую солидарность, очень часто «ложится» рядом с зависшей программой. Порой компьютер приходится перезагружать не один десяток раз на дню! И хорошо если только перезагружать, а не восстанавливать разрушенные сбоем диски (такое хотя и редко, но случается). Завесить же Windows 2000 на порядок сложнее — автору это «удается» не чаще одного-двух раз в месяц, да и то с недосыпу или по небрежности. Кроме того, Windows 2000 позволяет загружать Soft-Ice в любой момент без необходимости перезагрузки, что очень удобно! Наконец, весь материал этой книги рассчитан именно на Windows 2000 — а ее отличия от других систем упоминаются далеко не всегда. Все равно все мы когда-нибудь перейдем на Windows 2000 и забудем о Windows 9x как о страшном сне, так стоит ли хвататься за эту умирающую платформу? К слову сказать, Windows Me — это не то же самое, что Windows 2000, и ставить Me на свой компьютер автор никому не рекомендует (такое впечатление, что Windows Me вообще не тестировали, а о том, что ее писали садисты — кто ставил, тот поймет, — автор вообще молчит).

Худо-бедно разобравшись с инструментарием, поговорим о сером веществе, ибо в его отсутствии весь собранный инструмент бесполезен. Автор полагает, что читатель уже знаком с ассемблером, и если не пишет программ на этом языке, то по крайней мере представляет себе, что такое регистры, сегменты, машинные инструкции и т. д. В противном случае эта книга рискует показаться чересчур сложной и непонятной. Отыщите в магазине любой учебник по ассемблеру (например, В. Юрова «*ASSEMBLER — учебник*», П. И. Рудакова «*Программируем на языке ассемблера IBM PC*» или «*Assembler — язык неограниченных возможностей*» С. В. Зубкова) и основательно проштудируйте его.

Помимо знания ассемблера также надо иметь хотя бы общие понятия о функционировании операционной системы. Купите и вдумчиво изучите (если не сделали этого до сих пор) «*Windows для профессионалов*» Джеффри Рихтера¹ и (если найдете) «*Секреты системного программирования в Windows 95*» Мэта Питрека. Хотя его книга посвящена Windows 95, частично она справедлива и для Windows 2000. Для знакомства с архитектурой самой же Windows 2000 рекомендуется ознакомиться с шедевром Хелен Кастер «*Основы Windows NT*» и брошюрой «*Недокументированные возможности Windows NT*» А. В. Коберниченко.

¹ См. «Приложение», «Ошибки Джеффри Рихтера».

Что касается общей теории информатики и алгоритмов — бесспорный авторитет Кнут. Впрочем, на вкус автора, монография М. Броя «*Информатика*» куда лучше — при том, что она намного короче, круг охватываемых ею тем и глубина изложения намного шире. Зачем хакеру теория информатики? Да куда же без нее! Вот, скажем, встретится ему защита со встроенным эмулятором машины Тьюринга. С лету ее не сломать — надо как минимум опознать сам алгоритм: что это вообще такое — Тьюринг, Марков, сеть Петри, а затем отобразить его на язык высокого уровня, дабы в удобочитаемом виде анализировать работу защиты. Куда же тут без теории информатики!

Засим все. Ну, разве что стоит дополнить наш походный рюкзачок парой учебников по английскому (они пригодятся, поверьте) и выкачать с сайтов Intel и AMD всю имеющуюся там документацию по процессорам. На худой конец подойдет и ее русский перевод, например, А. А. Ровдо «*Микропроцессоры от 8086 до Pentium III Xeon и AMD K6-3*». Ну-с, рюкзачок на плечо и в путь...

Знакомство с базовыми приемами работы хакера

Введение

Классификация защит

Стать хакером очень просто. Достаточно выучить и понять: математический анализ, теорию функций комплексного переменного, алгебру, геометрию, теорию вероятностей, математическую статистику, математическую логику и дискретную математику...

Б. Леонтьев. Хакеры & Internet

Проверка **аутентичности** (от греч. *authentikos* — подлинный) — «сердце» подавляющего большинства защитных механизмов. Должны же мы удостовериться: то ли лицо, за которое оно себя выдает, работает с программой, и разрешено ли этому лицу работать с программой вообще!

В качестве «лица» может выступать не только пользователь, но и его компьютер или носитель информации, хранящий лицензионную копию программы. Таким образом, все защитные механизмы можно разделить на две основные категории:

- защиты, основанные на **знании** (пароля, серийного номера);
- защиты, основанные на **обладании** (ключевого диска, документации).

Защиты, основанные на знании, бесполезны, если обладатель защищенной с их помощью программы не заинтересован в сохранении ее секретности. Он может сообщить пароль (серийный номер) кому угодно, после чего любой сможет запустить такую программу на своем компьютере. Поэтому парольные защиты для предотвращения пиратского копирования программ непригодны. Почему же тогда практически все крупные производители в обязательном порядке используют серийные номера? Ответ прост — для защиты своей интеллектуальной собственности от грубой физической силы. Происходит это приблизительно так: работая тишина такой-то фирмы внезапно нарушается топотом парней в камуфляже, сверяющих лицензионные номера Windows (Microsoft Office, Microsoft Visual Studio) с лицензионными соглашениями, и стоит обнаружиться хотя бы одной «левой» копии, как появившийся словно из-под земли сотрудник фирмы начинает радостно потирать руки в предвкушении дождя вечнозеленых... В лучшем случае — заставят купить все «левые» копии, в худшем же...

К домашним пользователям в квартиру, понятное дело, никто не врывается — частная собственность и все такое, да к этому никто, впрочем, и не стремится

ся. Что с домашнего пользователя возьмешь-то? К тому же самим фирмам выгодно массовое распространение их продукции, а кто его обеспечит лучше пиратов? Но и здесь серийные номера не лишние — они разгружают службу технической поддержки от «левых» звонков незарегистрированных пользователей, одновременно с этим склоняя последних к покупке легальной версии.

Такая схема защиты идеальна для корпораций-гигантов, но она не подходит для мелких программистских коллективов и индивидуальных разработчиков, особенно если они зарабатывают на жизнь написанием узкоспециализированных программ с ограниченным рынком сбыта (скажем, анализаторов звездных спектров или системы моделирования ядерных реакций). Не имея достаточного влияния, раскатать сотрудников известных органов на облаву для проверки лицензионности своего ПО нереально, а выбить деньги из нелегальных пользователей можно разве что с помощью криминальных структур, да и то навряд ли. Вот и приходится рассчитывать лишь на собственную силу и смекалку.

Тут лучше подходит тип защит, основанных на обладании некоторым уникальным предметом, скопировать который чрезвычайно тяжело, а в идеале вообще невозможно. Первые ласточки этой серии — ключевые дискеты, записанные с таким расчетом, чтобы при их копировании копия чем-нибудь да отличалась от оригинала. Самое простое (но не самое лучшее) — слегка изуродовать дискету гвоздем (шилом, перочинным ножом), а затем, определив местоположение дефекта относительно сектора (это можно сделать записью-чтением некоторой тестовой информации — до какого-то момента чтение будет идти нормально, а потом начнется «мусор»), жестко прописать его в программе и при каждом запуске проверять, на том же самом месте дефект или нет. Когда же дискеты вышли из употребления, эта же техника была адаптирована и для компакт-дисков. Кто побогаче, уродует их лазером, кто победнее — все тем же шилом или гвоздем.

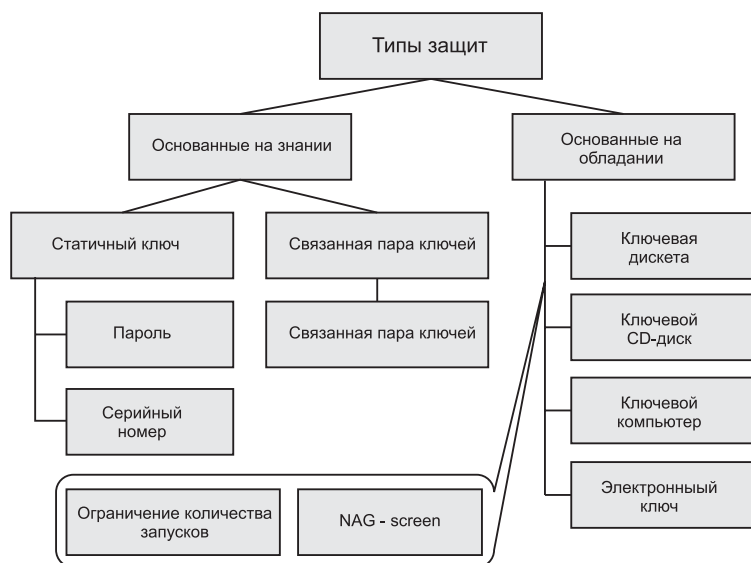


Рис. 1. Основные типы защит

Таким образом, программа жестко привязана к диску (дискете) и требует ее присутствия для своей работы, а поскольку скопировать такой диск нереально (попробуй-ка добиться идентичных дефектов на копиях), пираты остаются не у дел.

Защитные механизмы, основанные на обладании, часто модифицируют предмет обладания в процессе работы, ограничивая количество запусков программы или время ее использования. Особенно часто такой прием используется в инсталляторах: чтобы не нервировать пользователя, ключ запрашивается лишь однажды — на стадии установки программы, а работать с ней можно и без него. Если количество инсталляций ограничено, ущербом от несанкционированных установок одной копии программы на несколько компьютеров можно пренебречь.

Единственная проблема — все это ущемляет права легального пользователя. Кому понравится ограничение на количество инсталляций? (А ведь некоторые люди переустанавливают систему и все ПО буквально каждый месяц, а то и несколько раз на дню.) Ключевые диски распознаются не всеми типами приводов, зачастую «невидимы» по сети, а если защитный механизм для увеличения стойкости к взлому обращается к оборудованию напрямую, в обход драйверов, то такая программа наверняка не будет функционировать под Windows NT\2000 и, весьма вероятно, откажет в работе под Windows 9x (если, конечно, она не была заранее спроектирована соответствующим образом, но если так — тем хуже, ибо некорректно работающая защита, исполняющаяся с наивысшими привилегиями, может причинить немалый урон системе). Помимо этого, ключевой предмет можно потерять, его могут украсть, да и сам он может выйти из строя (дискеты склонны сыпаться и размагничиваться, диски — царапаться, а электронные ключи — «сгорать»).

Конечно, эти претензии относятся к качеству реализации, а не к идее ключей вообще, но конечным пользователям от этого ничуть не легче! Если же защита создает неудобства, у пользователей появляется очень сильная мотивация к посещению ближайшего доступного пирата на предмет приобретения у него контрафактного программного обеспечения. И никакие разговоры о морали, этике, добропорядочности и т. д. не подействуют — своя рубашка ближе к телу, а о добропорядочности в первую очередь нужно задуматься разработчикам таких защит. Тов... тьфу, господа, не отравляйте жизнь пользователям! Пользователи тоже люди!

В последнее время наибольшую популярность обрели защиты, основанные на регистрационных номерах. При первом запуске программа привязывается к компьютеру и включает «счетчик» (вариант — блокирует некоторые функциональные возможности). А чтобы ее «освободить», необходимо ввести пароль, сообщаемый разработчиком за некоторое материальное вознаграждение. Часто для предотвращения пиратского копирования пароль представляет собой некоторую производную от ключевых параметров компьютера (или производную от имени пользователя в простейшем случае).

Разумеется, в этом кратком обзоре типов защит очень многое осталось за кадром, но подробный разговор о классификации защит выходит за рамки этой книги, так что отложим его до второго тома.

Философия стойкости

Однажды один из друзей сказал Катону Старшему: «Какое безобразие, что в Риме тебе до сих пор не воздвигли памятника! Я обязательно позабочусь об этом».

«Не надо, — ответил Катон, — я предпочитаю, чтобы люди спрашивали, почему нет памятника Катону, чем почему он есть».

Т. Мессон

Если защита базируется на одном лишь предположении, что ее код не будет изучен и/или изменен, — это плохая защита. Отсутствие исходных текстов отнюдь не является непреодолимым препятствием для изучения и модификации приложения. Современные технологии обратного проектирования позволяют автоматически распознавать библиотечные функции, локальные переменные, стековые аргументы, типы данных, ветвления, циклы и т. д. А в недалеком будущем дизассемблеры, вероятно, вообще научатся генерировать листинги, близкие по внешнему виду к языкам высокого уровня.

Но даже сегодня трудоемкость анализа двоичного кода не настолько велика, чтобы надолго остановить злоумышленников. Огромное количество постоянно совершаемых взломов — лучшее тому подтверждение. В идеальном случае знание алгоритма работы защиты не должно влиять на ее стойкость, но это достижимо далеко не всегда. Например, если разработчик серверной программы решит установить в демонстрационной версии ограничение на количество одновременно обрабатываемых соединений (как часто и случается), злоумышленнику достаточно найти инструкцию процессора, осуществляющую такую проверку и удалить ее. Модификации программы можно воспрепятствовать постоянной проверкой ее целостности, но опять-таки, код, проверяющий целостность, может быть найден и удален.

Сколько бы уровней защиты ни существовало, один или миллион, программа *может* быть взломана! Это только вопрос времени и усилий. Но в отсутствие реально действующих законов защиты интеллектуальной собственности разработчикам приходится больше полагаться на стойкость своей защиты, чем на помощь правоохранительных органов. Бытует мнение, что если затраты на нейтрализацию защитного механизма будут не ниже стоимости легальной копии, ее никто не будет ломать. Это неверно! Материальный стимул — не единственное, что движет хакером. Гораздо более сильной мотивацией оказывается *интеллектуальная борьба* (кто умнее: я или автор защиты?), *спортивный азарт* (кто из хакеров сломает больше всего защит?), *любопытство* (а как это работает?), *повышение своего профессионализма* (чтобы научиться создавать защиты, сначала нужно научиться их снимать), да и просто *интересное времяпровождение* (если его нечем занять). Многие люди могут неделями корпеть над отладчиком, снимая защиту с программы стоимостью в несколько долларов, а то и вовсе распространяемой бесплатно (например, файл-менеджер FAR для жителей России и СНГ абсолютно бесплатен, но это не спасает его от взлома).

Целесообразность защиты ограничивается конкуренцией, при прочих равных условиях клиент всегда выбирает незащищенный продукт, даже если защита не ущемляет его прав. В настоящее время спрос на программистов значительно превышает предложение, но в отдаленном будущем разработчикам придется либо сговориться, либо полностью отказаться от защит. И специалисты по защитам будут вынуждены искать себе другую работу.

Это не значит, что данная книга бесполезна, напротив, полученные знания следует применить как можно быстрее, пока в защитах еще не отпала необходимость.

Шаг первый. Разминочный

Бороться со своими мыслями — это уподобиться одному глупцу, который в целях аккуратности и гигиены решил больше не какать. День не какал, два не какал. Потом, конечно, не выдержал, но всех продолжал уверять, что не какает.

Аноним

Алгоритм простейшего механизма аутентификации состоит в посимвольном сравнении введенного пользователем пароля с эталонным значением, хранящимся либо в самой программе (как часто и бывает), либо вне ее, например, в конфигурационном файле или реестре (что встречается реже).

Достоинство такой защиты — крайне простая программная реализация. Ее ядро состоит фактически из одной строки, которую на языке C можно записать так: *if (strcmp(введенный пароль, эталонный пароль)) { /* Пароль неверен */ } else { /* Пароль ОК */ }*

Давайте дополним этот код процедурами запроса пароля и вывода результатов сравнения, а затем испытаем полученную программу на прочность, т. е. на стойкость к взлому:

Листинг 1. Пример простейшей системы аутентификации

```
// Простейшая система аутентификации
// посимвольное сравнение пароля
#include <stdio.h>
#include <string.h>

#define PASSWORD_SIZE 100
#define PASSWORD      "myG00Dpassword\n"
// этот перенос нужен затем, чтобы ^^^^
// не выкусывать перенос из строки,
// введенной пользователем

int main()
{
    // Счетчик неудачных попыток аутентификации
    int count=0;
```

```
// Буфер для пароля, введенного пользователем
char buff[PASSWORD_SIZE];

// Главный цикл аутентификации
for(;;)
{
    // Запрашиваем и считываем пользовательский
    // пароль
    printf("Enter password:");
    fgets(&buff[0], PASSWORD_SIZE, stdin);

    // Сравниваем оригинальный и введенный пароль
    if (strcmp(&buff[0], PASSWORD))
        // Если пароли не совпадают - "ругаемся"
        printf("Wrong password\n");
        // Иначе (если пароли идентичны)
        // выходим из цикла аутентификации
    else break;

    // Увеличиваем счетчик неудачных попыток
    // аутентификации и, если все попытки
    // исчерпаны, завершаем программу
    if (++count>3) return -1;
}

// Раз мы здесь, то пользователь ввел правильный пароль
printf("Password OK\n");
}
```

В популярных кинофильмах крутые хакеры легко проникают в любые жутко защищенные системы, каким-то непостижимым образом угадывая искомый пароль с нескольких попыток. Почему бы не попробовать пойти их путем?

Не так уж редко пароли представляют собой осмысленные слова наподобие «Ferrari», «QWERTY», имена любимых хомячков, названия географических пунктов и т. д. Угадывание пароля сродни гаданию на кофейной гуще — никаких гарантий на успех нет, остается рассчитывать на одно лишь везение. А удача, как известно, птица гордая — палец в рот ей не кладут. Нет ли более надежного способа взлома?

Давайте подумаем. Раз эталонный пароль хранится в теле программы, то, если он не зашифрован каким-нибудь хитрым образом, его можно обнаружить тривиальным просмотром двоичного кода программы. Перебирая все встретившиеся в ней текстовые строки, начиная с тех, что более всего смахивают на пароль, мы очень быстро подберем нужный ключ и откроем им программу! Причем область просмотра можно существенно сузить — в подавляющем большинстве случаев компиляторы размещают все инициализированные переменные в сегменте данных (в PE-файлах он размещается в секции .data). Исключение составляют, пожалуй, ранние багдадские (Borland'вые) компиляторы с их маниакальной любовью всовывать текстовые строки в сегмент кода — непосредственно по месту их вызова. Это упрощает сам компилятор, но порождает множество проблем. Современные операционные системы, в отличие от старушки MS-DOS, запрещают модификацию кодового сегмента, и все размещен-

ные в нем переменные доступны лишь для чтения. К тому же на процессорах с отдельной системой кэширования (на тех же Pentium'ax, например) они «засоряют» кодовый кэш, попадая туда при упреждающем чтении, но при первом же обращении к ним вновь загружаются из медленной оперативной памяти (кэша второго уровня) в кэш данных. В результате — тормоза и падение производительности.

Что ж, пусть это будет секция данных! Остается только найти удобный инструмент для просмотра двоичного файла. Можно, конечно, нажать клавишу <F3> в своей любимой оболочке (FAR, DOS Navigator) и, придавив кирпичом клавишу <Page Down>, любоваться бегущими цифирьками до тех пор, пока не надоест.

Можно воспользоваться любым hex-редактором (QVIEW, HIEW...) — кому какой по вкусу, — но в книге по соображениям наглядности автор приведет результат работы утилиты DUMPBIN из штатной поставки Microsoft Visual Studio.

Попросим ее распечатать секцию данных (ключ /SECTION:.data) в «сыром» виде (ключ /RAWDATA:BYTES), указав значок «>» для перенаправления вывода в файл (ответ программы занимает много места, и на экране помещается один лишь «хвост»).

```
> dumpbin /RAWDATA:BYTES /SECTION:.data simple.exe >filename
```

```
RAW DATA #3
```

```
00406000: 00 00 00 00 00 00 00 00 00 00 00 00 3B 11 40 00 .....;.@.
00406010: A4 40 40 00 00 00 00 00 00 00 00 00 E0 11 40 00 д@@.....p.@.
00406020: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
00406030: 45 6E 74 65 72 20 70 61 73 73 77 6F 72 64 3A 00 Enter password:.
00406040: 6D 79 47 4F 4F 44 70 61 73 73 77 6F 72 64 0A 00 myG00Dpassword..
                                                ~~~~~~
00406050: 57 72 6F 6E 67 20 70 61 73 73 77 6F 72 64 0A 00 Wrong password..
00406060: 50 61 73 73 77 6F 72 64 20 4F 4B 0A 00 00 00 00 Password OK....
00406070: 40 6E 40 00 00 00 00 00 40 6E 40 00 01 01 00 00 @n@.....@n@.....
```

Смотрите! Среди всего прочего тут есть одна строка, до боли похожая на эталонный пароль (в тексте она выделена жирным шрифтом). Испытаем ее? Впрочем, какой смысл — судя по исходному тексту программы это действительно искомый пароль, открывающий защиту, словно золотой ключик. Слишком уж видное место выбрал компилятор для его хранения — пароль не мешало бы запрятать и получше.

Один из способов сделать это — насильно поместить эталонный пароль в собственноручно выбранную нами секцию. Такая возможность не предусмотрена стандартом, и потому каждый разработчик компилятора (строго говоря, не компилятора, а линкера, но это не суть важно) волен реализовывать ее по-своему или не реализовывать вообще. В Microsoft Visual C++ для этой цели предусмотрена специальная прагма **data_seg**, указывающая, в какую секцию помещать следующие за ней инициализированные переменные. Неинициализированные переменные по умолчанию располагаются в секции **.bss** и управляются прагмой **bss_seg** соответственно.

Добавим в листинг 1 следующие строки и посмотрим, что у нас из этого получится.

```
int count=0;
// С этого момента все инициализированные переменные будут
// размещаться в секции .kprnc
#pragma data_seg(.kprnc)    // точку перед именем ставить
                             // не обязательно - просто так
                             // принято

char passwd[]=PASSWORD;
#pragma data_seg()
// Теперь все инициализированные переменные вновь будут
// размещаться в секции по умолчанию, т.е. .data
char buff[PASSWORD_SIZE]="";
...
if (strcmp(&buff[0], &passwd[0]))

> dumpbin /RAWDATA:BYTES /SECTION:.data simple2.exe >filename

RAW DATA #3
00406000: 00 00 00 00 00 00 00 00 00 00 00 00 9B 11 40 00 .....bl.@.
00406010: 04 41 40 00 00 00 00 00 00 00 00 00 40 12 40 00 .A@.....@.@.
00406020: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
00406030: 45 6E 74 65 72 20 70 61 73 73 77 6F 72 64 3A 00 Enter password:.
00406040: 57 72 6F 6E 67 20 70 61 73 73 77 6F 72 64 0A 00 Wrong password..
00406050: 50 61 73 73 77 6F 72 64 20 4F 4B 0A 00 00 00 00 Password OK.....
00406060: 20 6E 40 00 00 00 00 00 20 6E 40 00 01 01 00 00 n@..... n@.....
00406070: 00 00 00 00 00 00 00 00 00 00 10 00 00 00 00 00 .....
```

Ага, теперь в секции данных пароля нет и хакеры «отдыхают»! Но не спешите с выводами. Давайте сначала выведем на экран список всех секций, имеющих в файле:

```
> dumpbin simple2.exe
```

```
Summary
 2000 .data
 1000 .kprnc
      ****
 1000 .rdata
 4000 .text
```

Нестандартная секция **.kprnc** сразу же привлекает к себе внимание. А ну-ка посмотрим, что там в ней?

```
dumpbin /SECTION:.kprnc /RAWDATA simple2.exe
```

```
RAW DATA #4
00408000: 6D 79 47 4F 4F 44 70 61 73 73 77 6F 72 64 0A 00 myG00Dpassword..
                                     ~~~~~~
```

Вот он, пароль! Спрятали, называется... Можно, конечно, извратиться и засунуть секретные данные в секцию неинициализированных данных (.bss), служебную RTL-секцию (.rdata) или даже секцию кода (.text) — не все там догадаться поискать, а работоспособность программы такое размещение не нарушит. Но не

стоит забывать о возможности автоматизированного поиска текстовых строк в двоичном файле. В какой бы секции ни содержался эталонный пароль, фильтр без труда его найдет (единственная проблема — определить, какая из множества текстовых строк представляет собой искомый ключ; возможно, потребуется перебрать с десяток-другой потенциальных «кандидатов»).

Правда, если пароль записан в уникоде, его поиск несколько осложняется, так как не все утилиты поддерживают эту кодировку, но надеяться, что это препятствие надолго задержит хакера, несколько наивно.

Шаг второй. Знакомство с дизассемблером

Надо ли милостивого Бога все время просить о пощаде?

Велимир

О'кей, пароль мы узнали. Но как же утомительно вводить его каждый раз с клавиатуры перед запуском программы! Хорошо бы ее хакнуть так, чтобы никакой пароль вообще не запрашивался или любой введенный пароль программа воспринимала бы как правильный.

Хакнуть, говорите?! Что ж, это несложно! Куда проблематичнее определить, *чем именно* ее хакать. Инструментарий хакеров чрезвычайно разнообразен, чего тут только нет: и дизассемблеры, и отладчики, и API-, и message-шпионы, и мониторы обращений к файлам (портам, реестру), и распаковщики исполняемых файлов, и... Сложновато начинающему кодокопателю со всем этим хозяйством разобраться!

Впрочем, шпионы, мониторы, распаковщики — второстепенные утилиты заднего плана, а основное оружие взломщика — отладчик и дизассемблер. Рассмотрим их поближе.

Как и следует из его названия, диз-ассемблер предназначен для диз-ассемблирования или «раз-ассемблирования». Если перейти с латыни на русский ДИС... ДИЗ... [лат. dis, гр. dys] — приставка, обозначающая разделение, отделение, отрицание; соответствует русским «раз...», «не...», сообщает понятию, к которому прилагается, отрицательный или противоположный смысл, например, *дисассоциация*, *дисгармония* — *Словарь иностранных слов*. То есть если ассемблирование — перевод ассемблерных команд в машинный код, то дизассемблирование — напротив, перевод машинного кода в ассемблерные команды.

Но пусть название не вводит вас в заблуждение: дизассемблер пригоден для изучения не только тех программ, что были написаны на ассемблере, круг его применения очень широк, хотя и не безграничен. Спрашиваете: где же пролегает эта граница? Отвечаю. Грубо говоря, все реализации языков программирования делятся на **компиляторы** и **интерпретаторы**.

Интерпретаторы выполняют программу в том виде, в каком она была составлена программистом. Другими словами говоря, интерпретаторы «пережевыва-

вают» исходный текст, при этом код программы доступен для непосредственного изучения безо всяких дополнительных средств. Примером могут служить приложения, написанные на Бацике или Перле. Как известно, для их запуска, помимо исходного текста программы, требуется иметь еще и сам интерпретатор, что неудобно ни пользователям (для исполнения программы в 10 килобайтов приходится устанавливать интерпретатор в 10 мегабайтов), ни разработчикам (в здравом уме и трезвой памяти раздавать всем исходные тексты своей программы!), к тому же синтаксический разбор отнимает много времени и ни один интерпретатор не может похвастаться производительностью.

Компиляторы ведут себя иначе — при первом запуске они «перемалывают» программу в машинный код, исполняемый непосредственно самим процессором без обращений к исходным текстам или самому компилятору. С человеческой точки зрения, откомпилированная программа представляет бессмысленную мешанину шестнадцатеричных байтов, разобраться в которой неспециалисту абсолютно невозможно. Это облегчает разработку защитных механизмов — не зная алгоритма, вслепую защиту не сломаешь, ну разве что она будет совсем простой.

Можно ли из машинного кода получить исходный текст программы? Нет! Компиляция — процесс однонаправленный. И дело тут не только в том, что безвозвратно удаляются метки и комментарии (ррразберемся и без комментариев — хакеры мы или нет?!), основной камень преткновения — *неоднозначность соответствия машинных инструкций конструкциям языков высокого уровня*. Более того, ассемблирование также являет собой однонаправленный процесс и автоматическое дизассемблирование принципиально невозможно. Впрочем, не будем сейчас забивать голову начинающих кодокопателей такими тонкостями и оставим эту проблему на потом.

Ряд систем разработки занимает промежуточное положение между компиляторами и интерпретаторами — исходная программа преобразуется не в машинный код, а в некоторый другой интерпретируемый язык, для исполнения которого к откомпилированному файлу дописывается собственный интерпретатор. Именно по такой схеме функционируют FoxPro, Clipper, многочисленные диалекты Бацика и некоторые другие языки.

Да, код программы по-прежнему исполняется в режиме интерпретации, но теперь из него удалена вся избыточная информация — метки, имена переменных, комментарии, а осмысленные названия операторов заменены их цифровыми кодами. Этот выстрел укладывает сразу двух зайцев: а) язык, на который переведена программа, заранее «заточен» под быструю интерпретацию и оптимизирован по размеру; б) код программы теперь недоступен для непосредственного изучения (и/или модификации).

Дизассемблирование таких программ невозможно — дизассемблер нацелен именно на машинный код, а неизвестный ему интерпретируемый язык (также называемый π -кодом) он «не переваривает». Разумеется, π -код не переваривает и процессор! Его исполняет интерпретатор, дописанный к программе. Вот интерпретатор-то дизассемблер и берет! Изучая алгоритм его работы, можно понять

«устройство» π -кода и выяснить назначение всех его команд. Это очень трудоемкий процесс! Интерпретаторы порой так сложны и занимают так много мегабайтов, что их анализ растягивается на многие месяцы, а то и годы. К счастью, нет нужды анализировать *каждую* программу — ведь интерпретаторы одной версии идентичны, а сам π -код обычно мало меняется от версии к версии, во всяком случае, его ядро не переписывается каждый день. Поэтому вполне возможно создать программу, занимающуюся переводом π -кода обратно в исходный язык. Конечно, символьные имена восстановить не удастся, но в остальном листинг будет выглядеть вполне читабельно.

Итак, дизассемблер применим для исследования откомпилированных программ и частично пригоден для анализа псевдокомпилированного кода. Раз так, он должен подойти для вскрытия парольной защиты simple.exe. Весь вопрос в том, какой дизассемблер выбрать.

Не все дизассемблеры одинаковы. Есть среди них и «интеллектуалы», автоматически распознающие многие конструкции, как то: прологи и эпилоги функций, локальные переменные, перекрестные ссылки и т. д., — а есть и «простаки», чьи способности ограничены одним лишь переводом машинных команд в ассемблерные инструкции.

Логичнее всего воспользоваться услугами дизассемблера-интеллектуала (если он есть), но... давайте не будем спешить, а попробуем выполнить весь анализ вручную. Техника, понятное дело, штука хорошая, да вот не всегда она оказывается под рукой и неплохо бы заранее научиться работе в полевых условиях. К тому же общение с плохим дизассемблером как нельзя лучше подчеркивает «вкусоности» хорошего.

Воспользуемся уже знакомой нам утилитой DUMPBIN, настоящим «швейцарским ножиком» со множеством полезных функций, среди которых притаился и дизассемблер. Дизассемблируем секцию кода (как мы помним, носящую имя .text), перенаправив вывод в файл, так как на экран он, очевидно, не поместится:

```
> dumpbin /SECTION:.text /DISASM simple.exe >.code
```

Итак, менее чем через секунду образовался файл .code размером... размером в целых триста с четвертью килобайтов. Да, исходная программа была на *два порядка* короче! Это же сколько времени потребуется, чтобы со всей этой шаманской грамотой разобраться?! Самое обидное — подавляющая масса кода никакого отношения к защитному механизму не имеет и представляет собой функции стандартных библиотек компилятора, анализировать которые нам ни к чему. Но как же их отличить от «полезного» кода?

Давайте подумаем. Мы не знаем, где именно расположена процедура сравнения паролей, и нам неизвестно ее устройство, но можно с уверенностью утверждать, что один из ее аргументов — указатель на эталонный пароль. Остается только выяснить, по какому адресу расположен этот пароль в памяти, — он-то и будет искомым значением указателя.

Заглянем еще раз в секцию данных (или в другую — в зависимости от того, где хранится пароль):

```
> dumpbin /SECTION:.data /RAWDATA simple.exe >.data
```

```
RAW DATA #3
```

```
00406000: 00 00 00 00 00 00 00 00 00 00 00 00 7B 11 40 00 .....{.@.
00406010: E4 40 40 00 00 00 00 00 00 00 00 00 20 12 40 00 ф@@.....@.
00406020: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
00406030: 45 6E 74 65 72 20 70 61 73 73 77 6F 72 64 3A 00 Enter password:.
00406040: 6D 79 47 4F 4F 44 70 61 73 73 77 6F 72 64 0A 00 myG00Dpassword..
00406050: 57 72 6F 6E 67 20 70 61 73 73 77 6F 72 64 0A 00 Wrong password..
00406060: 50 61 73 73 77 6F 72 64 20 4F 4B 0A 00 00 00 00 Password OK.....
```

Ага, пароль расположен по смещению 0x406040 (левая колонка чисел), стало быть, и указатель на него равен 0x406040. Попробуем найти это число в дизассемблированном листинге тривиальным контекстным поиском в любом текстовом редакторе.

Нашли? Вот оно (в тексте выделено жирным шрифтом):

```
00401045: 68 40 60 40 00      push      406040h
0040104A: 8D 55 98            lea       edx,[ebp-68h]
0040104D: 52                  push     edx
0040104E: E8 4D 00 00 00      call     004010A0
00401053: 83 C4 08            add      esp,8
00401056: 85 C0               test     eax,eax
00401058: 74 0F               je       00401069
```

Это один из двух аргументов функции 0x04010A0, заносимых в стек машинной командой push. Второй аргумент — указатель на локальный буфер, вероятно содержащий введенный пользователем пароль.

Тут нам придется немного отклониться от темы разговора и подробно рассмотреть передачу параметров. Наиболее распространенны следующие способы передачи аргументов функции — *через регистры* и *через стек*.

Передача параметров через регистры наиболее быстра, но не лишена недостатков, — во-первых, количество регистров весьма ограничено, а во-вторых, это затрудняет реализацию рекурсии — вызова функции из самой себя. Прежде чем заносить в регистры новые аргументы, необходимо предварительно сохранить старые в оперативной памяти. А раз так, не проще ли сразу передать аргументы через оперативную память, не мучаясь с регистрами?

подавляющее большинство компиляторов передает аргументы через стек. Единого мнения по вопросам передачи у разработчиков компиляторов нет, поэтому встречается по крайней мере два различных механизма, именуемые соглашениями *Си* и *Паскаль*.

Си-соглашение предписывает заталкивать в стек аргументы справа налево, т. е. самый первый аргумент функции заносится в стек последним и оказывается на его верхушке. Удаление аргументов из стека возложено не на саму функцию, а на вызываемый ею код. Это довольно расточительное решение, так как каждый вызов функции утяжеляет программу на несколько байтов кода, но зато это позволяет создавать функции с переменным числом аргументов — ведь удаляет-то их из стека не сама функция, а вызывающий ее код, который наверняка знает точное количество переданных аргументов.

Очистка стека обычно выполняется командой `ADD ESP, xxx`, где `xxx` количество удаляемых байтов. Поскольку, в 32-разрядном режиме каждый аргумент, **как правило**, занимает четыре байта, количество аргументов функции вычисляется так: $n_args = \frac{xxx}{4}$. Оптимизирующие компиляторы могут использовать более хитрый код — для очистки стека от нескольких аргументов они частенько их «выталкивают» в неиспользуемые регистры командой `POP` или вовсе очищают стек не сразу же после выхода из функции, а совсем в другом месте — где это удобнее компилятору.

Паскаль-соглашение предписывает заносить аргументы в стек слева направо, т. е. самый первый аргумент функции заносится в стек в первую очередь и оказывается в его «низу». Удаление аргументов из функции возложено на саму функцию и обычно осуществляется командой `RET xxx`, т. е. возврат из подпрограммы со снятием `xxx` байтов со стека.

Возвращаемое функцией значение в обоих соглашениях передается через регистр `EAX` (или `EDX:EAX` при возвращении 64-разрядных переменных).

Поскольку исследуемая нами программа написана на языке `C` и, стало быть, заносит аргументы справа налево, ее исходный текст выглядит приблизительно так:

```
*0x4010A0) (ebp-68, "myG00Dpassword")
```

В том, что аргументов именно два, а не, скажем, четыре или десять, нас убеждает команда `ADD ESP, 8`, расположенная вслед за `CALL`.

```
0040104E: E8 4D 00 00 00 call 004010A0
```

```
00401053: 83 C4 08 add esp, 8
```

Остается выяснить назначение функции `0x4010A0`, хотя, если поднапрячь свою интуицию, то этого можно и не делать! И так ясно — эта функция сравнивает пароль, иначе зачем бы ей его передавали? *Как* она это делает — вопрос десятый, а вот что нас действительно интересует, так возвращенное ею значение. Итак, опускаемся на одну строчку ниже:

```
0040104E: E8 4D 00 00 00 call 004010A0
00401053: 83 C4 08 add esp, 8
00401056: 85 C0 test eax, eax
00401058: 74 0F je 00401069
```

Что мы видим? Команда `TEST EAX, EAX` проверяет возвращенное функцией значение на равенство нулю, и если оно действительно равно нулю, следующая за ней команда `JE` совершает прыжок на `0x401096` строку.

В противном же случае (т. е. если `EAX != 0`)...

```
0040105A: 68 50 60 40 00 push 406050h
```

Похоже еще на один указатель, не правда ли? Проверим это предположение, заглянув в сегмент данных:

```
00406050: 57 72 6F 6E 67 20 70 61 73 73 77 6F 72 64 0A 00 Wrong password..
```

Уже теплее! Указатель вывел нас на строку *Wrong password*, очевидно, выводимую следующей функцией на экран. Значит, ненулевое значение EAX свидетельствует о ложном пароле, а ноль — об истинном.

О'кей, тогда переходим к анализу валидной ветви программы...

```
0040105F: E8 D0 01 00 00      call     00401234
00401064: 83 C4 04             add      esp, 4
00401067: EB 02                jmp      0040106B
00401069: EB 16                jmp      00401081
...
00401081: 68 60 60 40 00      push     406060h
00401086: E8 A9 01 00 00      call     00401234
```

Так, еще один указатель. С функцией 0x401234 мы уже встречались выше — она (предположительно) служит для вывода строк на экран. Ну а сами строки можно отыскать в сегменте данных. На этот раз там притаилось *Password OK*.

Оперативные соображения следующие: если заменить команду JE на JNE, то программа отвергнет истинный пароль как неправильный, а любой неправильный пароль воспримет как истинный. А если заменить TEST EAX,EAX на XOR EAX,EAX, то после исполнения этой команды регистр EAX будет *всегда* равен нулю, какой бы пароль ни вводился.

Дело за малым — найти эти самые байтики в исполняемом файле и малость подправить их.

Шаг третий. Хирургический

Не торопитесь на встречу с Богом, еще встретитесь.

Народная мудрость

Внесение изменений непосредственно в исполняемый файл — дело серьезное. Стиснутым уже существующим кодом, нам приходится довольствоваться только тем, что есть, и ни раздвинуть команды, ни даже сдвинуть их, выкинув из защиты «лишние запчасти», не получится. Ведь это привело бы к сдвигу смещений всех остальных команд, тогда как значения указателей и адресов переходов остались бы без изменений и стали бы указывать совсем не туда, куда нужно!

Ну, с «выкидыванием запчастей» справиться как раз так просто — достаточно забить код командами NOP (опкод который 0x90, а вовсе не 0x0, как почему-то думают многие начинающие кодокопатели), т. е. пустой операцией (вообще-то NOP — это просто другая форма записи инструкции XCHG EAX, EAX — если интересно). С «раздвижкой» куда сложнее! К счастью, в PE-файлах всегда присутствует множество «дыр», оставшихся от выравнивания, в них-то и можно разместить свой код или свои данные.

Но не проще ли просто откомпилировать ассемблированный файл, предварительно внося в него требуемые изменения? Нет, не проще, и вот почему: если ассемблер не распознает указатели, передаваемые функции (а как мы видели, наш

дизассемблер не смог отличить их от констант), он соответственно не позаботится должным образом их скорректировать, и, естественно, программа работать не будет.

Приходится резать программу живую. Легче всего это делать с помощью утилиты HIEW, «переваривающей» PE-формат файлов и упрощающей тем самым поиск нужного фрагмента. Запустим его, указав имя файла в командной строке `hiew simple.exe`, двойным нажатием клавиши **<Enter>**, переключимся в режим ассемблера и при помощи клавиши **<F5>** перейдем к требуемому адресу. Как мы помним, команда TEST, проверяющая результат, возвращенный функцией, на равенство нулю, располагалась по адресу 0x401056.

```
0040104E: E8 4D 00 00 00      call     004010A0
00401053: 83 C4 08            add      esp,8
00401056: 85 C0               test     eax,eax
00401058: 74 0F              je       00401069
```

Чтобы HIEW мог отличить адрес от смещения в самом файле, предварим его символом точки: `.401056`:

```
00401056: 85C0               test     eax,eax
00401058: 740F              je       .000401069 ----- (1)
```

Ага, как раз то, что нам надо! Нажмем клавишу **<F3>** для перевода HIEW в режим правки, подведем курсор к команде TEST EAX, EAX и, нажав клавишу **<Enter>**, заменим ее на XOR EAX, EAX.

```
00001056: 33C0               xor      eax,eax
00001058: 740F              je       000001069
```

С удовлетворением заметив, что новая команда в аккурат вписалась в предыдущую, нажмем клавишу **<F9>** для сохранения изменений на диске, а затем выйдем из HIEW и попробуем запустить программу, вводя первый пришедший на ум пароль:

```
>simple.exe
Enter password:Привет, шляпа!
Password OK
```

Получилось! Защита пала! Хорошо, а как бы мы действовали, не умея HIEW «переваривать» PE-файлы? Тогда пришлось бы прибегнуть к контекстному поиску. Обратим свой взор на шестнадцатеричный дамп, расположенный дизассемблером слева от ассемблерных команд. Конечно, если пытаться найти последовательность 85 C0 — код команды TEST EAX, EAX, ничего хорошего из этого не выйдет, — этих самых TEST'ов в программе может быть несколько сотен, а то и больше. Комбинация ADD ESP,8\TEST EAX, EAX также вряд ли будет уникальна, поскольку встречается во многих типовых конструкциях языка C: `if (func(arg1,arg2))...`, `if (!func(arg1,arg2))...`, `while(func(arg1,arg2))` и т. д. А вот адрес перехода, скорее всего, во всех ветках программы различен и подстрока ADD ESP,8/TEST EAX,EAX/JE 00401069 имеет хорошие шансы на уникальность. Попробуем найти в файле соответствующий ей код: 83 C4 08 85 C0 74 0F (в HIEW для этого достаточно нажать клавишу **<F7>**).

Опп-с! Найдено только одно вхождение, что нам, собственно, и нужно. Давайте теперь попробуем модифицировать файл непосредственно в hex-режиме, не переходя в ассемблер. Попутно возьмем себе на заметку — инверсия младшего бита кода команды приводит к изменению условия перехода на противоположное, т. е. 74 JE → 75 JNE.

Работает? (В смысле защита свихнулась окончательно — не признает истинные пароли, зато радостно приветствует остальные.) Замечательно! Остается решить, как эту взломанную программу распространять. То есть распространить-то ее дело нехитрое — на то и существуют CDR-писцы, BBS, сеть Интернет, наконец! Заливай, пиши, нарежай — не хочу. Не хотите — и правильно! Незаконное это дело — распространять программное обеспечение в обход его владельца. Эдак и засадить могут (причем прецеденты уже имеются). Куда безопаснее возложить распространение программы на ее дистрибьюторов, но до каждого пользователя донести, как эту программу взломать. Ковыряться в законным образом приобретенном приложении потребитель вправе, а распространение информации о взломе не запрещено в силу закона о свободе информации. Правда, при ближайшем рассмотрении выясняется, что этот закон и у нас, и за океаном действует лишь формально, и если не посадить, то по крайней мере попытаться это сделать, правоохранные органы вполне могут (и не только могут, но и делают). Когда дело касается чьих-то финансовых интересов, правосудие автоматически становится на их защиту. Наивно думать, что соблюдение закона само по себе дает некие гарантии. Нет и еще раз нет! Чувствовать себя в относительной безопасности можно лишь при условии соблюдения кодекса *«да не навреди сильным мира сего»*.

В любом случае информация о взломе — это не совсем то же, что сам взлом, и за это труднее привлечь к ответственности. Единственная проблема — попробуй-ка объясни этим пользователям, как пользоваться hex-редактором и искать в нем такие-то байтики. Запорют же ведь файл за милую душу! Вот для этой цели и существуют автоматические взломщики.

Для начала нужно установить, какие именно байты были изменены. Для этого нам вновь потребуется оригинальная копия модифицированного файла, предосторожно сохраненная перед его правкой, и какой-нибудь «сравниватель» файлов. Наиболее популярными на сегодняшний день являются **c2u by Professor Nimnul** и **MakeCrk by Doctor Stein's labs**. Первый гораздо предпочтительнее, так как он не только более точно придерживается наиболее популярного «стандарта», но и умеет генерировать расширенный хск-формат. На худой конец можно воспользоваться и штатной утилитой, входящей в поставку MS-DOS\Windows — fc.exe (сокращение от **FileCompare**).

Запустим свой любимый компаратор (это уж какой кому больше по душе) и посмотрим на результат его работы:

```
> fc simple.exe simple.ex_ > simple.dif
```

файл различий

хакнутый файл

оригинальный файл


```
> type simple.dif
Сравнение файлов simple.exe и SIMPLE.EX_
00001058: 74 75
```

Первая слева колонка указывает смещение байта от начала файла, вторая — содержимое байта оригинального файла, а третья — его значение после модификации. Теперь сравним это с отчетом утилиты c2u:

```
>c2u simple.exe simple.ex_
```

Все исправления заносятся в файл *.crx, где «*» — имя оригинального файла. Рассмотрим результат сравнения поближе:

```
>type simple.crx
[BeginXCK]
■ Description   : $) 1996 by Professor Nimnul
■ Crack subject :
■ Used packer   : None/UnKn0wN/WWPACK/PKLITE/AINEXE/DIET/EXEPACK/PRO-PACK/LZEXE
■ Used unpacker : None/UNP/X-TRACT/iNTRUDER/AUTOHack/CUP/TRON
■ Comments      :
■ Target OS     : DOS/WiN/WNT/W95/OSx/UNIX
■ Protection    : [REDACTED] %17
■ Type of hack  : Bit hack/JMP Correction
■ Language      : UnKn0wN/Turbo/Borland/Quick/MS/Visual C/C++/Pascal/Assembler
■ Size          : 28672
■ Price         : $000
■ Used tools    : TD386 v3.2, HiEW 5.13, C2U/486 v0.10
■ Time for hack : 00:00:00
■ Crack made at : 21-07-2001 12:34:21
■ Under Music   : iRON MAIDEN
[BeginCRA]
Difference(s) between simple.exe & simple.ex_
SIMPLE.EXE
00001058: 74 75
[EndCRA]
[EndXCK]
```

Собственно, сам результат сравнений ничуть не изменился, разве что к файлу добавился текстовый заголовок, поясняющий, что это за северный олень такой. Все поля не стандартизированы, и их набор сильно разнится от одного взломщика к другому — при желании вы можете снабдить заголовок своими собственными полями или же, напротив, выкинуть из него чужие. Однако не стоит злоупотреблять этим без серьезной необходимости, уж лучше придерживаться какого-то одного шаблона.

Итак. *Description* — пояснение к взлому, заполняемое в меру буйства фантазии и уровня распушенности. В нашем случае оно может выглядеть, например, так: «Тестовый взлом № 1».

Crack subject — предмет крака, — т. е. то, что, собственно говоря, мы только что сломали. Пишем: «Парольная защита simple.exe».

Used packer — используемый упаковщик. Еще во времена старушки MS-DOS существовали и были широко распространены упаковщики исполняемых файлов, автоматически разжимающие файл в памяти при его запуске. Этим достигалась экономия дискового пространства (помните, какими смехотворными по нынешним временам были размеры винчестеров конца восьмидесятых — начала девяно-

стых!) и параллельно с этим усиливалась защита — ведь упакованный файл недоступен для непосредственного изучения, а тем более — правки. Прежде чем начать что-то делать, файл необходимо распаковать, причем это делать приходится и самому ломателю, и всем пользователям этого `src`-файла. Поскольку наш файл не был упакован, оставим это поле пустым или запишем в него *None*.

Used unpacker — рекомендуемый распаковщик (если он необходим). Дело в том, что не все распаковщики одинаковы, многие упаковщики весьма искушены в технике защиты и умело сопротивляются попыткам их «снять». Понятное дело, распаковщики тоже не лыком шиты и держат своих тузов в рукавах, но... автоматическая распаковка — штука капризная. Бывает, «интеллектуальный» `unpacker` легко расправляется со всеми «крутыми» `packer`'ами, но тихо сдыхает на простых защитах, и соответственно случается и наоборот. Дабы не мучить пользователей утомительным перебором всех имеющихся у них распаковщиков (пользователь — он ведь тоже человек!), правила хорошего тона обязывают указывать по крайней мере один заведомо подходящий `unpacker`, а лучше — два или три сразу (вдруг какого-то из них у пользователя и не будет). Если же распаковщик не требуется — оставляйте это поле пустым или *None*.

Comments — комментарии. Вообще-то это поле задумано для перечисления дополнительных действий, которые пользователь должен выполнить перед взломом, ну, например, снять с файла атрибут «системный» или, напротив, установить его. Но поскольку какие-либо дополнительные действия требуются только в экзотических случаях, в это поле обычно помещают разнообразные лозунги и комментарии (да, правильно, бывает, что и нецензурную брань по поводу умственных способностей разработчика защиты).

Target OS — операционная система, для которой предназначен и (внимание!) в которой хакер тестировал сломанный продукт. Вовсе не факт, что программа сохранит после взлома черты своей прежней совместимости. Так, например, поле контрольной суммы `Win 9x` всегда игнорирует, а `Win NT` — нет и, если его не скорректировать, файл запускаться не будет! В нашем случае контрольная сумма заголовка `PE`-файла равна нулю (так ведет себя компилятор), что означает — целостность файла не проверяется и он после хака будет успешно работать как под `Win 9x`, так и под `Win NT`.

Protection — степень «крутизны» защиты, выражаемой в процентах. 100%, по идее, соответствуют пределу интеллектуальных возможностей хакера. Но кто же в этом захочет признаваться? Неудивительно, что «крутизну» защиты обычно занижают, порой даже больше, чем на порядок (смотрите все, вот я какой крутой хакер, для меня что угодно взломать не сложнее, чем кончик хвоста обмочить!). Нечестность — не порок, но...

Type of hack — тип хака — поле, полезное скорее для других хакеров, чем для пользователей, ничего не смыслящих в защитах и типах их взлома. Впрочем, с типами взломов не все гладко и у самих хакеров — общепризнанных классификаций нет. Наиболее употребляемый термин **bit-hack**, как и следует из его назва-

ния, обозначает взлом посредством изменения одного или нескольких битов в одном или нескольких байтах. Частный случай bit-hack'a — JMP correction (jumping) — модификация адреса или условия перехода (то, что мы только что и сделали). NOPing — это bit-hack с заменой прежних инструкций на команду NOP или вставку незначащих команд, например, для затирания двухбайтового JZ xxx можно применить сочетание однобайтовых INC EAX/DEC EAX.

Language — язык, а точнее, компилятор, на котором написана программа. В нашем случае — Microsoft Visual C++ (мы это знаем, поскольку только что ее компилировали), а вот как быть с чужими программами? Первое, что приходит на ум, — поискать в файле копирайты — их оставляют очень многие компиляторы, в том числе и Visual C++, — к примеру: 000053d9:Microsoft Visual C++ Runtime Library. Если же компиляторов нет, то пробуем прогнать файл через IDA — она автоматически распознает большинство стандартных библиотек даже с указанием конкретной версии, в крайнем случае пробует определить язык по самому коду, вспоминая о соглашениях Си и Паскаль и пытаясь найти знакомые черты известных вам компиляторов (у каждого компилятора свой «почерк», и опытный хакер способен узнать не только, чем компилировалась программа, но даже и определить ключи оптимизации).

Size — размер ломаемой программы, служащий для контроля версии (чаще всего, хотя и не всегда, каждая версия программы имеет свой размер). Размер автоматически определяется утилитой c2u, и самостоятельно вставлять его нет никакой нужды.

Price — стоимость лицензионной копии программы (должен же пользоваться знать, сколько денег ему сэконобил этот крак!).

Used tools — используемые инструменты. Незаполнение этого поля считается дурным тоном — действительно же интересно, чем именно была хакнута программа! Особенно этим интересуются пользователи, наивно полагающие, что, если они раздобудут тот же DUMPBIN и NIEW, защита сама собой сломается.

Time for hack — время, затраченное на хак, включая перерывы на «перекурить» и «сходить водички попить». Интересно, какой процент людей честно заполняет это поле, не пытаясь показаться «круче» в чужих глазах? Так что особенно доверять ему не следует...

Crack made at — дата завершения крака. Подставляется автоматически, и править ее нет необходимости (разве что вы «жаворонок» и хотите выдать себя за «сову», проставляя время окончания взлома 3 часами ночи 31 декабря).

Under Music — музыка, прослушиваемая во время хака (еще не хватает поля «Имя любимого хомячка»). Вы слушали музыку во время хака? Если да, то пишите — пусть все знают ваши вкусы (заодно не забудьте цвет майки и температуру воздуха за бортом выше нуля).

В результате всех мучений у нас должно получиться приблизительно следующее:

Шаг четвертый. Знакомство с отладчиком

Оставь свои мозги за дверью и внеси сюда только тело.

Ф. Тейлор

Помимо дизассемблирования существует и другой способ исследования программ — отладка. Изначально под отладкой понималось пошаговое исполнение кода, также называемое *трассировкой*. Сегодня же программы распухли настолько, что трассировать их бессмысленно — вы моментально утонете в омуте вложенных процедур, так и не поняв, что они, собственно, делают. Отладчик не лучшее средство изучения алгоритма программы — с этим эффективнее справляется интерактивный дизассемблер (например, IDA).

Подробный разговор об устройстве отладчика мы отложим на потом (см. главу «*Приемы против отладчиков*»), а здесь ограничимся лишь перечнем основных функциональных возможностей типовых отладчиков (без этого невозможно их осмысленное применение):

- отслеживание обращений на запись/чтение/исполнение к заданной ячейке (региону) памяти, далее по тексту именуемое бряком (брейком);
- отслеживание обращений на запись/чтение к портам ввода-вывода (уже неактуально для современных операционных систем, запрещающих пользовательским приложениям проделывать такие трюки, — это теперь прерогатива драйверов, а на уровне драйверов реализованы очень немногие защиты);
- отслеживание загрузки DLL и вызова из них таких-то функций, включая системные компоненты (как мы увидим далее, это основное оружие современного взломщика);
- отслеживание вызова программных/аппаратных прерываний (большой частью уже не актуально — не так много защит балуется с прерываниями);
- отслеживание сообщений, посылаемых приложением окну;
- и, разумеется, контекстный поиск в памяти.

Как именно это делает отладчик, пока знать необязательно, достаточно знать, что он это умеет, и все. Куда актуальнее вопрос: какой отладчик умеет это делать? Широко известный в пользовательских кругах Turbo Debugger на самом деле очень примитивный и никчемный отладчик — очень мало хакеров им что-то ломает.

Самое мощное и универсальное средство — Soft-Ice, сейчас доступный для всех Windows-платформ (а когда-то он поддерживал лишь одну Windows 95, но не Windows NT). Последняя на момент написания книги, четвертая версия, не очень-то стабильно работает с видеоадаптером автора, поэтому приходится ограничиваться более ранней, но зато устойчивой версией 3.25.

Способ 0. Бряк на оригинальный пароль

Используя поставляемую вместе с Айсом утилиту `wldr`, загрузим ломаемый нами файл, указав его имя в командной строке, например, так:

```
>wldr simple.exe
```

Да, автор знает, что `wldr` — 16-разрядный загрузчик, и NuMega рекомендует использовать его 32-разрядную версию `loader32`, специально разработанную для Win 9x\NT. Это так, но `loader32` частенько сбоит (в частности, не всегда останавливается на первой строчке запускаемой программы), а `wldr` успешно работает и с 32-разрядными приложениями; единственный присущий ему недостаток — отсутствие поддержки длинных имен файлов.

Если отладчик настроен корректно, на экране появится черное текстовое окно, обычно вызывающее большое удивление у начинающих, — это в нашу-то эпоху визуальщины серый текст и командный язык а la `command.com`! А почему бы и нет? Набрать на клавиатуре нужную команду куда быстрее, чем отыскать ее в длинной веренице вложенных меню, мучительно вспоминая, где же вы ее в последний раз видели. К тому же язык — это естественное средство выражения мыслей, а меню — оно годится разве что для выбора блюд в ресторане. Вот хороший пример — попробуйте с помощью проводника Windows вывести на печать список файлов такой-то директории. Не получается? А в MS-DOS это было так просто: `dir >PRN`, и никаких лаптей!

Если в окне кода видны одни лишь инструкции `INVALID` (а оно так и будет), не пугайтесь — просто Windows еще не успела спроецировать исполняемый файл в память и выделить ему страницы. Стоит нажать клавишу **<F10>** (аналог команды `P` — трассировка без заходов в функцию) или клавишу **<F8>** (аналог команды `T` — трассировка с заходами в функции), как все сразу же станет на свои места.

```
001B:00401277  INVALID
001B:00401279  INVALID
001B:0040127B  INVALID
001B:0040127D  INVALID
:P

001B:00401285  PUSH    EBX
001B:00401286  PUSH    ESI
001B:00401287  PUSH    EDI
001B:00401288  MOV     [EBP-18],ESP
001B:0040128B  CALL    [KERNEL32!GetVersion]
001B:00401291  XOR     EDX,EDX
001B:00401293  MOV     DL,AH
001B:00401295  MOV     [0040692C],EDX
```

Обратите внимание: в отличие от дизассемблера `DUMPBIN`, Айс распознает имена системных функций, чем существенно упрощает анализ. Впрочем, анализировать всю программу целиком нет никакой нужды. Давайте попробуем наско-ро найти защитный механизм и, не вникая в подробности его функционирования, напрочь отрубить защиту. Легко сказать, но сделать еще проще! Вспомним, по ка-

кому адресу расположен в памяти оригинальный пароль. Э... что-то плохо у нас с этим получается — то ли память битая, то ли медведь на лапоть наступил, но точный адрес никак не хочет вспоминаться. Не хочет — не надо. Найдем-ка мы его самостоятельно!

В этом нам поможет команда `map32`, выдающая карту памяти выбранного модуля (наш модуль называется `simple` — по имени исполняемого файла за вычетом расширения).

```
:map32 simple
Owner      Obj Name  Obj#   Address          Size      Type
simple      .text     0001   001B:00401000    00003F66  CODE   RO
simple      .rdata    0002   0023:00405000    0000081E  IDATA  RO
simple      .data     0003   0023:00406000    00001E44  IDATA  RW
```

Вот он, адрес начала секции: `.data`. То, что пароль находится в секции `.data`, надеюсь, читатель все еще помнит. Даем команду `d 23:406000` (возможно, предварительно придется создать окно командой `ws` — если окна данных нет) и, нажав сочетание клавиш **<ALT-D>** для перехода в это окно, прокрутим его содержимое клавишей **<стрелка вниз>** или кирпичом на клавише **<Page Down>**. Впрочем, кирпич излишен, долго искать не придется:

```
0023:00406040 6D 79 47 4F 4F 44 70 61-73 73 77 6F 72 64 0A 00 myG00Dpassword..
0023:00406050 57 72 6F 6E 67 20 70 61-73 73 77 6F 72 64 0A 00 Wrong password..
0023:00406060 50 61 73 73 77 6F 72 64-20 4F 4B 0A 00 00 00 00 Password OK.....
0023:00406070 47 6E 40 00 00 00 00 00-40 6E 40 00 01 01 00 00 Gn@.....@n@.....
0023:00406080 00 00 00 00 00 00 00 00-00 10 00 00 00 00 00 00 .....
0023:00406090 00 00 00 00 00 00 00 00-00 00 00 00 02 00 00 00 .....
0023:004060A0 01 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 .....
0023:004060B0 00 00 00 00 00 00 00 00-00 00 00 00 02 00 00 00 .....
```

Есть контакт! Задумаемся еще раз (второй раз за этот день), чтобы проверить корректность введенного пользователем пароля, защита, очевидно, должна сравнить его с оригинальным. А раз так, установив точку останова на чтении памяти по адресу `0x406040`, мы поймаем за хвост сравнивающий механизм. Сказано — сделано.

```
:bpm 406040
```

Теперь нажимаем сочетание клавиш **<CTRL-D>** для выхода из отладчика (или отдаем команду «х») и вводим любой пришедший на ум пароль, например `KPNC++`. Отладчик «всплывает» незамедлительно:

```
001B:004010B0 MOV     EAX,[EDX]
001B:004010B2 CMP     AL,[ECX]
001B:004010B4 JNZ     004010E4          (JUMP ↑)
001B:004010B6 OR      AL,AL
001B:004010B8 JZ      004010E0
001B:004010BA CMP     AH,[ECX+01]
001B:004010BD JNZ     004010E4
001B:004010BF OR      AH,AH
```

```
Break due to BPMB #0023:00406040 RW DR3 (ET=752.27 milliseconds)
```



```
MSR LastBranchFromIp=0040104E
MSR LastBranchToIp=004010A0
```

В силу архитектурных особенностей процессоров Intel бряк срабатывает после инструкции, выполнившей «поползновение», т. е. CS:EIP указывают на следующую выполняемую команду. В нашем случае — JNZ 004010E4, а к памяти, стало быть, обратилась инструкция CMP AL, [ECX]. А что находится в AL? Поднимаем взгляд еще строкой выше — MOV EAX, [EDX]. Можно предположить, что ECX содержит указатель на строку оригинального пароля (поскольку он вызвал всплытие отладчика), а EDX в таком случае — указатель на введенный пользователем пароль. Проверим наше предположение.

```
:d edx
0023:00406040 6D 79 47 4F 4F 44 70 61-73 73 77 6F 72 64 0A 00  myG00Dpassword..
:d edx
0023:0012FF18 4B 50 4E 43 2B 2B 0A 00-00 00 00 00 00 00 00 00  KPNC++.....
```

И правда — догадка оказалась верна. Теперь вопрос: а как это заломить? Вот, скажем, JNZ можно поменять на JZ или, еще оригинальнее, заменить EDX на ECX. Тогда оригинальный пароль будет сравниваться сам с собой! Погодите, погодите, не стоит так спешить! А что, если мы находимся не в теле защиты, а в библиотечной функции (действительно, мы находимся в теле strcmp), ее изменение приведет к тому, что программа будет воспринимать *любые* строки как идентичные. Любые — а не только строки пароля. Это не повредит нашему примеру, где strcmp вызывалась лишь однажды, но завалит нормальное полнофункциональное приложение. Что же делать?

Выйти из strcmp и подкорректировать тот самый IF, который анализирует правильный — неправильный пароль. Для этого служит команда **P RET** (трассировать, пока не встретится ret — инструкция возврата из функции).

```
:P RET
001B:0040104E CALL 004010A0
001B:00401053 ADD ESP, 08
001B:00401056 TEST EAX, EAX
001B:00401058 JZ 00401069
001B:0040105A PUSH 00406050
001B:0040105F CALL 00401234
001B:00401064 ADD ESP, 04
001B:00401067 JMP 0040106B
```

Знакомые места! Помните, мы их посещали дизассемблером? Алгоритм действий прежний — запоминаем адрес команды TEST для последующей замены ее на XOR или записываем последовательность байтов, идентифицирующую... эй, постойте, а где же наши байты — шестнадцатеричное представление команд? Коварный Айс по умолчанию их не выводит, и заставить его это сделать помогает команда **CODE ON**.

```
code on
001B:0040104E E84D000000 CALL 004010A0
001B:00401053 83C408 ADD ESP, 08
001B:00401056 85C0 TEST EAX, EAX
```

001B:00401058	740F	JZ	00401069
001B:0040105A	6850604000	PUSH	00406050
001B:0040105F	E8D0010000	CALL	00401234
001B:00401064	83C404	ADD	ESP, 04
001B:00401067	EB02	JMP	0040106B

Вот теперь совсем другое дело! Но можно ли быть уверенным, что эти байтики по этим самым адресам будут находиться в исполняемом файле? Вопрос не так глуп, как кажется на первый взгляд. Попробуйте сломать описанным выше методом пример crackme0x03. На первый взгляд он очень похож на simple.exe, даже оригинальный пароль располагается по тому же самому адресу. Ставим на него бряк, дожидаемся всплытия отладчика, выходим из сравнивающей процедуры и попадаем на точно такой же код, который уже встречался нам ранее:

001B:0042104E	E87D000000	CALL	004210D0
001B:00421053	83C408	ADD	ESP, 08
001B:00421056	85C0	TEST	EAX, EAX
001B:00421058	740F	JZ	00421069

Сейчас мы запустим HIEW, перейдем по адресу 0x421053 и... Эй, постой, HIEW ругается и говорит, что в файле нет такого адреса! Последний байт заканчивается на 0x407FFF. Быть может, мы находимся в теле системной функции Windows? Но нет, системные функции Windows расположены значительно выше, начиная с адреса 0x80000000.

Фокус состоит в том, что PE-файл может быть загружен по адресу, отличному от того, для которого он был создан (это свойство называется *перемещаемостью*), при этом система автоматически корректирует все ссылки на абсолютные адреса, заменяя их новыми значениями. В результате — образ файла в памяти не будет соответствовать тому, что записано на диске. Хорошенькое начало! Как же теперь найти место, которое нужно править?

Задачу несколько облегчает тот факт, что системный загрузчик умеет перемещать только DLL, а исполняемые файлы всегда пытается загрузить по «родному» для них адресу. Если же это невозможно — загрузка прерывается с выдачей сообщения об ошибке. Выходит, мы имеем дело с DLL, загруженной исследуемой нами защитой. Хм, вроде бы не должно здесь быть никаких DLL, да и откуда бы им взяться?

Что ж, изучим листинг 2 на предмет выяснения, как же он работает.

Листинг 2. Исходный текст защиты crackme0x3

```
#include <stdio.h>
#include <windows.h>

__declspec(dllexport) void Demo()
{
    #define PASSWORD_SIZE 100
    #define PASSWORD "myGOODpassword\n"

    int count=0;
```

```

char buff[PASSWORD_SIZE]="";

for(;;)
{
    printf("Enter password:");
    fgets(&buff[0],PASSWORD_SIZE-1,stdin);

    if (strcmp(&buff[0],PASSWORD))
        printf("Wrong password\n");
    else break;

    if (++count>2) return -1;
}
printf("Password OK\n");
}

main()
{
    HMODULE hmod;
    void (*zzz)();

    if ((hmod=LoadLibrary("crack0~1.exe"))
        && (zzz=(void (*)())GetProcAddress(h, "Demo")))
        zzz();
}

```

Какой, однако, извращенный способ вызова функции! Защита экспортирует ее непосредственно из самого исполняемого файла и этот же файл загружает как DLL (да, один и тот же файл может быть одновременно и исполняемым приложением, и динамической библиотекой!)

«Все равно ничего не сходится, — возразит программист средней квалификации, — всем же известно, что Windows не настолько глупа, чтобы дважды грузить один и тот же файл, — LoadLibrary всего лишь возвратит базовый адрес модуля crackme0x03, но не станет выделять для него память». А вот как бы не так! Хитрая защита обращается к файлу по его альтернативному короткому имени, вводя системный загрузчик в глубокое заблуждение!

Система выделяет память и возвращает базовый адрес загружаемого модуля в переменной hmod. Очевидно, код и данные этого модуля смещены на расстояние hmod — base, где base — базовый адрес модуля — тот, с которым работают HIEW и дизассемблер. Базовый адрес узнать нетрудно, достаточно вызвать тот же DUMPBIN с ключом /HEADERS (его ответ приведен в сокращенном виде):

```

>dumpbin /HEADERS crack0x03
OPTIONAL HEADER VALUES
...
400000 image base
...

```

Значит, базовый адрес — 0x400000 (в байтах). А определить адрес загрузки можно командой mod -u отладчика (ключ и разрешает выводить только прикладные, т. е. не системные, модули).

```
:mod -u
hMod Base PEHeader Module Name File Name
00400000 004000D8 crack0x0 \.PHCK\src\crack0x03.exe
00420000 004200D8 crack0x0 \.PHCK\src\crack0x03.exe
00000000
77E80000 77E800D0 kernel32 \WINNT\system32\kernel32.dll
77F80000 77F800C0 ntdll \WINNT\system32\ntdll.dll
```

Смотрите, загружено сразу две копии crack0x03, причем последняя расположена по адресу 0x420000, как раз то, что нам надо! Теперь нетрудно посчитать, что адрес 0x421056 (тот, что мы пытались последний раз найти в ломаемом файле) «на диске» будет соответствовать адресу 0x421056 - (0x42000 - 0x400000) == 0x421056 - 0x20000 == 0x401056. Смотрим:

```
00401056: 85C0          test     eax, eax
00401058: 740F          je      .000401069  ----- (1)
```

Все верно, посмотрите, как хорошо это совпадает с дампом отладчика:

```
001B:00421056 85C0          TEST     EAX, EAX
001B:00421058 740F          JZ      00421069
```

Разумеется, описанная методика вычислений применима к любым DLL, а не только к тем, что представляют собой исполняемый файл.

А вот если бы мы пошли не путем адресов, а попытались найти в ломаемой программе срисованную с отладчика последовательность байтов, включая и ту часть, которая входит в CALL 00422040, — интересно, нашли бы мы ее или нет?

```
001B:0042104E E87D000000    CALL     004210D0
001B:00421053 83C408        ADD     ESP, 08
001B:00421056 85C0          TEST     EAX, EAX
001B:00421058 740F          JZ      00421069
:Образ файла в памяти.
```

```
.0040104E: E87D000000    call     .0004010D0  ----- (1)
.00401053: 83C408        add      esp, 008 ; " "
.00401056: 85C0          test     eax, eax
.00401058: 740F          je      .000401069  ----- (2)
:Образ файла на диске.
```

Вот это новость — командам CALL 0x4210D0 и CALL 0x4010D0 соответствует один и тот же машинный код — E8 7D 00 00 00! Как же такое может быть?! А вот как — операнд процессорной инструкции 0xE8 представляет собой не смещение подпрограммы, а *разницу смещений подпрограммы и инструкции, следующей за командой call*. То есть в первом случае: 0x421053 (смещение инструкции, следующей за CALL) + 0x0000007D (не забываем об обратном порядке байтов в двойном слове) == 0x4210D0 — вот он, искомый адрес. Таким образом, при изменении адреса загрузки коррекция кодов команд CALL не требуется.

Оценка по аналогии основывается на предположении, что если два или более объекта согласуются друг с другом в некоторых отношениях, то они, вероятно, согласуются и в других отношениях.

Г. Селье. От мечты к открытию

Рассуждения по аналогии — опасная штука. Увлеченные стройностью аналогии, мы подчас даже не задумываемся о проверке. Между тем аналогии лгут чаще, чем этого хотелось бы.

В примере `scask0x03`, среди прочего кода, есть и такая строка (найдите ее с помощью `hiew`):

```
004012C5: 89154C694000      mov     [00040694C],edx
```

Легко видеть, что команда `MOV` обращается к ячейке не по относительному, а по абсолютному адресу. Вопрос: а) выясните, что произойдет при изменении адреса загрузки модуля; б) как вы думаете, будет ли теперь совпадать образ файла на диске и в памяти?

Заглянув отладчиком по адресу `0x4212C5` (`0x4012C5 + 0x2000`), мы увидим, что обращение идет совсем не к ячейке `0x42694C`, а к ячейке `0x40694C`! Наш модуль самым бессовестным образом вторгается в чужие владения, модифицируя их по своему усмотрению. Так и до краха системы докатиться недолго! В данном случае это не происходит только потому, что искомая строка расположена в `Startup`-процедуре (стартовом коде) и выполняется лишь однажды — при запуске приложения, а из загруженного модуля не вызывается.

Другое дело, если бы функция `Demo()` обращалась к какой-нибудь статической переменной — компилятор, подставив ее непосредственное смещение, сделал бы модуль перемещаемым! После сказанного становится непонятно: как же тогда ухитряются работать динамически подключаемые библиотеки (DLL), адрес загрузки которых заранее неизвестен? Поразмыслив некоторое время, мы найдем, по крайней мере, два решения проблемы:

Первое — вместо непосредственной адресации использовать относительную, например `[reg+offset_val]`, где `reg` — регистр, содержащий базовый адрес загрузки, а `offset_val` — смещение ячейки от начала модуля. Это позволит модулю грузиться по любому адресу, но заметно снизит производительность программы уже хотя бы за счет потери одного регистра...

Второе — научить загрузчик корректировать непосредственные смещения в соответствии с выбранным базовым адресом загрузки. Это, конечно, несколько замедлит загрузку, но зато не ухудшит быстродействие самой программы. Не факт, что временем загрузки можно свободно пренебречь, но специалисты из `Microsoft` выбрали именно этот способ.

Единственная проблема — как отличить действительные непосредственные смещения от констант, совпадающих с ними по значению? Не дизассемблировать же в самом деле DLL, чтобы разобраться, какие именно ячейки в ней необходимо «подкрутить»? Верно, куда проще перечислить их адреса в специальной таблице, расположенной непосредственно в загружаемом файле и носящей гордое имя **«Таблицы перемещаемых элементов»** или (*Relocation [Fix Up] table* — по-английски). За ее формирование отвечает линкер (он же компоновщик), и такая таблица присутствует в каждой DLL.

Чтобы познакомиться с ней поближе, откомпилируем и изучим следующий пример:

Листинг 3. Исходный текст fixupdemo.c

```

::fixupdemo.c
__declspec(dllexport) void meme(int x)
{
    static int a=0x666;
    a=x;
}
> cl fixupdemo.c /LD

```

Откомпилируем и тут же дизассемблируем его: DUMPBIN /DISASM fixupdemo.dll и DUMPBIN /SECTION:.data /RAWDATA.

```

10001000: 55                push     ebp
10001001: 8B EC            mov      ebp,esp
10001003: 8B 45 08         mov      eax,dword ptr [ebp+8]
10001006: A3 30 50 00 10   mov      [10005030],eax
                ~~~~~
1000100B: 5D                pop      ebp
1000100C: C3                ret

RAW DATA #3
10005000: 00 00 00 00 00 00 00 00 00 00 00 00 33 24 00 10 .....3$.
10005010: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
10005020: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
10005030: 66 06 00 00 E3 11 00 10 FF FF FF FF 00 00 00 00 f...y...
                ~~~~~

```

Судя по коду, запись содержимого EAX всегда происходит в ячейку 0x10005030. Но не торопитесь с выводами! DUMPBIN /RELOCATIONS fixupdemo.dll:

```

BASE RELOCATIONS #4
1000 RVA, 154 SizeOfBlock
    7 HIGHLOW
    ^
    1C HIGHLOW
    23 HIGHLOW
    32 HIGHLOW
    3A HIGHLOW

```

Таблица перемещаемых элементов-то не пуста! И первая же ее запись указывает на ячейку 0x100001007, полученную алгебраическим сложением смещения 0x7 с RVA-адресом 0x1000 и базовым адресом загрузки 0x10000000 (получите его с помощью DUMPBIN самостоятельно). Смотрим — ячейка 0x100001007 принадлежит инструкции MOV [0x10005030], EAX и указывает на самый старший байт непосредственного смещения. Вот это самое смещение и корректирует загрузчик в ходе подключения динамической библиотеки (разумеется, если в этом есть необходимость).

Хотите проверить? Пожалуйста, создадим две копии одной DLL (например, copy fixupdemo.dll fixupdemo2.dll) и загрузим их поочередно следующей программой:

Листинг 4. Исходный текст fixupload

```
::fixupload.c
#include <windows.h>

main()
{
    void (*demo) (int a);
    HMODULE h;
    if ((h=LoadLibrary("fixupdemo.dll")) &&
        (h=LoadLibrary("fixupdemo2.dll")) &&
        (demo=(void (*)(int a))GetProcAddress(h, "meme")))
        demo(0x777);
}
> cl fixupload
```

Поскольку по одному и тому же адресу две различные DLL не загрузишь (откуда же системе знать, что это одна и та же DLL!), загрузчику приходится прибегать к ее перемещению. Загрузим откомпилированную программу в отладчик и установим точку останова на функцию LoadLibraryA. Это, понятное дело, необходимо, чтобы пропустить Startup-код и попасть в тело функции main. (Как легко убедиться, исполнение программы начинается отнюдь не с main, а со служебного кода, в котором очень легко утонуть.) Но откуда взялась загадочная буква «А», на конце имени функции? Ее происхождение тесно связано с введением в Windows поддержки уникада — специальной кодировки, каждый символ в которой кодируется двумя байтами, благодаря чему приобретает способность выражать любой из $2^{16} = 65536$ знаков, — количество, достаточное для вмещения практически всех алфавитов нашего мира. Применительно к LoadLibrary — теперь имя библиотеки может быть написано на любом языке, а при желании и на любом количестве любых языков одновременно, например на русско-французско-китайском. Звучит заманчиво, но не ухудшает ли это производительность? Разумеется, ухудшает, еще как — уникад требует жертв! Самое обидное — в подавляющем большинстве случаев вполне достаточно старой доброй кодировки ASCII (во всяком случае, нам, русским и американцам). Так какой же смысл бросать драгоценные такты процессора на ветер? Ради производительности было решено поступиться размером, создав отдельные варианты функций для работы с уникадом и ASCII-символами. Первые получили суффикс «W» (от *Wide* — *широкий*), а вторые — «A» (от ASCII). Эта тонкость скрыта от прикладных программистов. Какую именно функцию вызывать — «W» или «A» — решает компилятор, но при работе с отладчиком необходимо указывать точное имя функции — самостоятельно определить суффикс он не в состоянии. Камень преткновения в том, что некоторые функции, например ShowWindows, вообще не имеют суффиксов — ни «A», ни «W», и их библиотечное имя совпадает с каноническим. Как же быть?

Самое простое — заглянуть в таблицу импорта препарируемого файла и отыскать там вашу функцию. Например, применительно к нашему случаю:

```
> DUMPBIN /IMPORTS fixupload.exe > filename
> type filename
```



```

19D HeapDestroy
1C2 LoadLibraryA
CA GetCommandLineA
174 GetVersion
7D ExitProcess
29E TerminateProcess

```

...

Из приведенного выше фрагмента видно, что LoadLibrary все-таки «А», а вот функции ExitProcess и TerminateProcess не имеют суффиксов, поскольку вообще не работают со строками.

Другой путь — заглянуть в SDK. Конечно, библиотечное имя функций в нем отсутствует, но в Quick Info мимоходом приводится информация о поддержке уникада (если таковая присутствует). А раз есть уникад — есть суффиксы «W» и «A», соответственно наоборот — где нет уникада, нет и суффиксов. Проверим?

Вот так выглядит Quick Info от LoadLibrary:

QuickInfo

```

Windows NT: Requires version 3.1 or later.
Windows: Requires Windows 95 or later.
Windows CE: Requires version 1.0 or later.
Header: Declared in winbase.h.
Import Library: Use kernel32.lib.
Unicode: Implemented as Unicode and ANSI versions on Windows NT.

```

На чистейшем английском языке здесь сказано — *«Реализовано как Unicode и ANSI версии на Windows NT»*. Стоп! С NT все понятно, а как насчет «народной» девяносто восьмой (пятой)? Беглый взгляд на таблицу экспорта KERNEL32.DLL показывает: такая функция там есть, но, присмотревшись повнимательнее, мы с удивлением обнаружим, что ее точка входа совпадает с точками входа десятка других функций!

```

ordinal hint RVA      name
556 1B3 00039031 LoadLibraryW

```

Третья колонка в отчете DUMPBIN — это RVA-адрес — виртуальный адрес начала функции за вычетом базового адреса загрузки файла. Простой контекстный поиск показывает, что он встречается не единожды. Воспользовавшись программой-фильтром srcln для получения связного протокола, мы увидим следующее:

```

21:      118      1 00039031 AddAtomW
116:     217     60 00039031 DeleteFileW
119:     220     63 00039031 DisconnectNamedPipe
178:     279     9E 00039031 FindAtomW
204:     305     B8 00039031 FreeEnvironmentStringsW
260:     361     F0 00039031 GetDriveTypeW
297:     398    115 00039031 GetModuleHandleW
341:     442    141 00039031 GetStartupInfoW
377:     478    165 00039031 GetVersionExW
384:     485    16C 00039031 GlobalAddAtomW
389:     490    171 00039031 GlobalFindAtomW

```

```

413:      514 189 00039031 HeapLock
417:      518 18D 00039031 HeapUnlock
440:      541 1A4 00039031 IsProcessorFeaturePresent
455:      556 1B3 00039031 LoadLibraryW
508:      611 1E8 00039031 OutputDebugStringW
547:      648 20F 00039031 RemoveDirectoryW
590:      691 23A 00039031 SetComputerNameW
592:      693 23C 00039031 SetConsoleCP
597:      698 241 00039031 SetConsoleOutputCP
601:      702 245 00039031 SetConsoleTitleW
605:      706 249 00039031 SetCurrentDirectoryW
645:      746 271 00039031 SetThreadLocale
678:      779 292 00039031 TryEnterCriticalSection

```

Вот это сюрприз! Все уникадовые функции под одной крышей! Поскольку трудно поверить в идентичность реализаций LoadLibraryW и, скажем, DeleteFileW, остается предположить, что мы имеем дело с «заглушкой», которая ничего не делает, а только возвращает ошибку. Следовательно, в 9х функция LoadLibraryW действительно не реализована.

Но вернемся к нашим баранам, от которых нам пришлось так далеко отойти. Итак, вызываем отладчик, ставим бряк на LoadLibraryA, выходим из отладчика и терпеливо ждем его всплытия. Долго ждать, к счастью, не приходится...

KERNEL32! LoadLibraryA

```

001B:77E98023 PUSH    EBP
001B:77E98024 MOV     EBP, ESP
001B:77E98026 PUSH    EBX
001B:77E98027 PUSH    ESI
001B:77E98028 PUSH    EDI
001B:77E98029 PUSH    77E98054
001B:77E9802E PUSH    DWORD PTR [EBP+08]

```

Отдаем команду **P RET** для выхода из LoadLibraryA (анализировать ее, в самом деле, ни к чему), и оказываемся в легко узнаваемом теле функции main:

```

001B:0040100B CALL    [KERNEL32! LoadLibraryA]
001B:00401011 MOV     [EBP-08], EAX
001B:00401014 CMP     DWORD PTR [EBP-08], 00
001B:00401018 JZ      00401051
001B:0040101A PUSH    00405040
001B:0040101F CALL    [KERNEL32! LoadLibraryA]
001B:00401025 MOV     [EBP-08], EAX
001B:00401028 CMP     DWORD PTR [EBP-08], 00

```

Обратите внимание на содержимое регистра EAX — функция возвратила в нем адрес загрузки (на моем компьютере равный 0x10000000). Продолжая трассировку (клавиша <F10>), дождитесь выполнения второго вызова LoadLibraryA. Не правда ли, на этот раз адрес загрузки изменился? (На компьютере автора он равен 0x0530000.)

Приблизившись к вызову функции demo (в отладчике это выглядит как PUSH 00000777\ CALL [EBP-04], EBP-04 ни о чем нам не говорит, но вот аргумент 0x777 определенно что-то нам напоминает. См. исходный текст fixupload.c),

не забудьте переставить палец с клавиши <F10> на клавишу <F8>, чтобы войти внутрь функции.

001B:00531000	55	PUSH	EBP
001B:00531001	8BEC	MOV	EBP, ESP
001B:00531003	8B4508	MOV	EAX, [EBP+08]
001B:00531006	A330505300	MOV	[00535030], EAX
001B:0053100B	5D	POP	EBP
001B:0053100C	C3	RET	

Вот оно! Системный загрузчик скорректировал адрес ячейки согласно базовому адресу загрузки самой DLL. Это, конечно, хорошо, да вот проблема — в оригинальной DLL нет ни такой ячейки, ни даже последовательности A3 30 50 53 00, в чем легко убедиться, произведя контекстный поиск. Допустим, вознамерились бы мы затереть эту команду NOP-ами. Как найти это место в оригинальной DLL?

Обратим свой взор выше, на команды, заведомо не содержащие перемещаемых элементов, — PUSH EBP/MOV EBP, ESP/MOV EAX, [EBP+08]. Отчего бы не поискать последовательность 55 8B EC xxx A3? В данном случае это сработает, но, если бы перемещаемые элементы были густо перемешаны с «нормальными», ничего бы не вышло. Опорная последовательность оказалась бы слишком короткой для поиска и выдала бы множество ложных срабатываний.

Более изящно и надежно вычислить истинное содержимое перемещаемых элементов, вычтя из них разницу между действительным и рекомендуемым адресом загрузки. В данном случае: $0x535030 / \text{модифицированный загрузчиком адрес} / - (0x530000 / \text{базовый адрес загрузки} / - 0x10000000 / \text{рекомендуемый адрес загрузки} /) == 0x10005030$. Учитывая обратный порядок следования байтов, получаем, что инструкция MOV [10005030], EAX в машинном коде должна выглядеть так: A3 30 50 00 10. Ищем ее HIEW'ом, и чудо — она есть!

Способ 1. Прямой поиск введенного пароля в памяти

Был бы омут, а черти будут.

Народная поговорка

Пароль, хранящийся в теле программы открытым текстом, — скорее из ряда вон выходящее исключение, чем правило. К чему услуги хакера, если пароль и без того виден невооруженным взглядом? Поэтому разработчики защиты всячески пытаются скрыть его от посторонних глаз (о том, как именно они это делают, мы поговорим позже). Впрочем, учитывая размер современных пакетов, программист может без особого труда поместить пароль в каком-нибудь завалившемся файле, попутно снабдив его «крякушами» — строками, выглядящими как пароль, но паролем не являющимися. Попробуй разберись, где тут липа, а где нет, тем паче что подходящих на эту роль строк в проекте средней величины может быть несколько сотен, а то и тысяч!

Давайте подойдем к решению проблемы от обратного — будем искать не оригинальный пароль, который нам неизвестен, а ту строку, которую мы скормили программе в качестве пароля. А найдя, установим на нее бряк, и дальше все точно так же, как и раньше. Бряк всплывает на обращение по сравнению, мы выходим из сравнивающей процедуры, корректируем JMP, и...

Взглянем еще раз на исходный текст ломаемого нами примера `simple.c`:

```
for(;;)
{
    printf("Enter password:");
    fgets(&buff[0], PASSWORD_SIZE, stdin);

    if (strcmp(&buff[0], PASSWORD))
        printf("Wrong password\n");
    else break;
    if (++count>2) return -1;
}
```

Обратите внимание — в `buff` читается введенный пользователем пароль, сравнивается с оригиналом, затем (при неудачном сравнении) запрашивается еще раз, но (!) при этом `buff` не очищается! Отсюда следует, что, если после выдачи ругательства `Wrong password` вызвать отладчик и пройтись по памяти контекстным поиском, можно обнаружить тот заветный `buff`, а остальное уже — дело техники!

Итак, приступим (мы еще не знаем, во что мы ввязываемся, — но, увы, в жизни все сложнее, чем в теории). Запускаем `SIMPLE.EXE`, вводим любой пришедший на ум пароль (например, `KPNC Kaspersky++`), пропускаем возмущенный вопль `Wrgong` мимо ушей и нажимаем **<Ctrl-D>** — «горячую» комбинацию клавиш для вызова Айса. Так, теперь будем искать? Подождите, не надо бежать впереди лошадей: `Windows 9x\NT` — это не `Windows 3.x` и тем более не `MS-DOS` с единым адресным пространством для всех процессоров. Теперь, по соображениям безопасности, дабы один процесс ненароком не залез во владения другого, каждому из них предоставляется собственное адресное пространство. Например, у процесса `A` по адресу `23:0146660` может быть записано число `0x66`, у процесса `B` по *тому же самому адресу* `23:0146660` может находиться `0x0`, а у процесса `C` и вовсе третье значение. При этом процессы `A`, `B` и `C` не будут даже подозревать о существовании друг друга (ну, разве что воспользуются специальными средствами межпроцессорного взаимодействия).

Подробнее обо всем этом читайте у Хелен или Рихтера, здесь же нас больше заботит другое — вызванный по нажатию комбинации клавиш **<Ctrl-D>** отладчик всплывает в произвольном процессе (скорее всего, `Idle`) и контекстный поиск в памяти ничего не даст. Необходимо насильно переключить отладчик в необходимый контекст адресного пространства и лишь затем что-то предпринимать.

Из прилагаемой к Айсу документации можно узнать, что переключение контекстов осуществляется командой **ADDR**, за которой следует либо имя процесса, урезанное до восьми символов, либо его `PID`. Узнать и то, и другое можно с помощью другой команды — **PROC** (в том случае, если имя процесса синтаксически

неотличимо от PID, например 123, приходится использовать PID процесса — вторая колонка цифр слева в отчете PROC).

```
:addr simple
```

Отдаем команду *addr simple*, и... ничего не происходит, даже значения регистров остаются неизменными! Не волнуйтесь, все о'кей, что и подтверждает надпись «simple» в правом нижнем углу, идентифицирующая текущий процесс. А регистры... это небольшой глюк Айса. Он их игнорирует, переключая только адреса. В частности, поэтому трассировка переключенной программы невозможна. Вот поиск — другое дело. Это — пожалуйста!

```
:s 23:0 L -1 "KPNC Kaspersky"
```

Пояснения: первый слева аргумент после s — адрес, записанный в виде «селектор: смещение». Под Windows 2000 для адресации данных и стека используется селектор номер 23, в других операционных системах он может отличаться (и отличается!). Узнать его можно, загрузив любую программу и списав содержимое регистра DS. *Смещение* — вообще-то начинать поиск с нулевого смещения — идея глупая. Судя по карте памяти, здесь расположен служебный код и искомого пароля быть не может. Впрочем, это ничему не вредит, и так гораздо быстрее, чем разбираться, с какого адреса загружена программа и откуда именно начинать поиск. Третий аргумент — *L-1* — длина региона для поиска. -1, как нетрудно догадаться, — поиск «до победы». Далее, обратите внимание, что мы ищем не всю строку, а только ее часть (*KPNC Kaspersky++ против KPNC Kaspersky*). Это позволяет избавиться от ложных срабатываний — Айс любит выдавать ссылки на свои внутренние буфера, содержащие шаблон поиска. Вообще-то они всегда расположены выше 0x80000000, там, где никакой нормальный пароль не живет, но все же будет нагляднее, если на неполной подстроке будет находиться именно наша строка.

```
Pattern found at 0023:00016E40 (00016E40)
```

Так, по крайней мере, одно вхождение уже найдено. Но вдруг в памяти есть еще несколько? Проверим это, последовательно отдавая команды s вплоть до выдачи сообщения Pattern not found или превышения адреса поиска 0x80000000.

```
:s
Pattern found at 0023:0013FF18 (0013FF18)
:s
Pattern found at 0023:0024069C (0024069C)
:s
Pattern found at 0023:80B83F18 (80B83F18)
```

Целых два вхождения, да еще одно «в уме» — итого *три*! Не много ли для нас, начинающих? Во-первых, неясно — вводимые пароли, они что, плодятся аки кролики? Во-вторых, ну не ставить же все три точки останова. В данном случае четырех отладочных регистров процессора хватит, а как быть, если бы мы нашли десяток вхождений? Да и в трех бряках немудрено заблудиться с непривычки!

Итак, начинаем работать головой. Вхождений много, вероятнее всего, потому, что при чтении ввода с клавиатуры символы сперва попадают в системные бу-

феры, которые и дают ложные срабатывания. Звучит вполне правдоподобно, но вот как отфильтровать помехи?

На помощь приходит карта памяти — зная владельца региона, которому принадлежит буфер, можно очень многое сказать об этом буфере. Наскоро набив команду `map32 simple`, мы получим приблизительно следующее:

```
:map32 simple
Owner   Obj Name  Obj#  Address          Size      Type
simple   .text     0001  001B:00011000    00003F66  CODE R0
simple   .rdata    0002  0023:00015000    0000081E  IDATA R0
simple   .data     0003  0023:00016000    00001E44  IDATA RW
```

Ура, держи Тигру за хвост, есть одно отождествление! Буфер на 0x16E40 принадлежит сегменту данных, и, видимо, это и есть то, что нам нужно. Но не стоит спешить! Все не так просто. Поищем-ка адрес 0x16E40 в самом файле `simple.exe` (учитывая обратный порядок байтов, это будет 40 E6 01 00):

```
> dumpbin /SECTION:.data /RAWDATA simple.exe
RAW DATA #3
00016030: 45 6E 74 65 72 20 70 61 73 73 77 6F 72 64 3A 00  Enter password:.
00016040: 6D 79 47 4F 4F 44 70 61 73 73 77 6F 72 64 0A 00  myG00Dpassword..
00016050: 57 72 6F 6E 67 20 70 61 73 73 77 6F 72 64 0A 00  Wrong password..
00016060: 50 61 73 73 77 6F 72 64 20 4F 4B 0A 00 00 00 00  Password OK.....
00016070: 40 6E 01 00 00 00 00 00 40 6E 01 00 01 01 00 00  @n.....@n.....
00016080: 00 00 00 00 00 00 00 00 00 10 00 00 00 00 00 00  .....
```

Есть, да? Даже два раза! Посмотрим теперь, кто на него ссылается, попробуем найти в дизассемблированном тексте подстроку «16070» — адрес первого двойного слова, указывающего на наш буфер.

```
00011032: 68 70 60 01 00      push     16070h      ; <<<
00011037: 6A 64              push     64h        ; Макс. длина пароля (== 100 dec)
00011039: 8D 4D 98          lea      ecx,[ebp-68h]
                                ; Указатель ^^^^^ на буфер, куда
                                ; записывать пароль.
0001103C: 51              push     ecx
0001103D: E8 E2 00 00 00     call    00011124     ; fgets
00011042: 83 C4 0C          add      esp,0Ch     ; Выталкиваем три аргумента.
```

В общем, все ясно, за исключением загадочного указателя на указатель 0x16070. Заглянув в MSDN, где описан прототип этой функции, мы обнаружим, что «таинственный незнакомец» — указатель на структуру `FILE` (аргументы по С-соглашению, как мы помним, заносятся в стек справа налево). Первый член структуры `FILE` — указатель на буфер (файловый ввод-вывод в стандартной библиотеке С буферизован, и размер буфера по умолчанию составляет 4 кбайт). Таким образом, адрес 0x16E40 — это указатель на служебный буфер, и из списка «кандидатов в мастера» мы его вычеркиваем.

Двигаемся дальше. Претендент номер два — 0x24069C. Легко увидеть, что он выходит за пределы сегмента данных, и вообще непонятно, чему принадлежит. Почесав затылок, мы вспомним о такой «вкусности» Windows, как *куча* (*heap*). Посмотрим, что у нас там...

```
:heap 32 simple
Base      Id  Cmnt/Psnt/Rsvd  Segments  Flags      Process
00140000  01  0003/0003/00FD      1  00000002  simple
00240000  02  0004/0003/000C      1  00008000  simple
00300000  03  0008/0007/0008      1  00001003  simple
```

Ну, Тигр, давай на счастье хвост! Есть отождествление! Остается выяснить, кто выделил этот блок памяти: система под какие-то свои нужды или же сам программист. Первое, что бросается в глаза, — какой-то подозрительно-странный недокументированный флаг 0x8000. Заглянув в WINNT.H можно даже найти его определение, которое, впрочем, мало чем нам поможет, разве что наметит на системное происхождение оного.

```
#define HEAP_PSEUDO_TAG_FLAG 0x8000
```

А чтобы окончательно укрепить нашу веру, загрузим в отладчик любое подвернувшееся под лапу приложение и тут же отдадим команду *heap 32 proc_name*. Смотрите — система автоматически выделяет из кучи три региона! Точь-в-точь такие, как и в нашем случае. О'кей, значит, и этот кандидат ушел лесом.

Остается последний адрес — 0x13FF18. Ничего он не напоминает? Постойка, постой... Какое было значение ESP при загрузке?! Кажется, 0x13FFC4 или около того (внимание, в Windows 9x стек расположен совершенно в другом месте, но все рассуждения справедливы и для нее — необходимо лишь помнить местоположение стека в собственной операционной системе и уметь навскидку его узнать).

Поскольку стек растет снизу вверх (т. е. от старших адресов к младшим), адрес 0x13FF18 явно находится в стеке, а потому очень сильно похож на наш буфер. Уверенность подогревает тот факт, что большинство программистов размещает буферы в локальных переменных, ну а локальные переменные, в свою очередь, размещаются компилятором в стеке.

Ну что, попробуем установить сюда бряк?

```
:bpm 23:13FF18
:x
Break due to BPMB #0023:0013FF18 RW DR3 (ET=369.65 microseconds)
MSR LastBranchFromIp=0001144F
MSR LastBranchToIp=00011156

001B:000110B0  MOV     EAX,[EDX]
001B:000110B4  JNZ     000110E4
001B:000110B6  OR      AL,AL
001B:000110B8  JZ      000110E0
001B:000110BA  CMP     AH,[ECX+01]
001B:000110BD  JNZ     000110E4
001B:000110BF  OR      A H,AH
```

И вот мы в теле уже хорошо нам знакомой (развивайте зрительную память!) процедуры сравнения. На всякий случай, для пушей убежденности, выведем значение указателей EDX и ECX, чтобы узнать, что с чем сравнивается:


```
:d edx
0023:0013FF18 4B 50 4E 43 2D 2D 0A 00-70 65 72 73 6B 79 2B 2B KPNC Kaspersky++

:d ecx
0023:00016040 6D 79 47 4F 4F 44 70 61-73 73 77 6F 72 64 0A 00 myG00Dpassword..
```

Ну а остальное мы уже проходили. Выходим из сравнивающей процедуры по RET, находим условный переход, записываем его адрес (ключевую последовательность для поиска), правим исполняемый файл, и все о'кей.

Итак, мы познакомились с одним более или менее универсальным способом взлома защит, основанных на сравнении пароля (позже мы увидим, что он также подходит и для защит, основанных на регистрационных номерах). Его основное достоинство — простота. А недостатки... недостатков у него много:

- если программист очистит буфера после сравнения, поиск введенного пароля ничего не даст, разве что останутся системные буферы, которые так просто не затрешь, но отследить перемещения пароля из системных буферов в локальные не так-то просто!

- ввиду изобилия служебных буферов очень трудно определить, какой из них «настоящий». Программист же может располагать буфер и в сегменте данных (статический буфер), и в стеке (локальный буфер), и в куче, и даже выделять память низкоуровневыми вызовами типа VirtualAlloc или... да мало ли как разыграется его фантазия. В результате подчас приходится просеивать все найденные вхождения тупым перебором.

В качестве тренировки разберем другой пример — crackme01. Это то же самое, что и simple.exe, только с GUI'рым интерфейсом, и его ключевая процедура выглядит так:

Листинг 5. Исходный текст ядра защитного механизма crackme01

```
void CCrackme_01Dlg::OnOK()
{
    char buff[PASSWORD_SIZE];

    m_password.GetWindowText(&buff[0], PASSWORD_SIZE);
    if (strcmp(&buff[0], PASSWORD))
    {
        MessageBox("Wrong password");
        m_password.SetSel(0, -1, 0);
        return;
    }
    else
    {
        MessageBox("Password OK");
    }
    CDialog::OnOK();
}
```

Кажется, никаких сюрпризов не предвидится. Что ж, вводим пароль (как обычно, KPNC Kaspersky++), выслушиваем «ругательство» и, до нажатия кнопки **ОК**, вызываем отладчик, переключаем контекст...

```
:s 23:0 L -1 'KPNC Kaspersky'
Pattern found at 0023:0012F9FC (0012F9FC)
:s
Pattern found at 0023:00139C78 (00139C78)
```

Есть два вхождения! И оба лежат в стеке. Подбросим монетку, чтобы определить, с какого из них начать. (Правильный ответ — с первого.) Устанавливаем точку останова и терпеливо ждем всплытия отладчика. Всплытие ждать себя не заставляет, но показывает какой-то странный, откровенно «левый» код. Жмем **x** для выхода, следует целый каскад всплытий, одно непонятнее другого.

Лихорадочно подергивая бородку (варианты — усики, волосы в разных местах), соображаем: функция `CCrackme_01Dlg::OnOK` вызывается непосредственно в момент нажатия на кнопку **OK** — ей отводится часть стекового пространства под локальные переменные, которая автоматически «экспроприируется» при выходе из функции, переходя во всеобщее пользование. Таким образом, локальный буфер с введенным нами паролем существует только в момент его проверки, а потом автоматически затирается.

Единственная зацепка — модальный диалог с ругательством. Пока он на экране, буфер еще содержит пароль и его можно найти в памяти. Но это не сильно помогает в отслеживании, когда к этому буферу произведет обращение... Приходится терпеливо ждать, отсеивая ложные всплытия одно за другим. Наконец, в окне данных появляется искомая строка, а в окне кода — какой-то осмысленный код:

```
0023:0012F9FC 4B 50 4E 43 20 4B 61 73-70 65 72 73 6B 79 2B 2B KPNC Kaspersky++
0023:0012FA0C 00 01 00 00 0D 00 00 00-01 00 1C C0 A8 AF 47 00 .....G.
0023:0012FA1C 10 9B 13 00 78 01 01 00-F0 3E 2F 00 00 00 00 00 ....x...>/....
0023:0012FA2C 01 01 01 00 83 63 E1 77-F0 AD 47 00 78 01 01 00 .....c.w..G.x...
```

```
001B:004013E3 8A10 MOV DL,[EAX]
001B:004013E5 8A1E MOV BL,[ESI]
001B:004013E7 8ACA MOV CL,DL
001B:004013E9 3AD3 CMP DL,BL
001B:004013EB 751E JNZ 0040140B
001B:004013ED 84C9 TEST CL,CL
001B:004013EF 7416 JZ 00401407
001B:004013F1 8A5001 MOV DL,[EAX+01]
```

На всякий пожарный смотрим, на что указывает ESI:

```
:d esi
0023:0040303C 4D 79 47 6F 6F 64 50 61-73 73 77 6F 72 64 00 00 MyGoodPassword..
```

Остается «пропадать» исполняемый файл, и тут (как и следовало ожидать по закону бутерброда) нас ждут очередные трудности. Во-первых, хитрый компилятор оптимизировал код, подставив код функции `strcmp` вместо ее вызова, а во-вторых, условных переходов... да ими все кишит! Попробуй-ка найди нужный. На этот раз бросать монетку мы не станем, а попытаемся подойти к делу по-научному. Итак, перед нами дизассемблированный код, точнее, его ключевой фрагмент, осуществляющий анализ пароля:

```
>dumpbin /DISASM crackme_01.exe
```

```
004013DA: BE 3C 30 40 00      mov     esi,40303Ch
```

```
0040303C: 4D 79 47 6F 6F 64 50 61 73 73 77 6F 72 64 00 MyGoodPassword
```

В регистр ESI помещается указатель на оригинальный пароль.

```
004013DF: 8D 44 24 10        lea     eax,[esp+10h]
```

В регистр EAX — указатель на пароль, введенный пользователем.

```
004013E3: 8A 16              mov     dl,byte ptr [esi]
```

```
004013E5: 8A 1E              mov     bl,byte ptr [esi]
```

```
004013E7: 8A CA              mov     cl,dl
```

```
004013E9: 3A D3              cmp     dl,bl
```

Проверка первого символа на совпадение.

```
004013EB: 75 1E              jne     0040140B <---- (3) --> (1)
```

Первый символ уже не совпадает — дальше проверять бессмысленно!

```
004013ED: 84 C9              test     cl,cl
```

Первый символ первой строки равен нулю?

```
004013EF: 74 16              je       00401407 ----> (2)
```

Да, достигнут конец строки — значит, строки идентичны.

```
004013F1: 8A 50 01           mov     dl,byte ptr [eax+1]
```

```
004013F4: 8A 5E 01           mov     bl,byte ptr [esi+1]
```

```
004013F7: 8A CA              mov     cl,dl
```

```
004013F9: 3A D3              cmp     dl,bl
```

Проверяем следующую пару символов.

```
004013FB: 75 0E              jne     0040140B ----> (1)
```

Если не равна — конец проверке.

```
004013FD: 83 C0 02           add     eax,2
```

```
00401400: 83 C6 02           add     esi,2
```

Перемещаем указатели строк на два символа вперед.

```
00401403: 84 C9              test     cl,cl
```

Достигнут конец строки?

```
00401405: 75 DC              jne     004013E3 ----> (3)
```

Нет, еще не конец, сравниваем дальше.

```
00401407: 33 C0              xor     eax,eax <---- (2)
```

```
00401409: EB 05              jmp     00401410 ----> (4)
```

Обнуляем EAX (strcmp в случае успеха возвращает ноль) и выходим.

```
0040140B: 1B C0              sbb     eax,eax <---- (3)
```

```
0040140D: 83 D8 FF           sbb     eax,0FFFFFFFh
```

Эта ветка получает управление при несовпадении строк. EAX устанавливает равным в ненулевое значение (подумайте, почему).

```
00401410: 85 C0              test     eax,eax <---- (4)
```

Проверка значения EAX на равенство нулю.

```
00401412: 6A 00          push     0
00401414: 6A 00          push     0
```

Что-то заносим в стек...

```
00401416: 74 38          je       00401450 <<<< ---->(5)
```

Прыгаем куда-то...

```
00401418: 68 2C 30 40 00  push     40302Ch
0040302C: 57 72 6F 6E 67 20 70 61 73 73 77 6F 72 64 00 .Wrong password
```

Ага, «Вронг пысворд». Значит, прыгать все-таки надо.... Смотрим, куда указывает `je` (а код ниже уже не представляет интереса, и так ясно: это «матюгальщик»).

Теперь, когда алгоритм защиты в общих чертах ясен, можно ее и сломать, например поменяв условный переход в строке `0x401416` на безусловный `jump short` (код `0xEB`).

Способ 2. Бряк на функции ввода пароля

Вы боитесь творить, потому что творения ваши отражают вашу истинную суть.

Ф. Херберт. Ловец душ

При всем желании метод прямого поиска пароля в памяти элегантным назвать нельзя, да и практичным тоже. А, собственно, зачем искать сам пароль, спотыкаясь о беспорядочно разбросанные буфера, когда можно поставить бряк непосредственно на функцию, его считывающую? Хм, можно и так... да вот угадать, какой именно функцией разработчик вздумал читать пароль, вряд ли будет намного проще.

На самом деле одно и то же действие может быть выполнено всего лишь несколькими функциями и их перебор не займет много времени. В частности, содержимое окна редактирования обычно добывается либо при помощи функции `GetWindowTextA` (что чаще всего и происходит), либо функции `GetDlgItemTextA` (а это значительно реже).

Раз уж речь зашла об окнах, запустим наш GUI «крякмис» и установим точку останова на функцию `GetWindowTextA` (*bpx GetWinodwTextA*). Поскольку эта функция — системная, точка останова будет глобальной, т. е. затронет все приложения в системе, поэтому заблаговременно закройте все лишнее от греха подальше. Если установить бряк до запуска «крякмиса», то мы увидим несколько ложных всплывтий, возникающих вследствие того, что система сама читает содержимое окна в процессе формирования диалога.

Вводим какой-нибудь пароль (*KPNC Kaspersky++*, по обыкновению), нажимаем клавишу **<ENTER>** — и отладчик не замедливает всплыть:

```
USER32!GetWindowTextA
001B:77E1A4E2 55          PUSH EBP
001B:77E1A4E3 8BEC        MOV EBP,ESP
001B:77E1A4E5 6AFF        PUSH FF
001B:77E1A4E7 6870A5E177  PUSH 77E1A570
```

```

001B:77E1A4EC 68491DE677    PUSH 77E61D49
001B:77E1A4F1 64A100000000    MOV EAX,FS:[00000000]
001B:77E1A4F7 50              PUSH EAX

```

Во многих руководствах по взлому советуется тут же выйти из функции по команде `P RET`, мол, что ее анализировать-то, но не стоит спешить! Сейчас самое время выяснить, где расположен буфер вводимой строки и установить на него бряк. Вспомним, какие аргументы и в какой последовательности принимает функция (а если не вспомним, то заглянем в SDK):

```

int GetWindowText(
    HWND hWnd,           // handle to window or control with text
    LPTSTR lpString,     // address of buffer for text
    int nMaxCount        // maximum number of characters to copy
);

```

Может показаться, раз программа написана на языке C, то и аргументы заносятся в стек по C-соглашению. А вот и нет! Все API-функции Windows всегда вызываются по Паскаль-соглашению, на каком бы языке программа ни была написана. Таким образом, аргументы заносятся в стек слева направо, а последним в стек попадает адрес возврата. В 32-разрядной Windows все аргументы и сам адрес возврата занимают двойное слово (4 байта), поэтому, чтобы добраться до указателя на строку, необходимо к регистру указателю вершины стека (ESP) добавить восемь байтов (одно двойное слово на `nMaxCount`, другое — на сам `lpString`). Нагляднее это изображено на рис. 3.

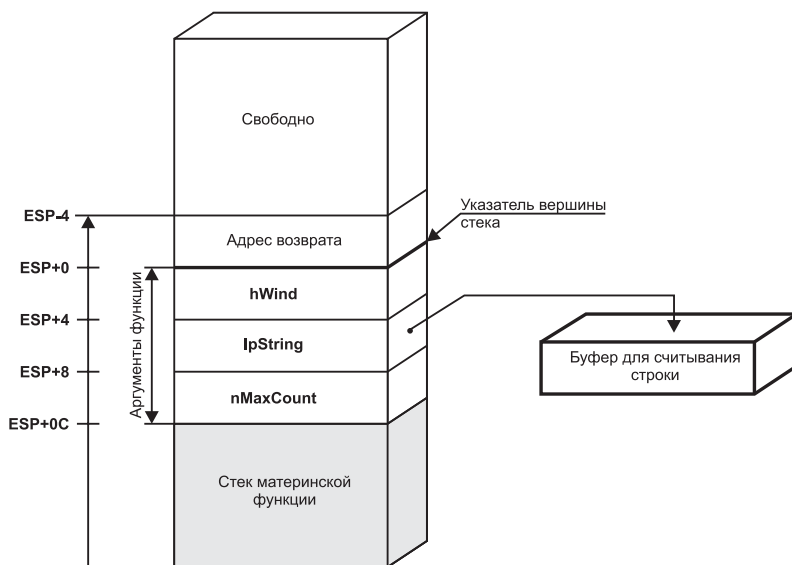


Рис. 3. Состояние стека на момент вызова `GetWindowsText`

Получить содержимое ячейки по заданному адресу в Айсе можно с помощью оператора «*», вызов которого в нашем случае выглядит так (подробнее см. документацию, прилагаемую к отладчику):

```
:d *(esp+8)
```

```
0023:0012F9FC 1C FA 12 00 3B 5A E1 77-EC 4D E1 77 06 02 05 00 .....;Z.w.M.w....
0023:0012FA0C 01 01 00 00 10 00 00 00-01 00 2A C0 10 A8 48 00 .....*...H.
0023:0012FA1C 10 9B 13 00 0A 02 04 00-E8 3E 2F 00 00 00 00 .....>/.....
0023:0012FA2C 01 02 04 00 83 63 E1 77-08 DE 48 00 0A 02 04 00 .....c.w..H.....
```

В буфере «мусор» — так и следовало ожидать — ведь строка еще не считана. Давайте выйдем из функции по команде `P RET` и посмотрим, что произойдет (только потом уже нельзя будет пользоваться конструкцией `d *esp+8`, так как после выхода из функции аргументы будут вытолкнуты из стека):

```
: p ret
```

```
:d 0012F9FC
```

```
0023:0012F9FC 4B 50 4E 43 20 4B 61 73-70 65 72 73 6B 79 2B 2B KPNC Kaspersky++
0023:0012FA0C 00 01 00 00 0D 00 00 00-01 00 1C 80 10 A8 48 00 .....H.
0023:0012FA1C 10 9B 13 00 0A 02 04 00-E8 3E 2F 00 00 00 00 .....>/.....
0023:0012FA2C 01 02 04 00 83 63 E1 77-08 DE 48 00 0A 02 04 00 .....c.w..H.....
```

ОК, это действительно тот буфер, который нам нужен. Ставим бряк на его начало и ожидаем всплывтия. Смотрите, с первого же раза мы очутились именно там, где и надо (узнаете код сравнивающей процедуры?):

```
001B:004013E3 8A10 MOV DL,[EAX]
001B:004013E5 8A1E MOV BL,[ESI]
001B:004013E7 8ACA MOV CL,DL
001B:004013E9 3AD3 CMP DL,BL
001B:004013EB 751E JNZ 0040140B
001B:004013ED 84C9 TEST CL,CL
001B:004013EF 7416 JZ 00401407
001B:004013F1 8A5001 MOV DL,[EAX+01]
```

Замечательно! Вот так, безо всяких ложных срабатываний, элегантно, быстро и красиво мы победили защиту!

Этот способ универсален, и впоследствии мы еще не раз им воспользуемся. Вся соль — определить ключевую функцию защиты и поставить на нее бряк. Под Windows все попользования (будь то обращения к ключевому файлу, реестру и т. д.) сводятся к вызову API-функций, перечень которых хотя и велик, но все же конечен и известен заранее.

Способ 3. Бряк на сообщения

Любая завершенная дисциплина имеет свои штампы, свои модели, свое влияние на обучающихся.

Ф. Херберт. Дюна

Если у вас еще не закружилась голова от количества выпитого во время хака пива, с вашего позволения мы продолжим. Каждый, кто хоть однажды программировал под Windows, наверняка знает, что в Windows все взаимодействие с окнами завязано на *сообщениях*. Практически все оконные API-функции на самом деле

представляют собой высокоуровневые «обертки», посылающие окну сообщения. Не является исключением и `GetWindowTextA` — аналог сообщения `WM_GETTEXT`.

Отсюда следует: чтобы считать текст из окна, вовсе не обязательно обращаться к `GetWindowTextA`, можно сделать это через **`SendMessageA(hWnd, WM_GETTEXT, (LPARAM) &buff[0])`**. Именно так и устроена защита в примере *crack 02*. Попробуйте загрузить его и установить бряк на `GetWindowTextA` (`GetDlgItemTextA`). Что, не срабатывает? Подобная мера используется разработчиками для запутывания совсем уж желторотых новичков, бегло изучивших пару *faq* по хаку и тут же бросившихся в бой.

Так, может, поставить бряк на `SendMessageA`? В данном случае в принципе можно, но бряк на сообщение `WM_GETTEXT` — более универсальное решение, срабатывающее независимо от того, как читают окно.

Для установки бряка на сообщение в Айсе предусмотрена специальная команда — **`BMSG`**, которой мы и пользовались в первом издании этой книги. Но не интереснее ли сделать это своими руками?

Как известно, с каждым окном связана специальная *оконная процедура*, обслуживающая это окно, т. е. отвечающая за прием и обработку сообщений. Вот если бы узнать ее адрес да установить на него бряк! И это действительно можно сделать! Специальная команда `HWND` выдает всю информацию об окнах указанного процесса.

<Ctrl-D>

:addr crack02

:hwnd crack02

Handle	Class	WinProc	TID	Module
050140	#32770 (Dialog)	6C291B81	2DC	crack02
05013E	Button	77E18721	2DC	crack02
05013C	Edit	6C291B81	2DC	crack02
05013A	Static	77E186D9	2DC	crack02

Быстро обнаруживает себя окно редактирования с адресом оконной процедуры `0x6C291B81`. Поставим сюда бряк? Нет, еще не время — ведь оконная процедура вызывается не только при чтении текста, а гораздо чаще. Как бы установить бряк на то, что нам нужно, отсеяв все остальные сообщения? Для начала изучим прототип этой функции:

```

LRESULT CALLBACK WindowProc(
    HWND hwnd,      // handle to window
    UINT uMsg,      // message identifier
    WPARAM wParam,  // first message parameter
    LPARAM lParam   // second message parameter
);

```

Как нетрудно подсчитать, в момент вызова функции аргумент `uMsg` — идентификатор сообщения — будет лежать по смещению 8 относительно указателя вершины стека `ESP`. Если он равен `WM_GETTEXT` (непосредственное значение `0xD`) — недурно бы всплыть!

Вот и настало время познакомиться с условными бряками. Подробнее об их синтаксисе рассказано в прилагаемой к отладчику документации. А впрочем, программисты, знакомые с языком C, вряд ли к ней обратятся, ибо синтаксис лаконичен и интуитивно понятен.

```
:7bpw 6C291B81 IF (esp->8)==WM_GETTEXT
:x
```

Выходим из отладчика, вводим какой-нибудь текст в качестве пароля, скажем, *Hello*, нажимаем на клавишу **<ENTER>** — отладчик тут же всплывает:

```
Break due to BPX #0008:6C291B81 IF ((ESP->8)==0xD) (ET=2.52 seconds)
```

Вот он, хвост Тигры, и уши плюшевого медведя! Остается определить адрес буфера, в который возвращается считанная строка. Начинаем соображать: указатель на буфер передается через аргумент `lParam` (см. в SDK описание `WM_GETTEXT`), а сам `lParam` размещается в стеке по смещению `0x10` относительно `ESP`:

```
адрес возврата ← ESP
hwnd           ← ESP + 0x4
uMsg           ← ESP + 0x8
wParam         ← ESP + 0xC
lParam         ← ESP + 0x10
```

Даем команду вывода этого буфера в окно данных, выходим из оконной процедуры по команде `P RET` и... видим только что введенный нами текст *Hello*:

```
:d *(esp+10)
:p ret
0023:0012EB28 48 65 6C 6C 6F 00 05 00-0D 00 00 00 FF 03 00 00 Hello.....
0023:0012EB38 1C ED 12 00 01 00 00 00-0D 00 00 00 FD 86 E1 77 .....w
0023:0012EB48 70 3C 13 00 00 00 00 00-00 00 00 00 00 00 00 00 p<.....
0023:0012EB58 00 00 00 00 00 00 00 00-98 EB 12 00 1E 87 E1 77 .....w

:bpm 23:12EB28
```

Установив точку останова, мы ловим одно откровенно «левое» всплытие отладчика (это видно по явно не юзерскому значению селектора `CS`, равного 8) и уже тянем руку, чтобы нажать на клавишу **<X>**, продолжив отслеживание нашего бряка, как вдруг краем глаза замечаем...

```
0008:A00B017C 8A0A          MOV     CL,[EDX]
0008:A00B017E 8808          MOV     [EAX],CL
0008:A00B0180 40           INC     EAX
0008:A00B0181 42           INC     EDX
0008:A00B0182 84C9         TEST    CL,CL
0008:A00B0184 7406         JZ      A00B018C
0008:A00B0186 FF4C2410     DEC     DWORD PTR [ESP+10]
0008:A00B018A 75F0         JNZ     A00B017C
```


Эге, буфер-то не сквозной, система не отдает его народу, а копирует в другой буфер. Это видно по тому, как из указателя на наш буфер `EDX` символ копируется в `CL` (то, что `EDX` — указатель на наш буфер, следует из того, что он вызвал всплытие отладчика), а из `CL` он копируется в `[EAX]`, где `EAX` какой-то ука-

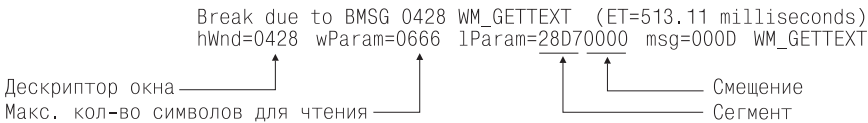
затель (о котором мы пока еще не можем сказать ничего определенного). Далее, оба указателя увеличиваются на единицу и CL (последний считанный символ) проверяется на равенство нулю. Если конец строки не достигнут, то все повторяется. Что ж, нам суждено следить сразу за двумя буферами — ставим еще один бряк.

```
:bpm EAX
:x
```

На втором бряке отладчик вскорости всплывает, и мы узнаем нашу родную процедуру сравнения. Ну а дальнейшее дело техники.

001B:004013F2	8A1E	MOV	BL,[ESI]
001B:004013F4	8ACA	MOV	CL,DL
001B:004013F6	3AD3	CMP	DL,BL
001B:004013F8	751E	JNZ	00401418
001B:004013FA	84C9	TEST	CL,CL
001B:004013FC	7416	JZ	00401414
001B:004013FE	8A5001	MOV	DL,[EAX+01]
001B:00401401	8A5E01	MOV	BL,[ESI+01]

 В Windows 9x обработка сообщений реализована несколько иначе, чем в NT. В частности, оконная процедура окна редактирования находится в 16-разрядном коде. А это сегментная модель памяти (треска хвостом вперед под хвост Тигре) а la сегмент:смещение. Представляется любопытным механизм передачи адреса: в какой же параметр засунут сегмент? Чтобы ответить на это, взглянем на отчет Айса:



Адрес целиком умещается в 32-разрядном аргументе lParam — 16-разрядный сегмент и 16-разрядное смещение. Посему точка останова должна выглядеть так: *bpm 28D7:0000*.

Шаг пятый. На сцене появляется IDA

Реальность такова, какой ее описывает язык.
*Б. Л. Уорф. Тезис лингвистической
относительности*

С легкой руки Дениса Ричи повелось начинать освоение нового языка программирования с создания простейшей программы «Hello, World!». И здесь не будет нарушена эта традиция. Оценим возможности IDA Pro следующим примером

(для совместимости с книгой рекомендуется откомпилировать его с помощью Microsoft Visual C++ 6.0 вызовом `cl.exe first.cpp` в командной строке):

```
#include <iostream.h>
void main()
{
    cout<<"Hello, Sailor!\n";
}
```

Компилятор сгенерирует исполняемый файл размером почти в 40 килобайтов, большую часть которого займет служебный, стартовый или библиотечный код! Попытка дизассемблирования с помощью таких дизассемблеров, как W32DASM (или аналогичных ему) не увенчается быстрым успехом, поскольку над полученным листингом размером **в пятьсот килобайтов (!)** можно просидеть не час и не два. Легко представить, сколько времени уйдет на серьезные задачи, требующие изучения десятков мегабайтов дизассемблированного текста.

Попробуем дизассемблировать эту программу с помощью IDA. Если все настройки оставить по умолчанию, после завершения анализа экран (в зависимости от версии) должен выглядеть следующим образом:

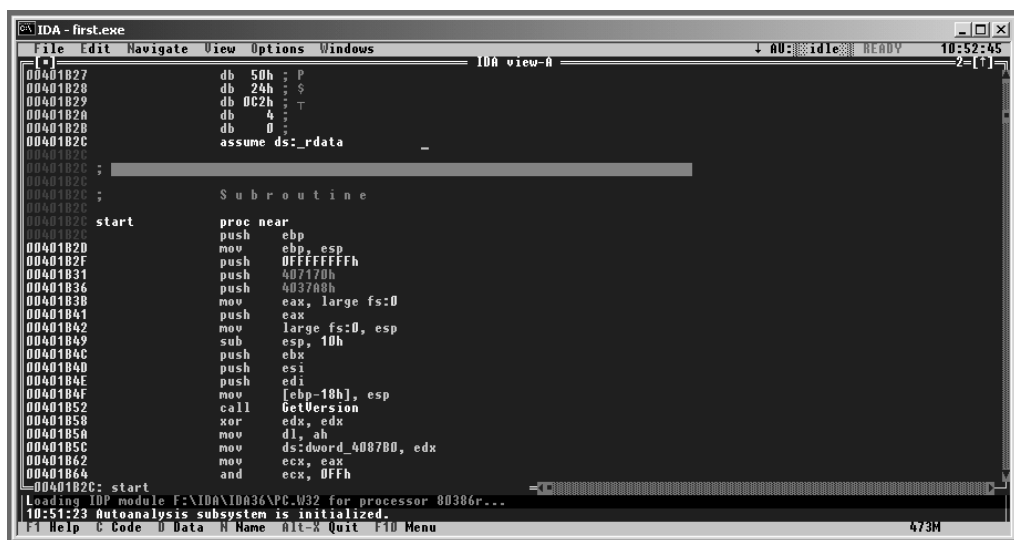


Рис. 4. Так выглядит результат работы консольной версии IDA Pro 3.6

С версии 3.8x² в IDA появилась поддержка **сворачивания** (Collapsed) функций. Такой прием значительно упрощает навигацию по тексту, позволяя убрать с экрана неинтересные на данный момент строки. По умолчанию все библиотечные функции сворачиваются автоматически.

Развернуть функцию можно, подведя к ней курсор и нажав клавишу <+> на дополнительной цифровой клавиатуре, расположенной справа. Соответственно клавиша <-> предназначена для сворачивания.

² А может и чуточку раньше.

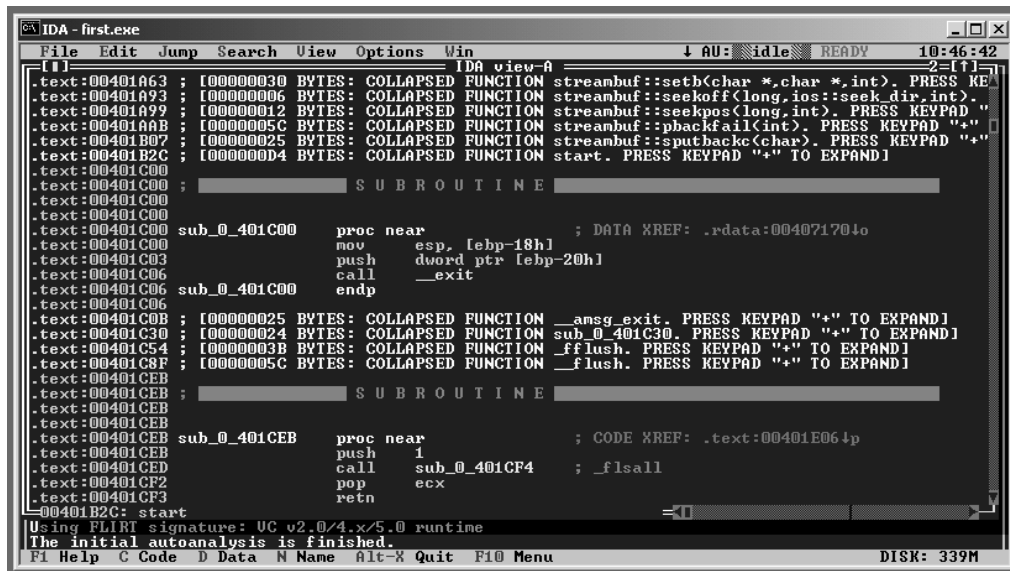


Рис. 5. Так выглядит результат работы консольной версии IDA Pro 4.6

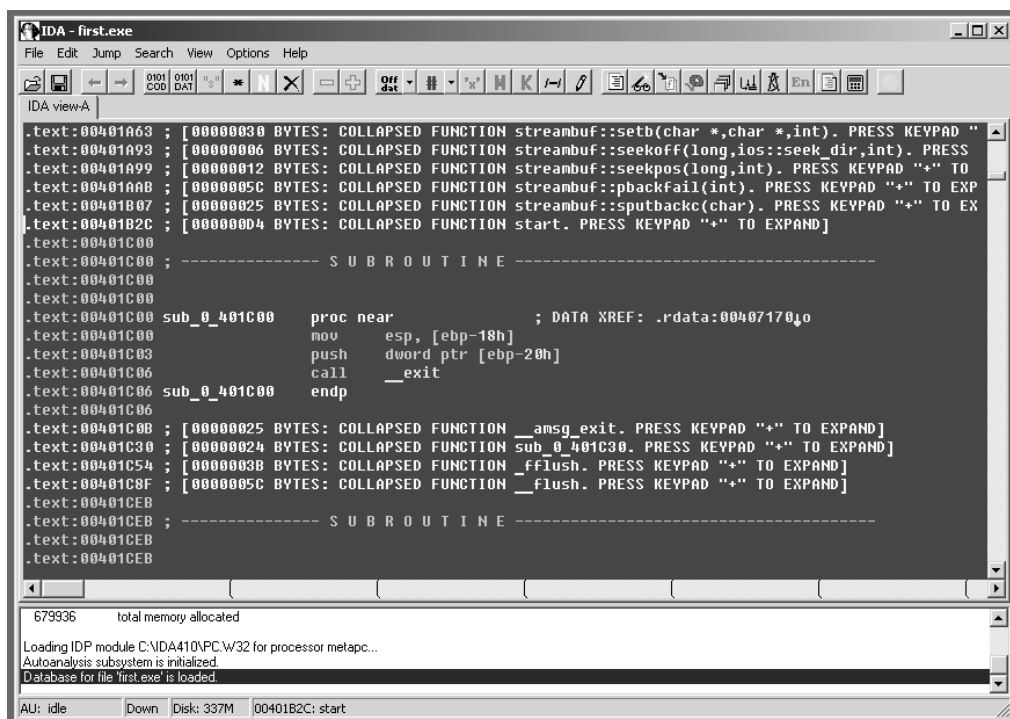


Рис. 6. Так выглядит результат работы графической версии IDA Pro 4.0

По окончании автоматического анализа файла `first.exe` IDA переместит курсор к строке `.text:00401B2C` — точке входа в программу. Среди начинающих программистов широко распространено заблуждение, якобы программы, написанные на языке C, начинают выполняться с функции `main`, но в действительности это не совсем так. На самом деле сразу после загрузки файла управление передается на функцию `Start`, вставленную компилятором. Она подготавливает глобальные переменные `_osver` (билд), `_winmajor` (старшая версия операционной системы), `_winminor` (младшая версия операционной системы), `_winver` (полная версия операционной системы), `__argc` (количество аргументов командной строки), `__argv` (массив указателей на строки аргументов), `_environ` (массив указателей на строки переменных окружения); инициализирует кучи (`heap`); вызывает функцию `main`, а после возвращения управления завершает процесс с помощью функции `Exit`.

Наглядно продемонстрировать инициализацию переменных, совершаемую стартовым кодом, позволяет следующая программа (исходный текст программы `Crt0.demo.c`):

```
#include <stdio.h>
#include <stdlib.h>
void main()
{
    int a;
    printf(">Версия OS:\t\t%d.\td\n\
>Билд:\t\t%d\n\
>Количество аргументов:\td\n", \
        _winmajor, _winminor, _osver, __argc);
    for (a=0; a<__argc; a++)
        printf(">\tАргумент %02d:\t\t%s\n", a+1, __argv[a]);
    a=!a-1;
    while(_environ[++a]) ;
    printf(">Количество переменных окружения:\td\n", a);
    while(a) printf(">\tПеременная %d:\t\t%s\n", a, _environ[-a]);
}
```

Прототип функции `main` как будто указывает, что приложение не принимает никаких аргументов командной строки, но результат работы программы доказывает обратное и на машине автора выглядит так (приводится в сокращенном виде):

```
>Версия OS:                               5.0
>Билд:                                     2195
>Количество аргументов:                   1
>    Аргумент      01:                     Crt0.demo
>Количество переменных окружения:         30
>    Переменная   29:                     windir=C:\WINNT
>...
```

Очевидно, нет никакой необходимости анализировать стандартный стартовый код приложения, и первая задача исследователя — найти место передачи управления на функцию `main`. К сожалению, гарантированное решение этой задачи требует полного анализа содержимого функции `Start`. У исследователей суще-

ствуется множество хитростей, но все они базируются на особенностях реализации конкретных компиляторов³ и не могут считаться универсальными.

Рекомендуется изучить исходные тексты стартовых функций популярных компиляторов, находящиеся в файлах CRt0.c (Microsoft Visual C) и c0w.asm (Borland C), — это упростит анализ дизассемблерного листинга.

Ниже, в качестве иллюстрации, приводится содержимое стартового кода программы first.exe, полученное в результате работы W32Dasm:

```
//***** Program Entry Point *****
:00401B2C 55          push ebp
:00401B2D 8BEC       mov ebp, esp
:00401B2F 6AFF       push FFFFFFFF
:00401B31 6870714000 push 00407170
:00401B36 68A8374000 push 004037A8
:00401B3B 64A100000000 mov eax, dword ptr fs:[00000000]
:00401B41 50          push eax
:00401B42 64892500000000 mov dword ptr fs:[00000000], esp
:00401B49 83EC10     sub esp, 00000010
:00401B4C 53          push ebx
:00401B4D 56          push esi
:00401B4E 57          push edi
:00401B4F 8965E8     mov dword ptr [ebp-18], esp
```

Reference To: KERNEL32.GetVersion, Ord:0174h

```
|
:00401B52 FF1504704000 Call dword ptr [00407004]
:00401B58 33D2       xor edx, edx
:00401B5A 8AD4       mov dl, ah
:00401B5C 8915B0874000 mov dword ptr [004087B0], edx
:00401B62 8BC8       mov ecx, eax
:00401B64 81E1FF000000 and ecx, 000000FF
:00401B6A 890DAC874000 mov dword ptr [004087AC], ecx
:00401B70 C1E108     shl ecx, 08
:00401B73 03CA       add ecx, edx
:00401B75 890DA8874000 mov dword ptr [004087A8], ecx
:00401B7B C1E810     shr eax, 10
:00401B7E A3A4874000 mov dword ptr [004087A4], eax
:00401B83 6A00       push 00000000
:00401B85 E8D91B0000 call 00403763
:00401B8A 59         pop ecx
:00401B8B 85C0       test eax, eax
:00401B8D 7508       jne 00401B97
:00401B8F 6A1C       push 0000001C
:00401B91 E89A000000 call 00401C30
:00401B96 59         pop ecx
```

Referenced by a (U)nconditional or (C)onditional Jump at Address:

|:00401B8D(C)

³ Например, Microsoft Visual C всегда, независимо от прототипа функции main передает ей три аргумента — указатель на массив указателей переменных окружения, указатель на массив указателей аргументов командной строки и количество аргументов командной строки, а все остальные функции стартового кода принимают меньшее количество аргументов.

```
|
:00401B97 8365FC00          and dword ptr [ebp-04], 00000000
:00401B9B E8D70C0000         call 00402877
```

Reference To: KERNEL32.GetCommandLineA, Ord:00CAh

```
|
:00401BA0 FF1560704000       Call dword ptr [00407060]
:00401BA6 A3E49C4000       mov dword ptr [00409CE4], eax
:00401BAB E8811A0000       call 00403631
:00401BB0 A388874000       mov dword ptr [00408788], eax
:00401BB5 E82A180000       call 004033E4
:00401BBA E86C170000       call 0040332B
:00401BBF E8E1140000       call 004030A5
:00401BC4 A1C0874000       mov eax, dword ptr [004087C0]
:00401BC9 A3C4874000       mov dword ptr [004087C4], eax
:00401BCE 50              push eax
:00401BCF FF35B8874000       push dword ptr [004087B8]
:00401BD5 FF35B4874000       push dword ptr [004087B4]
:00401BDB E820F4FFFF       call 00401000
:00401BE0 83C40C          add esp, 0000000C
:00401BE3 8945E4          mov dword ptr [ebp-1C], eax
:00401BE6 50              push eax
:00401BE7 E8E6140000       call 004030D2
:00401BEC 8B45EC          mov eax, dword ptr [ebp-14]
:00401BEF 8B08          mov ecx, dword ptr [eax]
:00401BF1 8B09          mov ecx, dword ptr [ecx]
:00401BF3 894DE0          mov dword ptr [ebp-20], ecx
:00401BF6 50              push eax
:00401BF7 51              push ecx
:00401BF8 E8AA150000       call 004031A7
:00401BFD 59              pop ecx
:00401BFE 59              pop ecx
:00401BFF C3              ret
```

Иначе выглядит результат работы IDA, умеющей распознавать библиотечные функции по их сигнатурам (приблизительно по такому же алгоритму работает множество антивирусов). Поэтому способности дизассемблера тесно связаны с его версией и полнотой комплекта поставки — далеко не все версии IDA Pro в состоянии работать с программами, сгенерированными современными компиляторами. (Перечень поддерживаемых компиляторов можно найти в файле %IDA%/SIG/list.)

```
00401B2C start      proc near
00401B2C
00401B2C var_20      = dword ptr -20h
00401B2C var_1C      = dword ptr -1Ch
00401B2C var_18      = dword ptr -18h
00401B2C var_14      = dword ptr -14h
00401B2C var_4       = dword ptr -4
00401B2C
00401B2C          push    ebp
00401B2D          mov     ebp, esp
00401B2F          push    0FFFFFFFh
```



```

00401B31      push     offset stru_407170
00401B36      push     offset __except_handler3
00401B3B      mov      eax, large fs:0
00401B41      push     eax
00401B42      mov      large fs:0, esp
00401B49      sub      esp, 10h
00401B4C      push     ebx
00401B4D      push     esi
00401B4E      push     edi
00401B4F      mov      [ebp+var_18], esp
00401B52      call     ds:GetVersion
00401B58      xor      edx, edx
00401B5A      mov      dl, ah
00401B5C      mov      dword_4087B0, edx
00401B62      mov      ecx, eax
00401B64      and      ecx, 0FFh
00401B6A      mov      dword_4087AC, ecx
00401B70      shl      ecx, 8
00401B73      add      ecx, edx
00401B75      mov      dword_4087A8, ecx
00401B7B      shr      eax, 10h
00401B7E      mov      dword_4087A4, eax
00401B83      push     0
00401B85      call     __heap_init
00401B8A      pop      ecx
00401B8B      test     eax, eax
00401B8D      jnz      short loc_401B97
00401B8F      push     1Ch
00401B91      call     sub_401C30      ; _fast_error_exit
00401B96      pop      ecx
00401B97
00401B97 loc_401B97:                                ; CODE XREF: start+61↑j
00401B97      and      [ebp+var_4], 0
00401B9B      call     __ioint
00401BA0      call     ds:GetCommandLineA
00401BA6      mov      dword_409CE4, eax
00401BAB      call     ___crtGetEnvironmentStringsA
00401BB0      mov      dword_408788, eax
00401BB5      call     __setargv
00401BBA      call     __setenvp
00401BBF      call     __cinit
00401BC4      mov      eax, dword_4087C0
00401BC9      mov      dword_4087C4, eax
00401BCE      push     eax
00401BCF      push     dword_4087B8
00401BD5      push     dword_4087B4
00401BDB      call     sub_401000
00401BE0      add      esp, 0Ch
00401BE3      mov      [ebp+var_1C], eax
00401BE6      push     eax
00401BE7      call     _exit
00401BEC ; -----
00401BEC

```

```

00401BEC loc_401BEC:          ; DATA XREF: _rdata:00407170↓
00401BEC      mov     eax, [ebp-14h]
00401BEF      mov     ecx, [eax]
00401BF1      mov     ecx, [ecx]
00401BF3      mov     [ebp-20h], ecx
00401BF6      push    eax
00401BF7      push    ecx
00401BF8      call   __XcptFilter
00401BFD      pop     ecx
00401BFE      pop     ecx
00401BFF      retn
00401BFF start      endp ; sp = -34h

```

С приведенным примером IDA Pro успешно справляется, о чем свидетельствует строка Using FLIRT signature: VC v2.0/4.x/5.0 runtime в окне сообщений.

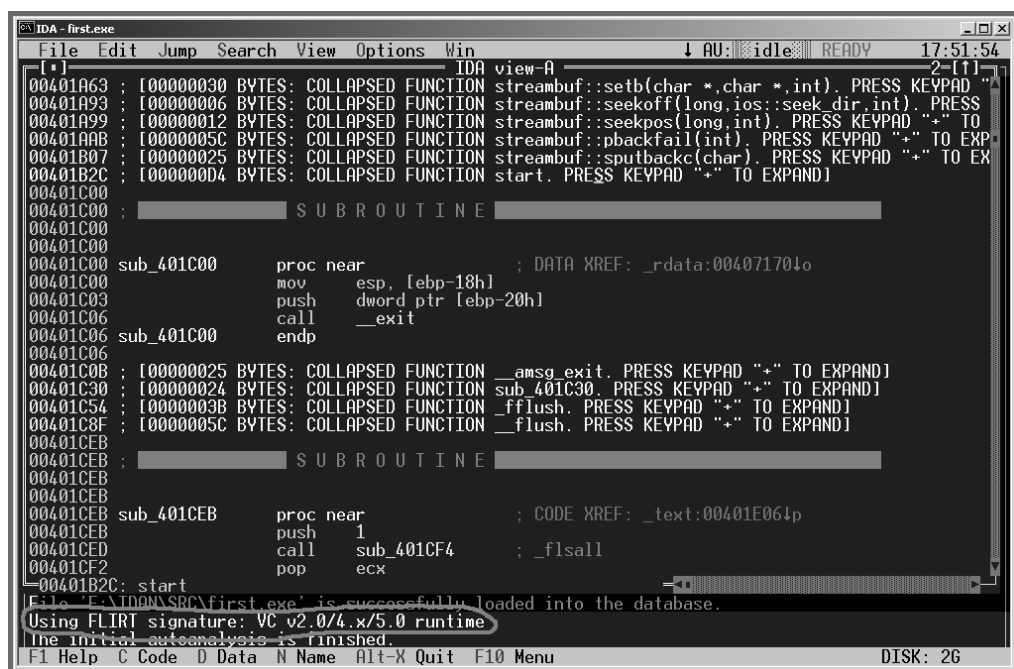


Рис. 7. Загрузка библиотеки сигнатур

Дизассемблер сумел определить имена всех функций, вызываемых стартовым кодом, за исключением одной, расположенной по адресу 0x0401BDB. Учитывая передачу трех аргументов и обращение к `_exit` после возвращения функцией управления, можно предположить, что это `main` и есть.

Перейти по адресу 0x0401000 для изучения содержимого функции `main` можно несколькими способами: прокрутить экран с помощью стрелок управления курсором, нажать клавишу **<G>** и ввести требуемый адрес в появившемся окне диалога, но проще и быстрее всего воспользоваться встроенной в IDA Pro системой навигации. Если поместить курсор в границы имени, константы или выражения и нажать клавишу **<Enter>**, IDA автоматически перейдет на требуемый адрес.

В данном случае требуется подвести курсор к строке sub_401000 (аргументу команды call) и нажать на клавишу **<Enter>**; если все сделано правильно, то экран дизассемблера должен выглядеть следующим образом:

```
00401000 ; ----- S U B R O U T I N E -----
00401000
00401000 ; Attributes: bp-based frame
00401000
00401000 sub_401000 proc near          ; CODE XREF: start+AF↓p
00401000         push ebp
00401001         mov ebp, esp
00401003         push offset aHelloSailor ; "Hello, Sailor!\n"
00401008         mov ecx, offset dword_408748
0040100D         call ??6ostream@@QAEAAV0@PBD@Z ; ostream::operator<<(char const *)
00401012         pop ebp
00401013         retn
00401013 sub_401000 endp
```

Дизассемблер сумел распознать строковую переменную и дал ей осмысленное имя «aHelloSailor», а в комментарии, расположенном справа, для наглядности привел оригинальное содержимое «Hello, Sailor!\n». Если поместить курсор в границы имени «aHelloSailor» и нажать клавишу **<Enter>**, IDA автоматически перейдет к требуемой строке:

```
00408040 aHelloSailor    db 'Hello, Sailor!',0Ah,0 ; DATA XREF: sub_401000+3↑o
```

Выражение DATA XREF: sub_401000+3↑o называется перекрестной ссылкой и свидетельствует о том, что в третьей строке процедуры sub_401000 произошло обращение к текущему адресу по его смещению («o» от offset), а стрелка, направленная вверх, указывает на относительное расположение источника перекрестной ссылки.

Если в границы выражения sub_401000+3 подвести курсор и нажать на клавишу **<Enter>**, IDA Pro перейдет к следующей строке:

```
00401003         push    offset aHelloSailor ; "Hello, Sailor!\n"
```

Нажатие клавиши **<Ecs>** отменяет предыдущее перемещение, возвращая курсор в исходную позицию (аналогично команде back в web-браузере.) Смещение строки «Hello, Sailor!\n» передается процедуре ??6ostream@@QAEAAV0@PBD@Z, представляющей собой оператор «<<» языка C++. Странное имя объясняется ограничениями, наложенными на символы, допустимые в именах библиотечных функций. Поэтому компиляторы автоматически преобразуют (**замангляют**) такие имена в абракадабру, пригодную для работы с линкером, и многие начинающие программисты даже не догадываются об этой скрытой кухне.

Для облегчения анализа текста IDA Pro в комментариях отображает «правильные» имена, но существует возможность заставить ее везде показывать незамангленные имена. Для этого необходимо в меню Options выбрать пункт Demangled names и в появившемся окне диалога переместить радиокнопку на Names, после чего вызов оператора «<<» станет выглядеть так:

```
0040100D call ostream::operator<<(char const *)
```

```

00401000 ; ----- S U B R O U T I N E -----
00401000
00401000 ; Attributes: bp-based frame
00401000
00401000 main                proc near                ; CODE XREF: start+AF↓p
00401000                     push     ebp
00401001                     mov     ebp, esp
00401003                     push     offset aHelloSailor ; "Hello, Sailor!\n"
00401008                     mov     ecx, offset dword_408748
0040100D                     call    ostream::operator<<(char const *)
00401012                     pop     ebp
00401013                     retn
00401013 main                endp

```

```

:00401000 55                                push ebp
:00401001 8BEC                               mov ebp, esp

Possible StringData Ref from Data Obj ->"Hello, Sailor!"
|
:00401003 6840804000                        push 00408040
:00401008 B948874000                        mov ecx, 00408748
:0040100D E8AB000000                        call 004010BD
:00401012 5D                                pop ebp
:00401013 C3                                ret

```

[illegible]

```

7E5B:114 18 00          sbb          [bx+si],al
7E5B:116 D2 6F DC      shr          byte ptr [bx-24h],cl ; Shift w/zeros fill
7E5B:119 6E 67 AB 47 A5 2E db 6Eh, 67h,0ABh, 47h,0A5h, 2Eh
7E5B:11F 03 0A 0A 09 4A 35 db 03h, 0Ah, 0Ah, 09h, 4Ah, 35h
7E5B:125 07 0F 0A 09 14 47 db 07h, 0Fh, 0Ah, 09h, 14h, 47h
7E5B:12B 6B 6C 42 E8 00 00 db 6Bh, 6Ch, 42h, E8h, 00h, 00h
7E5B:131 59 5E BF 00 01 57 db 59h, 5Eh, BFh, 00h, 01h, 57h
7E5B:137 2B CE F3 A4 C3   db 2Bh, CEh, F3h, A4h, C3h

Crypt                      endp

```

SOURCER половину кода вообще не смог дизассемблировать, оставив его в виде дампа, а другую половину дизассемблировал неправильно! Команда JMP SI в строке :0x103 осуществляет переход по адресу :0x106 (значение регистра SI после загрузки com-файла равно 0x100, поэтому после команды ADD SI,6 регистр SI равен 0x106). Но следующая за JMP команда расположена по адресу 0x105! В исходном тексте в это место вставлен байт-пустышка, сбивающий дизассемблер с толку.

Start:

```

ADD SI,6
JMP SI
DB 0B9h ;
LEA SI,_end ; На начало зашифрованного фрагмента

```

SOURCER не обладает способностью предсказывать регистровые переходы и, встретив команду JMP SI, продолжает дизассемблирование, молчаливо предполагая, что команды последовательно расположены вплотную друг к другу. Существует возможность создать файл определений, указывающий, что по адресу: 0x105 расположен байт данных, но подобное взаимодействие с пользователем очень неудобно.

Напротив, IDA изначально проектировалась как дружелюбная к пользователю интерактивная среда. В отличие от SURCER-подобных дизассемблеров, IDA не делает никаких молчаливых предположений, и при возникновении затруднений обращается за помощью к человеку. Поэтому, встретив регистровый переход по неизвестному адресу, она прекращает дальнейший анализ, и результат анализа файла Crypt.com выглядит так:

```

seg000:0100 start      proc near
seg000:0100             add     si, 6
seg000:0103             jmp     si
seg000:0103 start      endp
seg000:0103
seg000:0103 ; -----
seg000:0105             db 0B9h ; ш
seg000:0106             db 0BEh ; -
seg000:0107             db 14h ;
seg000:0108             db 1 ;
seg000:0109             db 0ADh ; i
seg000:010A             db 91h ; N
...

```

Необходимо помочь дизассемблеру, указав адрес перехода. Начинаящие пользователи в этой ситуации обычно подводят курсор к соответствующей строке и нажимают клавишу **<C>**, заставляя IDA дизассемблировать код с текущей позиции до конца функции. Несмотря на кажущуюся очевидность, такое решение ошибочно, ибо по-прежнему остается неизвестным, куда указывает условный переход в строке :0x103 и откуда код, расположенный по адресу :0x106, получает управление.

Правильное решение — добавить перекрестную ссылку, связывающую строку: 0x103 со строкой: 0x106. Для этого необходимо в меню **View** выбрать пункт **Cross references** и в появившемся окне диалога заполнить поля **from** и **to** значениями seg000:0103 и seg000:0106 соответственно.

После этого экран дизассемблера должен выглядеть следующим образом (в IDA версии 4.01.300 содержится ошибка, и добавление новой перекрестной ссылки не всегда приводит к автоматическому дизассемблированию):

```
seg000:0100                public start
seg000:0100 start          proc near
seg000:0100                add     si, 6
seg000:0103                jmp     si
seg000:0103 start          endp
seg000:0103
seg000:0103 ; -----
seg000:0105                db 0B9h ; ш
seg000:0106 ; -----
seg000:0106
seg000:0106 loc_0_106:      ; CODE XREF: start+3↑u
seg000:0106                mov     si, 114h
seg000:0109                lodsw
seg000:010A                xchg    ax, cx
seg000:010B                push    si
seg000:010C
seg000:010C loc_0_10C:      ; CODE XREF: seg000:0110(j
seg000:010C                xor     byte ptr [si], 66h
seg000:010F                inc     si
seg000:0110                loop    loc_0_10C
seg000:0112                jmp     si
seg000:0112 ; -----
seg000:0114                db 18h ;
seg000:0115                db 0 ;
seg000:0116                db 0D2h ; T
seg000:0117                db 6Fh ; o
...
```

Поскольку IDA Pro не отображает адреса-приемника перекрестной ссылки, то рекомендуется выполнить это самостоятельно. Такой прием улучшит наглядность текста и упростит навигацию. Если повести курсор к строке :0x103, нажать клавишу **<:>**, введя в появившемся диалоговом окне любой осмысленный комментарий (например, «переход по адресу 0106»), то экран примет следующий вид:

```
seg000:0103                jmp     si                ; Переход по адресу 0106
```

Ценность такого приема заключается в возможности быстрого перехода по адресу, на который ссылается `JMP SI`, достаточно лишь подвести курсор к числу `0106` и нажать клавишу **<Enter>**. Важно соблюдать правильность написания — IDA Pro не распознает шестнадцатеричный формат ни в стиле C (`0x106`), ни в стиле `MASM\TASM` (`0106h`).

Что представляет собой число `114h` в строке `:0x106` — константу или смещение? Чтобы узнать это, необходимо проанализировать команду `LODSW`, поскольку ее выполнение приводит к загрузке в регистр `AX` слова, расположенного по адресу `DS:SI`, очевидно, в регистр `SI` заносится смещение.

```
seg000:0106      mov     si, 114h
seg000:0109      lodsw
```

Однократное нажатие клавиши **<O>** преобразует константу в смещение, и дизассемблируемый текст станет выглядеть так:

```
seg000:0106      mov     si, offset unk_0_114
seg000:0109      lodsw
...
seg000:0114 unk_0_114 db  18h ;                ; DATA XREF: seg000:0106↑
seg000:0115      db      0 ;
seg000:0116      db  0D2h ; T
seg000:0117      db  6Fh ; o
...
```

IDA Pro автоматически создала новое имя `unk_0_114`, ссылающееся на переменную неопределенного типа размером в **байт**, но команда `LODSW` загружает в регистр `AX` **слово**, поэтому необходимо перейти к строке `:0144` и дважды нажать клавишу **<D>**. В этом случае экран станет выглядеть так:

```
seg000:0114 word_0_114 dw 18h                ; DATA XREF: seg000:0106↑
seg000:0116      db  0D2h ; T
```

Но что именно содержится в ячейке `word_0_144`? Понять это позволит изучение следующего кода:

```
seg000:0106      mov     si, offset word_0_114
seg000:0109      lodsw
seg000:010A      xchg    ax, cx
seg000:010B      push    si
seg000:010C
seg000:010C loc_0_10C:                ; CODE XREF: seg000:0110↓
seg000:010C      xor     byte ptr [si], 66h
seg000:010F      inc     si
seg000:0110      loop    loc_0_10C
```

В строке `:0x10A` значение регистра `AX` помещается в регистр `CX`, и затем он используется командой `LOOP LOC_010C` как счетчик цикла. Тело цикла представляет собой простейший расшифровщик — команда `XOR` расшифровывает один байт, на который указывает регистр `SI`, а команда `INC SI` перемещает указатель на следующий байт. Следовательно, в ячейке `word_0_144` содержится коли-

чество байтов, которые необходимо расшифровать. Подведя к ней курсор, нажатием клавиши <N> можно дать ей осмысленное имя, например BytesToDecrypt.

После завершения цикла расшифровщика встречается еще один безусловный регистровый переход:

```
seg000:0112          jmp     si
```

Чтобы узнать, куда именно он передает управление, необходимо проанализировать код и определить содержимое регистра SI. Для этой цели часто прибегают к помощи отладчика — устанавливают точку останова в строке 0x112 и, дождавшись его всплытия, просматривают значения регистров. Специально для этой цели IDA Pro поддерживает генерацию тар-файлов, содержащих символьную информацию для отладчика. В частности, чтобы не заучивать численные значения всех подопытных адресов, каждому из них можно присвоить легко запоминаемое символьное имя. Например, если подвести курсор к строке seg000:0112, нажать клавишу <N> и ввести команду BreakHere, отладчик сможет автоматически вычислить обратный адрес по его имени.

Для создания тар-файла в меню File необходимо щелкнуть мышью по команде **Produce output file** и в развернувшемся подменю выбрать команду **Produce MAP file** или вместо всего этого нажать на клавиатуре «горячую» комбинацию <Shift-F10>. Независимо от способа вызова на экране должно появиться диалоговое окно, позволяющее выбрать, какого рода данные будут включены в тар-файл — информация о сегментах, имена, автоматически сгенерированные IDA Pro (такие, как, например, loc_0_106, sub_0x110 и т. д.), и «размангленные» (т. е. приведенные в читабельный вид) имена. Содержимое полученного тар-файла должно быть следующим:

Start	Stop	Length	Name	Class
00100H	0013BH	0003CH	seg000	CODE

Address	Publics by Value
0000:0100	start
0000:0112	BreakHere
0000:0114	BytesToDecrypt

Program entry point at 0000:0100

Такой формат поддерживает большинство отладчиков, в том числе и популярнейший Soft-Ice, в поставку которого входит утилита msym, запускаемая с указанием имени конвертируемого тар-файла в командной строке. Полученный sym-файл необходимо разместить в одной директории с отлаживаемой программой, загружаемой в загрузчик **без указания расширения**, т. е., например, так: WLDR CRYPT. В противном случае символьная информация не будет загружена!

Затем необходимо установить точку останова командой brx BreakHere и покинуть отладчик командой x. Спустя секунду его окно вновь появится на экране, извещая о достижении процессором контрольной точки. Посмотрев на значения регистров, отображаемых по умолчанию вверху экрана, можно выяснить, что содержимое SI равно 0x12E.

С другой стороны, это же значение можно вычислить в уме, не прибегая к отладчику. Команда MOV в строке 0x106 загружает в регистр SI смещение 0x114, откуда командой LODSW считывается количество расшифровываемых байтов —

0x18, при этом содержимое SI увеличивается на размер слова — два байта. Отсюда в момент завершения цикла расшифровки значение SI будет равно $0x114 + 0x18 + 0x2 = 0x12E$.

Вычислив адрес перехода в строке 0x112, рекомендуется создать соответствующую перекрестную ссылку (from: 0x122; to: 0x12E) и добавить комментарий к строке 0x112 («Переход по адресу 012E»). Создание перекрестной ссылки автоматически дизассемблирует код, начиная с адреса seg000:012E и до конца файла.

```
seg000:012E loc_0_12E:                ; CODE XREF: seg000:0112u
seg000:012E      call $+3
seg000:0131      pop cx
seg000:0132      pop si
seg000:0133      mov di, 100h
seg000:0136      push di
seg000:0137      sub cx, si
seg000:0139      repe movsb
seg000:013B      retn
```

Назначение команды `CALL $ + 3` (где \$ обозначает текущее значение регистра указателя команд IP) состоит в заталкивании в стек содержимого регистра IP, откуда впоследствии оно может быть извлечено в любой регистр общего назначения. Необходимость подобного трюка объясняется тем, что в микропроцессорах серии Intel 80x86 регистр IP не входит в список непосредственно адресуемых и читать его значение могут лишь команды, изменяющие ход выполнения программы, в том числе и `call`.

Для облегчения анализа листинга можно добавить к строкам 0x12E и 0x131 комментарий `MOV CX, IP` или, еще лучше, сразу вычислить и подставить непосредственное значение `MOV CX, 0x131`.

Команда `POP SI` в строке 0x132 снимает слово из стека и помещает его в регистр SI. Прокручивая экран дизассемблера вверх, в строке 0x10B можно обнаружить парную ей инструкцию `PUSH SI`, заносящую в стек смещение первого расшифровываемого байта. После этого становится понятным смысл последующих команд `MOV DI, 0x100\SUB CX, SI\REPE MOVSB`. Они перемещают начало расшифрованного фрагмента по адресу, начинающегося со смещения 0x100. Такая операция характерна для конвертных защит, накладывающихся на уже откомпилированный файл, который перед запуском должен быть размещен по своим родным адресам.

Перед началом перемещения в регистр CX заносится длина копируемого блока, вычисляемая путем вычитания смещения первого расшифрованного байта от смещения второй команды перемещающего кода. В действительности истинная длина блока на три байта короче вычисленной, и по идее, от полученного значения необходимо вычесть три. Однако такое несогласование не нарушает работоспособности, поскольку содержимое ячеек памяти, лежащих за концом расшифрованного фрагмента, не определено и может быть любым.

Пара команд 0x136:PUSH DI и 0x13B:RETN образует аналог инструкции `CALL DI` — PUSH заталкивает адрес возврата в стек, а RETN извлекает его оттуда и передает управление по соответствующему адресу. Зная значение DI (оно

равно 0x100), можно было бы добавить еще одну перекрестную ссылку (from:0x13B; to:0x100) и комментарий к строке :0x13B — «Переход по адресу 0x100», но ведь к этому моменту по указанным адресам расположен совсем другой код! Поэтому логически правильнее добавить перекрестную ссылку from:0x13B; to:0x116 и комментарий «Переход по адресу 0x116».

Сразу же после создания новой перекрестной ссылки IDA попытается дизассемблировать зашифрованный код, в результате чего получится следующее:

```
seg000:0116 loc_0_116:                ; CODE XREF: seg000:013Bu
seg000:0116          shr             byte ptr [bx-24h], cl
seg000:0119          outsb
seg000:011A          stos            word ptr es:[edi]
seg000:011C          inc             di
seg000:011D          movsw
seg000:011E          add             cx, cs:[bp+si]
seg000:0121          or              cl, [bx+di]
seg000:0123          dec             dx
seg000:0124          xor             ax, 0F07h
seg000:0127          or              cl, [bx+di]
seg000:0129          adc             al, 47h
seg000:0129; -----
seg000:012B          db             6Bh ; k
seg000:012C          db             6Ch ; l
seg000:012D          db             42h ; B
seg000:012E; -----
```

Непосредственное дизассемблирование зашифрованного кода невозможно — предварительно его необходимо расшифровать. Подавляющее большинство дизассемблеров не могут модифицировать анализируемый текст на лету, и до загрузки в дизассемблер исследуемый файл должен быть полностью расшифрован. На практике, однако, это выглядит несколько иначе. Прежде чем расшифровывать, необходимо выяснить алгоритм расшифровки, проанализировав доступную часть файла. Затем нужно выйти из дизассемблера, тем или иным способом расшифровать «секретный» фрагмент, вновь загрузить файл в дизассемблер (причем предыдущие результаты дизассемблирования окажутся утерянными) и продолжить его анализ до тех пор, пока не встретится еще один зашифрованный фрагмент, после чего описанный цикл «выход из дизассемблера — расшифровка — загрузка — анализ» повторится вновь.

Достоинство IDA заключается в том, что она позволяет выполнить ту же задачу значительно меньшими усилиями, никуда не выходя из дизассемблера. Это достигается за счет наличия механизма виртуальной памяти. Если не вдаваться в технические тонкости, можно упрощенно изобразить IDA в виде прозрачной виртуальной машины, оперирующей с физической памятью компьютера. Для модификации ячеек памяти необходимо знать их адрес, состоящий из пары чисел — сегмента и смещения.

Слева каждой строки указывается ее смещение и имя сегмента, например seg000:0116. Узнать базовый адрес сегмента по его имени можно, открыв окно «Сегменты» и выбрав в меню View пункт Segments.

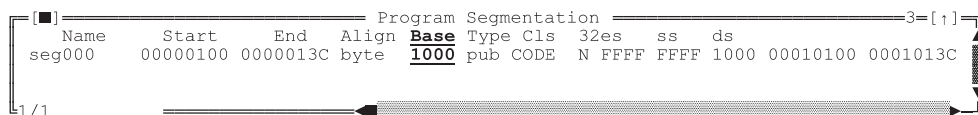


Рис. 8. Окно «Сегменты»

Искомый адрес находится в столбце Base; для наглядности на приведенной копии экрана он выделен жирным шрифтом. Обратиться к любой ячейке сегмента поможет конструкция [segment:offset], а для чтения и модификации ячеек предусмотрены функции Byte и PatchByte соответственно. Их вызов может выглядеть, например, так: `a=Byte([0x1000,0x100])` — читает ячейку, расположенную по смещению 0x100 в сегменте с базовым адресом 0x1000; `PatchByte([0x1000,0x100],0x27)` — присваивает значение 0x27 ячейке памяти, расположенной по смещению 0x100 в сегменте с базовым адресом 0x1000. Как следует из названия функций, они манипулируют с ячейками размером в один байт.

Знания этих двух функций вполне достаточно для написания скрипта-расшифровщика при условии, что читатель знаком с языком C.

Реализация IDA-C не полностью придерживается стандарта, в частности, IDA не позволяет разработчику задавать тип переменной и определяет его автоматически по ее первому использованию, а объявление осуществляется ключевым словом `auto`. Например, `auto MyVar, s0` объявляет две переменных — `MyVar` и `s0`.

Для создания скрипта необходимо нажать комбинацию клавиш **<Shift-F2>** или выбрать в меню File пункт IDC Command и в появившемся окне диалога ввести исходный текст программы:

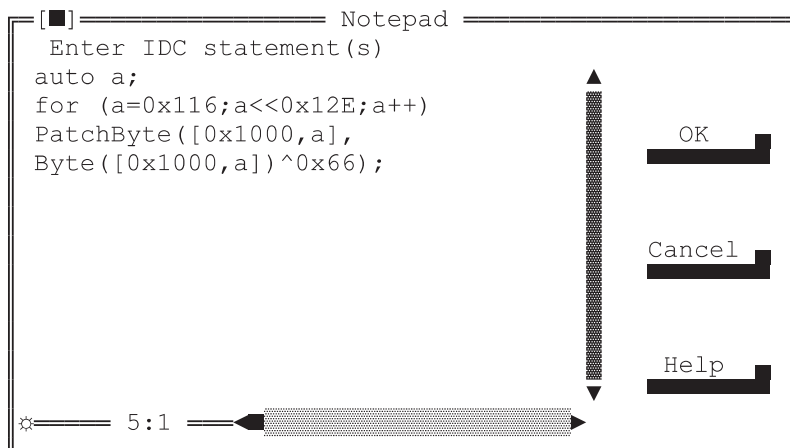


Рис. 9. Встроенный редактор скриптов

```
auto a;
for (a=0x116;a<0x12E;a++)
PatchByte([0x1000,a],Byte([0x1000,a])^0x66);
```

Пояснение:

Как было показано выше, алгоритм расшифровщика сводится к последовательному преобразованию каждой ячейки зашифрованного фрагмента операцией XOR 0x66 (см. ниже — выделено жирным шрифтом):

```
seg000:010C      xor     byte ptr [si], 66h
seg000:010F      inc     si
seg000:0110      loop    loc_0_10C
```

Сам же зашифрованный фрагмент начинается с адреса seg000:0x116 и продолжается вплоть до seg000:0x12E.

Отсюда цикл расшифровки на языке Си выглядит так:

```
for (a=0x116;a<0x12E;a++) PatchByte([0x1000,a],Byte([0x1000,a]^0x66));
```

В зависимости от версии IDA для выполнения скрипта необходимо нажать либо на клавишу **<Enter>** (версия 3.8х и старше), либо на комбинацию клавиш **<Ctrl-Enter>** в более ранних версиях. Если все сделано правильно, после выполнения скрипта экран дизассемблера должен выглядеть так:

```
seg000:0116 loc_0_116:                                ; CODE XREF: seg000:013Bu
seg000:0116      mov     ah, 9
seg000:0118      mov     dx, 108h
seg000:011B      int     21h                ; DOS - PRINT STRING
seg000:011B                                     ; DS:DX -> string terminated by "$"
seg000:011D      retn
seg000:011D ; -----
seg000:011E      db     48h ; H
seg000:011F      db     65h ; e
seg000:0120      db     6Ch ; l
seg000:0121      db     6Ch ; l
seg000:0122      db     6Fh ; o
seg000:0123      db     2Ch ; ,
seg000:0124      db     53h ; S
seg000:0125      db     61h ; a
seg000:0126      db     69h ; i
seg000:0127      db     6Ch ; l
seg000:0128      db     6Fh ; o
seg000:0129      db     72h ; r
seg000:012A      db     21h ; !
seg000:012B      db     0Dh ;
seg000:012C      db     0Ah ;
seg000:012D      db     24h ; $
seg000:012E ; -----
```

Возможные ошибки — несоблюдение регистра символов (IDA к этому чувствительна), синтаксические ошибки, базовый адрес вашего сегмента отличается от 0x1000 (еще раз вызовите окно «Сегменты», чтобы узнать его значение). В противном случае необходимо подвести курсор к строке seg000:0116, нажать клавишу **<U>** — для удаления результатов предыдущего дизассемблирования зашифрованного фрагмента и затем клавишу **<C>** — для повторного дизассемблирования расшифрованного кода.

Цепочку символов, расположенную начиная с адреса `seg000:011E`, можно преобразовать в удобочитаемый вид, подведя к ней курсор и нажав клавишу **<A>**. Теперь экран дизассемблера будет выглядеть так:

```
seg000:0116 loc_0_116:                                ; CODE XREF: seg000:013Bu
seg000:0116      mov     ah, 9
seg000:0118      mov     dx, 108h
seg000:011B      int     21h                        ; DOS - PRINT STRING
seg000:011B      ; DS:DX -> string terminated by "$"
seg000:011D      retn
seg000:011D      ; -----
seg000:011E      aHelloSailor db 'Hello,Sailor!',0Dh,0Ah,'$'
seg000:012E      ; -----
```

Команда `MOV AH,9` в строке `:0116` подготавливает регистр `AH` перед вызовом прерывания `0x21`, выбирая функцию вывода строки на экран, смещение которой заносится следующей командой в регистр `DX`. Иными словами, для успешного ассемблирования листинга необходимо заменить константу `0x108` соответствующим смещением. Но ведь выводимая строка на этапе ассемблирования (до перемещения кода) расположена совсем в другом месте! Одно из возможных решений этой проблемы заключается в создании нового сегмента с последующим копированием в него расшифрованного кода, в результате чего достигается эмуляция перемещения кода работающей программы.

Для создания нового сегмента можно выбрать в меню **View** пункт **Segments** и в раскрывшемся окне нажать клавишу **<Insert>**. Появится диалог следующего вида (рис. 10):

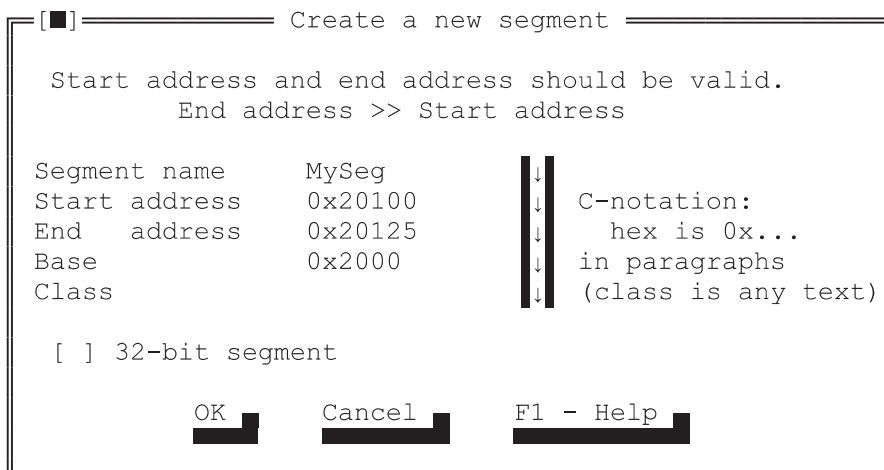


Рис. 10. IDAC: создание нового сегмента

Пояснение

Базовый адрес сегмента может быть любым, если при этом не происходят перекрытия сегментов `seg000` и `MySeg`; начальный адрес сегмента задается так, чтобы смещение первого байта было равно `0x100`; разница между конечным и

начальным адресом равна длине сегмента, вычислить которую можно вычитанием смещения начала расшифрованного фрагмента от смещения его конца — $0x13B - 0x116 = 0x25$.

Скопировать требуемый фрагмент в только что созданный сегмент можно скриптом следующего содержания:

```
auto a;
for (a=0x0; a<0x25; a++) PatchByte([0x2000, a+0x100], Byte([0x1000, a+0x116]));
```

Для его ввода необходимо вновь нажать комбинацию клавиш **<Shift-F2>**, при этом предыдущий скрипт будет утерян (IDA позволяет работать не более чем с одним скриптом одновременно). После завершения его работы экран дизассемблера будет выглядеть так:

```
MySeg:0100 MySeg      segment byte public `` use16
MySeg:0100            assume cs:MySeg
MySeg:0100            ;org 100h
MySeg:0100            assume es:nothing, ss:nothing, ds:nothing, fs:nothing, gs:nothing
MySeg:0100            db 0B4h ; 4
MySeg:0101            db 9 ;
MySeg:0102            db 0BAh ; B
MySeg:0103            db 8 ;
MySeg:0104            db 1 ;
MySeg:0105            db 0CDh ; D
MySeg:0106            db 21h ; 1
MySeg:0107            db 0C3h ; 3
MySeg:0108            db 48h ; H
MySeg:0109            db 65h ; e
MySeg:010A            db 6Ch ; l
MySeg:010B            db 6Ch ; l
MySeg:010C            db 6Fh ; o
MySeg:010D            db 2Ch ; ,
MySeg:010E            db 53h ; S
MySeg:010F            db 61h ; a
MySeg:0110            db 69h ; i
MySeg:0111            db 6Ch ; l
MySeg:0112            db 6Fh ; o
MySeg:0113            db 72h ; r
MySeg:0114            db 21h ; 1
MySeg:0115            db 0Dh ;
MySeg:0116            db 0Ah ;
MySeg:0117            db 24h ; $
MySeg:0117 MySeg      ends
```

Теперь необходимо создать перекрестную ссылку from:seg000:013B; to:MySeg:0x100, преобразовать цепочку символов в удобочитаемую строку, подведя курсор к строке MySeg:0108 и нажав клавишу **<A>**. Экран дизассемблера должен выглядеть так:

```
MySeg:0100 loc_1000_100:                                ; CODE XREF: seg000:013Bu
MySeg:0100            mov     ah, 9
MySeg:0102            mov     dx, 08h
```

```

MySeg:0105          int     21h          ; DOS - PRINT STRING
MySeg:0105          ; DS:DX -> string terminated by "$"
MySeg:0107          retn
MySeg:0107 ; -----
MySeg:0108 aHelloSailorS db 'Hello,Sailor!',0Dh,0Ah
MySeg:0108          db '$'
MySeg:0118 MySeg     ends

```

Результатом всех этих операций стало совпадение смещения строки со значением, загружаемым в регистр **DX** (в тексте они выделены жирным шрифтом). Если подвести курсор к константе 108h и нажать сочетание клавиш **<Ctrl-O>**, она будет преобразована в смещение:

```

MySeg:0102          mov dx, offset aHelloSailorS ; "Hello,Sailor!\r\n$w"
MySeg:0105          int 21h          ; DOS - PRINT STRING
MySeg:0105          ; DS:DX -> string terminated by "$"
MySeg:0107          retn
MySeg:0107 ; -----
MySeg:0108 aHelloSailorS db 'Hello,Sailor!',0Dh,0Ah ; DATA XREF: MySeg:0102o

```

Полученный листинг удобен для анализа, но все еще не готов к ассемблированию хотя бы потому, что никакой ассемблер не в состоянии зашифровать требуемый код. Конечно, эту операцию можно выполнить вручную после компиляции, но IDA позволит проделать то же самое, не выходя из нее и не прибегая к помощи стороннего инструментария.

Демонстрация получится намного нагляднее, если в исследуемый файл внести некоторые изменения, например, добавить ожидание клавиши на выходе. Для этого можно прибегнуть к интегрированному в IDA ассемблеру, но прежде, разумеется, необходимо несколько «раздвинуть» границы сегмента MySeg, дабы было к чему дописывать новый код.

Выберите в меню View пункт Segments и в открывшемся окне подведите курсор к строке MySeg. Нажатие комбинации клавиш **<Ctrl-E>** открывает диалог свойств сегмента, содержащий среди прочих полей конечный адрес, который и требуется изменить. Не обязательно указывать точное значение, можно «растянуть» сегмент с небольшим запасом от предполагаемых изменений.

Если попытаться добавить к программе код `XOR AX,AX; INT 16h`, он неминуемо затрет начало строки «Hello, Sailor!» — поэтому ее необходимо заблаговременно передвинуть немного «вниз» (т. е. в область более старых адресов), например, с помощью скрипта следующего содержания: `for(a=0x108;a<0x11A;a++) PatchByte([0x2000,a+0x20],Byte([0x2000,a]));`.

Пояснение

Объявление переменной «a» для краткости опущено (сами должны понимать, не маленькие :-), длина строки, как водится, берется с запасом, чтобы не утомлять себя лишними вычислениями, и перемещение происходит справа налево, поскольку исходный и целевой фрагменты заведомо не пересекаются.

Подведя курсор к строке :0128, нажатием **<A>** преобразуем цепочку символов к удобочитаемому виду; подведем курсор к строке :0102 и, выбрав в меню Edit

пункт Path program, Assembler, введем команду MOV DX,128h, где 128h — новое смещение строки, и тут же преобразуем его в смещение нажатием <Ctrl-O>.

Вот теперь можно вводить новый текст. Переместив курсор на инструкцию ret, вновь вызовем ассемблер и введем XOR AX,AX<ENTER>INT 16h<Enter>RET<Enter><Esc>. Напоследок рекомендуется произвести «косметическую» чистку — уменьшить размер сегмента до необходимого и переместить строку Hello, Sailor вверх, прижав ее вплотную к коду.

Пояснение

Удалить адреса, оставшиеся при уменьшении размеров сегмента за его концом, можно взводом флажка Disable Address в окне свойств сегмента, вызываемом нажатием сочетания клавиш <Alt-S>.

Если все было сделано правильно, то конечный результат должен выглядеть так, как показано ниже:

```
seg000:0100 ; File Name      : F:\IDAN\SRC\Crypt.com
seg000:0100 ; Format        : MS-DOS COM-file
seg000:0100 ; Base Address: 1000h Range: 10100h-1013Ch Loaded length: 3Ch
seg000:0100
seg000:0100
seg000:0100 ; =====
seg000:0100
seg000:0100 ; Segment type: Pure code
seg000:0100 seg000          segment byte public 'CODE' use16
seg000:0100                  assume cs:seg000
seg000:0100                  org 100h
seg000:0100                  assume es:nothing, ss:nothing, ds:seg000, fs:nothing, gs:nothing
seg000:0100
seg000:0100 ; ----- S U B R O U T I N E -----
seg000:0100
seg000:0100
seg000:0100          public start
seg000:0100 start          proc near
seg000:0100          add     si, 6
seg000:0103          jmp     si                ; переход по адресу 0106
seg000:0103 start          endp
seg000:0103
seg000:0103 ; -----
seg000:0105          db 0B9h ; ш
seg000:0106 ; -----
seg000:0106          mov     si, offset BytesToDecrypt
seg000:0109          lodsw
seg000:010A          xchg    ax, cx
seg000:010B          push    si
seg000:010C
seg000:010C loc_0_10C:          ; CODE XREF: seg000:0110j
seg000:010C          xor     byte ptr [si], 66h
seg000:010F          inc     si
seg000:0110          loop    loc_0_10C
seg000:0112
seg000:0112 BreakHere:          ; переход по адресу 012E
```

```

seg000:0112                jmp     si
seg000:0112 ; -----
seg000:0114 BytesToDecrypt dw     18h                ; DATA XREF: seg000:0106o
seg000:0116 ; -----
seg000:0116
seg000:0116 loc_0_116:                ; CODE XREF: seg000:013Bu
seg000:0116                mov     ah, 9
seg000:0118                mov     dx, 108h           ; "Hello,Sailor!\r\n$"
seg000:011B                int     21h               ; DOS - PRINT STRING
seg000:011B                ; DS:DX -> string terminated by "$"
seg000:011D                retn
seg000:011D ; -----
seg000:011E aHelloSailor    db     'Hello,Sailor!',0Dh,0Ah,'$' ; DATA XREF: seg000:0118o
seg000:012E ; -----
seg000:012E
seg000:012E loc_0_12E:                ; CODE XREF: seg000:0112u
seg000:012E                call    $+3
seg000:0131                pop     cx
seg000:0132                pop     si
seg000:0133                mov     di, 100h
seg000:0136                push    di
seg000:0137                sub     cx, si
seg000:0139                repe   movsb
seg000:013B                retn
seg000:013B seg000                ends
seg000:013B
MySeg:0100 ; -----
MySeg:0100 ; =====
MySeg:0100
MySeg:0100 ; Segment type: Regular
MySeg:0100 MySeg                segment byte public `` use16
MySeg:0100                assume cs:MySeg
MySeg:0100                ;org 100h
MySeg:0100                assume es:nothing, ss:nothing, ds:nothing, fs:nothing, gs:nothing
MySeg:0100
MySeg:0100 loc_1000_100:            ; CODE XREF: seg000:013Bu
MySeg:0100                mov     ah, 9
MySeg:0102                mov     dx, offset aHelloSailor_0 ; "Hello,Sailor!\r\n$"
MySeg:0105                int     21h               ; DOS - PRINT STRING
MySeg:0105                ; DS:DX -> string terminated by "$"
MySeg:0107                xor     ax, ax
MySeg:0109                int     16h ; KEYBOARD - READ CHAR FROM BUFFER, WAIT IF EMPTY
MySeg:0109                ; Return: AH = scan code, AL = character
MySeg:010B                retn
MySeg:010B ; -----
MySeg:010C aHelloSailor_0 db     'Hello,Sailor!',0Dh,0Ah,'$' ; DATA XREF: MySeg:0102o
MySeg:010C MySeg                ends
MySeg:010C
MySeg:010C
MySeg:010C end                start

```

Структурно программа состоит из следующих частей: расшифровщика, занимающего адреса seg000:0x100 — seg000:0x113, переменной размером в слово, со-

держащей количество расшифровываемых байтов, занимающей адреса `seg000:0x114` — `seg000:0x116`, исполняемого кода программы, занимающего целиком сегмент `MySeg`, и загрузчика, занимающего адреса `seg000:0x12E` — `seg000:0x13B`. Все эти части должны быть в перечисленном порядке скопированы в целевой файл, причем исполняемый код программы необходимо предварительно зашифровать, произведя над каждым его байтом операцию `XOR 0x66`.

Ниже приведен пример скрипта, автоматически выполняющего указанные действия. Для его загрузки достаточно нажать на клавишу **<F2>** или выбрать в меню File пункт Load file, IDC file.

```
// Компилятор для файла Crypt
//
static main()
{
    auto a, f;

    // Открывается файл Crtypt2.com для записи в двоичном режиме
    f=fopen("crypt2.com", "wb");

    // В файл Crypt2 копируется расшифровщик
    for (a=0x100; a<0x114; a++) fputc(Byte([0x1000, a]), f);
    // Определяется и копируется в файл слово, содержащее число
    // байтов для расшифровки
    fputc( SegEnd([0x2000, 0x100]) - SegStart([0x2000, 0x100]), f);
    fputc(0, f);

    // Копируется и на лету шифруется расшифрованный фрагмент
    for(a=SegStart([0x2000, 0x100]); a!=SegEnd([0x2000, 0x100]); a++)
        fputc(Byte(a) ^ 0x66, f);

    // Дописывается загрузчик
    for(a=0x12E; a<0x13C; a++)
        fputc(Byte([0x1000, a]), f);

    // Закрывается файл.
    fclose(f);
}
```

Выполнение скрипта приведет к созданию файла `Crypt2.com`, запустив который можно убедиться в его работоспособности — он выводит строку на экран и, дождавшись нажатия любой клавиши, завершает свою работу.

Огромным преимуществом такого подхода является сквозная компиляция файла, т. е. дизассемблированный листинг в действительности не ассемблировался! Вместо этого из виртуальной памяти байт за байтом читалось оригинальное содержимое, которое за исключением модифицированных строк доподлинно идентично исходному файлу. Напротив, повторное ассемблирование практически никогда не позволяет добиться полного сходства с дизассемблируемым файлом.

IDA — очень удобный инструмент для модификации файлов, исходные тексты которых утеряны или отсутствуют; она практически единственный дизассемблер, способный анализировать зашифрованные программы, не прибегая к сторонним средствам; она обладает развитым пользовательским интерфейсом и

удобной системой навигации по исследуемому тексту; она может справиться с любой мыслимой и немыслимой задачей...

...но эти и многие другие возможности невозможно реализовать в полной мере без владения языком скриптов, что и подтвердил приведенный выше пример.

Шаг шестой. Дизассемблер & отладчик в связке

Кот с улыбкой — и то редкость, но уж улыбка без кота — это я прямо не знаю, что такое.

Льюис Кэрролл. Алиса в стране чудес

Существует два способа исследования программ, распространяющихся без исходных текстов: дизассемблирование (статический анализ) и отладка (динамический анализ). Вообще-то любой отладчик обязательно включает в себя дизассемблер — иначе отлаживать программу пришлось бы непосредственно в машинных кодах!

Однако дизассемблер, включенный в отладчик, обычно слишком примитивен и не может похвастаться богатыми функциональными возможностями. Во всяком случае, дизассемблер, встроенный в популярнейший отладчик Soft-Ice, недалеко ушел от DUMPBIN, с недостатками которого мы уже имели честь столкнуться. Насколько же понятнее становится код, если его загрузить в IDA!

Чем же тогда ценен отладчик? Дело в том, что дизассемблер в силу своей статичности имеет ряд ограничений. Во-первых, исследователю приходится выполнять программу на «эмуляторе» процессора, «зашитом» в их собственной голове, следовательно, необходимо знать и назначение всех команд процессора, и всех структур операционной системы (включая недокументированные), и... Во-вторых, начать анализ с произвольного места программы не так-то просто — требуется знать содержимое регистров и ячеек памяти на данный момент, а как их узнать? С регистрами и локальными переменными еще бы куда ни шло — прокрутим экран дизассемблера вверх и посмотрим, какие значения им присваиваются. Но этот фокус не пройдет с глобальными переменными, модифицировать которые может кто угодно и когда угодно. Вот бы установить точку останова... но какая же в дизассемблере может быть точка останова? В-третьих, дизассемблирование вынуждает производить реконструкцию алгоритма каждой функции, в то время как отладка позволяет рассматривать ее как «черный ящик» со входом и выходом. Допустим, имеется у нас функция, которая расшифровывает основной модуль программы. В дизассемблере нам придется сначала разобраться в алгоритме шифрования (что может оказаться совсем не просто), затем переложить эту функцию на язык IDA-C, отладить ее, запустить расшифровщик... В отладчике же можно поручить выполнение этой функции процессору, не вникая в то, как она работает, и, дождавшись ее завершения, продолжить анализ уже расшифрованного модуля

программы. Можно перечислять бесконечно, но и без того ясно, что отладчик отнюдь не конкурент дизассемблеру, а напротив, партнер.

Опытные хакеры всегда используют эти инструменты в паре. Алгоритм реконструируется в дизассемблере, а все непонятные моменты оперативно уточняются прогоном программы под отладчиком. При этом возникает естественное желание видеть в отладчике все те символические имена, которые были внесены в дизассемблерный листинг.

И IDA Pro действительно позволяет это сделать! Выберем в меню File подменю Produce output file, а в нем пункт Produce MAP file (или нажмем «горячее» сочетание клавишу **<Shift-F10>**). На экране появится окно с запросом имени файла (введем, например, *simple.map*), а затем возникнет модальный диалог, уточняющий, какие именно имена стоит включать в тар-файл. Нажмем клавишу **<Enter>**, оставив все галочки в состоянии по умолчанию (подробно о назначении каждой из них можно прочесть в книге автора «Образ мышления — дизассемблер IDA»). Парой секунд спустя на диске образуется файл *simple.map*, содержащий всю необходимую отладочную информацию, представленную в тар-формате Borland. Отладчик Soft-ice не поддерживает такой формат, поэтому перед его использованием файл необходимо конвертировать в sym-формат специально на то предназначенной утилитой *idasym*, которую можно бесплатно скачать с сайта www.idapro.com или получить у дистрибьютора, продавшего вам IDA.

Набрав в командной строке *idasym simple.map* и, с удовлетворением отметив, что файл *simple.sym* действительно создан, загрузим исследуемое приложение *simple.exe* в отладчик любым возможным способом. Дождавшись появления Soft-Ice на экране, отдадим ему команду SYM для отображения содержимого таблицы символов. Если все сделано правильно, ответ Soft-Ice должен выглядеть приблизительно так (ниже приведен сокращенный вариант):

```
:sym
CODE(001B)
  001B:00401000 start
  001B:00401074 __GetExceptDLLInfo
  001B:0040107C _Main
  001B:00401104 _memchr
  001B:00401124 _memcpy
  001B:00401148 _memmove
  001B:00401194 _memset
  001B:004011C4 _strcmp
  001B:004011F0 _strlen
  001B:0040120C _memcmp
  001B:00401250 _strchr
  001B:00403C08 _printf
DATA(0023)
  0023:00407000 aBorlandCCopyri
  0023:004070D9 aEnterPassword
  0023:004070E9 aMygoodpassword
  0023:004070F9 aWrongPassword
  0023:00407109 aPasswordOk
  0023:00407210 aNotype
  0023:00407219 aBccxh1
```

Ура! Заработало! Теперь символьные имена не только отображаются на экране, упрощая понимание кода, — на любое из них можно быстро и с комфортом установить точку останова, скажем `bpn aMygoodpassword`, и отладчик поймет, что от него хотят! Нет больше нужды держать в памяти эти трудно, запоминаемые шестнадцатеричные адреса!

Шаг седьмой. Идентификация ключевых структур языков высокого уровня

Если твое оружие стоит только малой части энергии, затраченной твоим врагом, ты имеешь мощный рычаг, который может одолеть непреодолимые с виду трудности.

Ф. Херберт. Дом глав Дюны

Исследование алгоритма работы программ, написанных на языках высокого уровня, традиционно начинается с реконструкции ключевых структур исходного языка — *функций, локальных и глобальных переменных, ветвлений, циклов* и т. д. Это делает дизассемблерный листинг более наглядным и значительно упрощает его анализ.

Современные дизассемблеры достаточно интеллектуальны и львиную долю работы по распознаванию ключевых структур берут на себя. В частности, IDA Pro успешно справляется с идентификацией стандартных библиотечных функций, локальных переменных, адресуемых через регистр ESP, case-ветвлений и т. д. Однако порой она ошибается, вводя исследователя в заблуждение, к тому же ее высокая стоимость не всегда оправдывает применение. Так, например, студентам, изучающим ассемблер (а лучшее средство изучения ассемблера — дизассемблирование чужих программ), она едва ли по карману.

Разумеется, на IDA свет клином не сошелся, существуют же и другие дизассемблеры, скажем, тот же DUMPBIN, входящий в штатную поставку SDK. Почему бы на худой конец не воспользоваться им? Конечно, если под рукой нет ничего лучшего, сойдет и DUMPBIN, но в этом случае об интеллектуальности дизассемблера придется забыть и все делать своей головой.

Первым делом мы познакомимся не с оптимизирующими компиляторами — анализ их кода относительно прост и вполне доступен для понимания даже новичкам в программировании. Затем же, освоившись с дизассемблером, перейдем к вещам более сложным — оптимизирующим компиляторам, генерирующим очень хитрый, запутанный и витиеватый код.

Идентификация функций

Для некоторых людей программирование является такой же внутренней потребностью, подобно тому, как коровы дают молоко, или писатели стремятся писать.

Н. Безруков

Функция (также называемая процедурой или подпрограммой) — основная структурная единица процедурных и объективно-ориентированных языков, поэтому дизассемблирование кода обычно начинается с отождествления функций и идентификации передаваемых им аргументов.

Строго говоря, термин «функция» присутствует не во всех языках, но даже там, где он присутствует, его определение варьируется от языка к языку. Не вдаваясь в детали, мы будем понимать под функцией обособленную последовательность команд, вызываемую из различных частей программы. Функция может принимать один и более аргументов, а может не принимать ни одного; может возвращать результат своей работы, а может и не возвращать — это уже не суть важно. Ключевое свойство функции — возвращение управления на место ее вызова, а ее характерный признак — множественный вызов из различных частей программы (хотя некоторые функции вызываются лишь из одного места).

Откуда функция знает, куда следует возвратить управление? Очевидно, вызывающий код должен предварительно сохранить адрес возврата и вместе с прочими аргументами передать его вызываемой функции. Существует множество способов решения этой проблемы: можно, например, перед вызовом функции поместить в ее конец безусловный переход на адрес возврата, можно сохранить адрес возврата в специальной переменной и после завершения функции выполнить косвенный переход, используя эту переменную как операнд инструкции `jump...` Не останавливаясь на обсуждении сильных и слабых сторон каждого метода, отметим, что компиляторы в подавляющем большинстве случаев используют специальные машинные команды `ALL` и `RET`, соответственно предназначенные для вызова функций и возврата из них.

Инструкция `CALL` закидывает адрес следующей за ней инструкции на вершину стека, а `RET` стягивает и передает на него управление. Тот адрес, на который указывает инструкция `CALL`, и есть адрес начала функции. А замыкает функцию инструкция `RET` (но внимание: не всякий `RET` обозначает конец функции! Подробнее об этом см. раздел «Идентификация значения, возвращенного функцией»).

Таким образом, распознать функцию можно двояко: по **перекрестным ссылкам**, ведущим к машинной инструкции `CALL`, и по ее **эпилогу**, завершающемуся инструкцией `RET`. Перекрестные ссылки и эпилог в совокупности позволяют определить адреса начала и конца функции. Немного забегаая вперед (см. раздел «Идентификация локальных стековых переменных»), заметим, что в начале многих функций присутствует характерная последовательность команд, называемая **прологом**, которая также пригодна и для идентификации функций. А теперь расскажем обо всем этом поподробнее.

Перекрестные ссылки. Просматривая дизассемблерный код, находим все инструкции CALL — содержимое их операнда и будет искомым адресом начала функции. Адрес неvirtуальных функций, вызываемых по имени, вычисляется еще на стадии компиляции, и операнд инструкции CALL в таких случаях представляет собой непосредственное значение. Благодаря этому адрес начала функции выявляется простым синтаксическим анализом: ищем контекстным поиском все подстроки CALL и запоминаем (записываем) непосредственные операнды.

Рассмотрим следующий пример:

Листинг 6. Пример, демонстрирующий непосредственный вызов функции

```
func();

main(){
    int a;
    func();
    a=0x666;
    func();
}

func(){
    int a;
    a++;
}
```

Результат его компиляции должен выглядеть приблизительно так:

Листинг 7

```
.text:00401000 push    ebp
.text:00401001 mov     ebp, esp
.text:00401003 push    ecx
.text:00401004 call    401019
.text:00401004      ; Вот мы выловили инструкцию call с непосредственным операндом,
.text:00401004      ; представляющим собой адрес начала функции. Точнее - ее смещение
.text:00401004      ; в кодовом сегменте (в данном случае в сегменте .text)
.text:00401004      ; Теперь можно перейти к строке ".text:00401019" и, дав функции
.text:00401004      ; собственное имя, заменить операнд инструкции call на конструкцию
.text:00401004      ; "call offset Имя функции"
.text:00401004      ;
.text:00401009 mov     dword ptr [ebp-4], 666h
.text:00401010 call    401019
.text:00401010      ; А вот еще один вызов функции! Обратившись к строке .text:401019,
.text:00401010      ; мы увидим, что эта совокупность инструкций уже определена как функция
.text:00401010      ; и все, что потребуется сделать, - заменить call 401019 на
.text:00401010      ; "call offset Имя функции"
.text:00401010
.text:00401015 mov     esp, ebp
.text:00401017 pop     ebp
.text:00401018 retn
.text:00401018      ; Вот нам встретилась инструкция возврата из функции, однако не факт,
.text:00401018      ; что это действительно конец функции, - ведь функция может иметь и
.text:00401018      ; и несколько точек выхода. Однако смотрите: следом за ret
```



```
.text:00401018 ; расположено начало функции "моя функция", отождествленное по
.text:00401018 ; операнду инструкции call.
.text:00401018 ; Поскольку функции не могут перекрываться, выходит, что данный ret -
.text:00401018 ; конец функции!
.text:00401018 ;
.text:00401019 push    ebp
.text:00401019 ; На эту строку ссылаются операнды нескольких инструкций call.
.text:00401019 ; Следовательно, это адрес начала функции.
.text:00401019 ; Каждая функция должна иметь собственное имя - как бы нам ее назвать?
.text:00401019 ; Назовем ее "моя функция" :-)
.text:00401019 ;
.text:0040101A mov     ebp, esp      ; <-
.text:0040101C push    ecx      ; <-
.text:0040101D mov     eax, [ebp-4] ; <-
.text:00401020 add     eax, 1      ; <- Это - тело "моей функции"
.text:00401023 mov     [ebp-4], eax ; <-
.text:00401026 mov     esp, ebp ; <-
.text:00401028 pop     ebp      ; <-
.text:00401029 retn
.text:00401029; Конец "моей функции"
```

Как мы видим, все очень просто. Однако задача заметно усложняется, если программист (или компилятор) использует косвенные вызовы функций, передавая их адрес в регистре и динамически вычисляя его (адрес, а не регистр!) на стадии выполнения программы. Именно так, в частности, реализована работа с виртуальными функциями (см. раздел «Идентификация виртуальных функций»), однако в любом случае компилятор должен каким-то образом сохранить адрес функции в коде, значит, его можно найти и вычислить! Еще проще загрузить исследуемое приложение в отладчик, установить на «подследственную» инструкцию CALL точку останова и, дождавшись всплытия отладчика, посмотреть, по какому адресу она передаст управление.

Рассмотрим следующий пример:

Листинг 8. Пример, демонстрирующий вызов функции по указателю

```
func();
main(){
    int (a*)();
    a=func;
    a();
}
```

Результат его компиляции должен в общем случае выглядеть так:

Листинг 9

```
.text:00401000 push    ebp
.text:00401001 mov     ebp, esp
.text:00401003 push    ecx
.text:00401004 mov     dword ptr [ebp-4], 401012
.text:0040100B call    dword ptr [ebp-4]
.text:0040100B ; Вот инструкция CALL, осуществляющая косвенный вызов функции
```

```
.text:0040100B ; по адресу, содержащемуся в ячейке [EBP-4].
.text:0040100B ; Как узнать, что же там содержится? Прокрутим экран дизассемблера
.text:0040100B ; немного вверх, пока не встретим строку mov dword ptr [ebp-4],401012
.text:0040100B ; Ага! Значит, управление передается по адресу .text: 401012, -
.text:0040100B ; это и есть адрес начала функции!
.text:0040100B ; Даем функции имя и заменяем mov dword ptr [ebp-4], 401012 на
.text:0040100B ; mov dword ptr [ebp-4], offset Имя функции
.text:0040100B ;
.text:0040100E mov     esp, ebp
.text:00401010 pop     ebp
.text:00401011 retn
```

В некоторых, достаточно немногочисленных, программах встречается и косвенный вызов функции с комплексным вычислением ее адреса. Рассмотрим следующий пример:

Листинг 10. Пример, демонстрирующий вызов функции по указателю с комплексным вычислением целевого адреса

```
func_1();
func_2();
func_3();

main()
{
    int x;
    int a[3]={func_1,func_2, func_3};
    int (*f)();

    for (x=0;x < 3;x++)
    {
        f=(int (*)()) a[x];
        f();
    }
}
```

Результат его дизассемблирования в общем случае должен выглядеть так:

```
.text:00401000 push    ebp
.text:00401001 mov     ebp, esp
.text:00401003 sub     esp, 14h
.text:00401006 mov     [ebp+0xC], offset sub_401046
.text:0040100D mov     [ebp+0x8], offset sub_401058
.text:00401014 mov     [ebp+0x4], offset sub_40106A
.text:0040101B mov     [ebp+0x14], 0
.text:00401022 jmp     short loc_40102D
.text:00401024 mov     eax, [ebp+0x14]
.text:00401027 add     eax, 1
.text:0040102A mov     [ebp+0x14], eax
.text:0040102D cmp     [ebp+0x14], 3
.text:00401031 jge     short loc_401042
.text:00401033 mov     ecx, [ebp+0x14]
.text:00401036 mov     edx, [ebp+ecx*4+0xC]
.text:0040103A mov     [ebp+0x10], edx
.text:0040103D call    [ebp+0x10]
```

```
.text:0040103D ; Так-с, косвенный вызов функции. А что у нас в [EBP+0x10]?
.text:0040103D ; Поднимаем глаза на строку вверх - в [EBP+0x10] у нас значение EDX.
.text:0040103D ; А чему равен сам EDX? Прокручиваем еще одну строку вверх - EDX равен
.text:0040103D ; содержимому ячейки [EBP+ECX*4+0xC]. Вот дела! Мало того, что нам надо
.text:0040103D ; узнать содержимое этой ячейки, так еще и предстоит вычислить ее адрес!
.text:0040103D ; Чему равен ECX? Содержимому [EBP+0x14]. А оно чему равно?
.text:0040103D ; "Сейчас выясним..." - бормочем мы себе под нос, прокручивая экран
.text:0040103D ; дизассемблера вверх. Ага, нашли, - в строке 0x40102A в него
.text:0040103D ; загружается содержимое EAX! Какая радость! И долго мы по коду так
.text:0040103D ; блуждать будем?!
.text:0040103D ; Конечно, можно, затратив неопределенное количество времени и усилий,
.text:0040103D ; реконструировать весь ключевой алгоритм целиком (тем более что мы
.text:0040103D ; практически подошли к концу анализа), но где гарантия, что при этом
.text:0040103D ; не будут допущены ошибки?
.text:0040103D ; Гораздо быстрее и надежнее загрузить исследуемую программу в
.text:0040103D ; отладчик, установить бряк на строку "text:0040103D" и,
.text:0040103D ; дождавшись всплытия отладчика, посмотреть, что у нас расположено
.text:0040103D ; в ячейке [EBP+0x10]. Отладчик будет всплывать трижды, причем каждый
.text:0040103D ; раз показывать новый адрес! Заметим, что определить этот факт в
.text:0040103D ; дизассемблере можно только после полной реконструкции алгоритма!
.text:0040103D ; Однако не стоит по поводу мощи отладчика питать излишних иллюзий!
.text:0040103D ; Программа может тысячу раз вызывать одну и ту же функцию, а на
.text:0040103D ; тысяче первый - вызвать совсем другую! Отладчик бессилен это
.text:0040103D ; определить. Ведь вызов такой функции может произойти в
.text:0040103D ; непредсказуемый момент, например, при определенном сочетании времени,
.text:0040103D ; данных, обрабатываемых программой, и текущей фазы Луны. Ну не будем же
.text:0040103D ; мы целую вечность гонять программу под отладчиком?
.text:0040103D ; Дизассемблер - дело другое. Полная реконструкция алгоритма позволит
.text:0040103D ; однозначно и гарантированно отследить все адреса косвенных вызовов.
.text:0040103D ; Вот потому дизассемблер и отладчик должны скакать в одной упряжке!
.text:0040103D ;
.text:00401040 jmp     short loc_401024
.text:00401042
.text:00401042 mov     esp, ebp
.text:00401044 pop     ebp
.text:00401045 retn
```

Самый тяжелый случай представляют «ручные» вызовы функции командой **JMP** с предварительной засылкой в стек адреса возврата. Вызов через **JMP** в общем случае выглядит так: **PUSH ret_addr/JMP func_addr**, где **ret_addr** и **func_addr** — непосредственные или косвенные адреса возврата и начала функции соответственно. (Кстати, заметим, что команды **PUSH** и **JMP** не всегда следуют одна за другой и порой бывают разделены другими командами.)

Возникает резонный вопрос: чем же так плох **CALL** и зачем прибегать к **JMP**? Дело в том, что функция, вызванная по **CALL**, после возврата управления материнской функции всегда передает управление команде, следующей за **CALL**. В ряде случаев (например, при структурной обработке исключений) возникает необходимость после возврата из функции продолжать выполнение не со следующей за **CALL** командой, а совсем с другой ветки программы. Тогда-то и приходится вручную заносить требуемый адрес возврата и вызывать дочернюю функцию через **JMP**.

Идентифицировать такие функции (особенно если они не имеют пролога — см. раздел «*Пролог*») очень сложно — контекстный поиск ничего не дает, поскольку команд JMP, использующихся для локальных переходов, в теле любой программы очень и очень много — попробуй-ка проанализируй их все! Если же этого не сделать, из поля зрения выпадут сразу две функции — вызываемая функция и функция, на которую передается управление после возврата. К сожалению, быстрых решений этой проблемы не существует, единственная зацепка — вызывающий JMP практически всегда выходит за границы функции, в теле которой он расположен. Определить же границы функции можно по эпилогу (см. раздел «*Эпилог*»).

Рассмотрим следующий пример:

Листинг 11. Пример, демонстрирующий «ручной» вызов функции инструкцией JMP

```
func();

main()
{
    __asm
    {
        LEA ESI, return_addr
        PUSH ESI
        JMP funct
    return_addr:
    }
}
```

Результат его компиляции в общем случае должен выглядеть так:

Листинг 12

```
.text:00401000 push    ebp
.text:00401001 mov     ebp, esp
.text:00401003 push    ebx
.text:00401004 push    esi
.text:00401005 push    edi
.text:00401006 lea     esi, [401012h]
.text:0040100C push    esi
.text:0040100D jmp     401017
.text:0040100D ; Смотрите, казалось бы, тривиальный условный переход, что в нем
.text:0040100D ; такого? Ан нет! Это не простой переход, это замаскированный
.text:0040100D ; вызов функции! Откуда это следует? А давайте перейдем по адресу
.text:0040100D ; 0x401017 и посмотрим.
.text:0040100D ; .text:00401017 push ebp
.text:0040100D ; .text:00401018 mov ebp, esp
.text:0040100D ; .text:0040101A pop ebp
.text:0040100D ; .text:0040101B retn
.text:0040100D ; ~~~~
.text:0040100D ; Как вы думаете, куда этот ret возвращает управление? Естественно,
.text:0040100D ; по адресу, лежащему на верхушке стека. А что у нас лежит на стеке?
.text:0040100D ; PUSH EBP из строки 401017 обратно выталкивается инструкцией POP
.text:0040100D ; из строки 40101B, так... возвращаемся назад, к месту безусловного
.text:0040100D ; перехода, и начинаем медленно прокручивать экран дизассемблера вверх,
```

```
.text:0040100D ; отслеживая все обращения к стеку. Ага, попалась птичка! Инструкция
.text:0040100D ; PUSH ESI из строки 401000C закидывает на вершину стека содержимое
.text:0040100D ; регистра ESI, а он сам, в свою очередь, строкой выше принимает
.text:0040100D ; “на грудь” значение 0x401012 – это и есть адрес начала функции,
.text:0040100D ; вызываемой командой JMP (вернее, не адрес, а смещение, но это не
.text:0040100D ; принципиально важно).
.text:0040100D ;
.text:00401012 pop     edi
.text:00401013 pop     esi
.text:00401014 pop     ebx
.text:00401015 pop     ebp
.text:00401016 ret     0
```

Автоматическая идентификация функций посредством IDA Pro. Дизассемблер IDA Pro способен анализировать операнды инструкций CALL, что позволяет ему автоматически разбивать программу на функции. Причем IDA вполне успешно справляется с большинством косвенных вызовов! С комплексными вызовами и «ручными» вызовами функций командой JMP она, правда, совладать пока не в состоянии, но это не повод для огорчения — ведь подобные конструкции крайне редки и составляют менее процента от «нормальных» вызовов функций, тех, которые IDA без труда распознает!

Пролог. Большинство неоптимизирующих компиляторов помещают в начало функции следующий код, называемый *прологом*.

Листинг 13. Обобщенный код пролога функции

```
push ebp
mov  ebp, esp
sub  esp, xx
```

В общих чертах назначение пролога сводится к следующему: если регистр EBP используется для адресации локальных переменных (как часто и бывает), то перед его использованием он должен быть сохранен в стеке (иначе вызываемая функция «сорвет крышу» материнской), затем в EBP копируется текущее значение регистра указателя вершины стека (ESP) — происходит так называемое *открытие кадра стека*, и значение ESP уменьшается на размер области памяти, выделенной под локальные переменные.

Последовательность PUSH EBP/MOV EBP,ESP/SUB ESP,xx может служить хорошей сигнатурой для нахождения всех функций в исследуемом файле, включая и те, на которые нет прямых ссылок. Такой прием, в частности, использует в своей работе IDA Pro, однако оптимизирующие компиляторы умеют адресовать локальные переменные через регистр ESP и используют EBP как и любой другой регистр общего назначения. Пролог оптимизированных функций состоит из одной лишь команды SUB ESP, xxx — последовательность слишком короткая для использования ее в качестве сигнатуры функции, увы. Более подробный рассказ об эпилогах функций нас ждет впереди (см. *раздел «Идентификация локальных стековых переменных»*), поэтому во избежание никому не нужного дублирования не будем здесь на нем останавливаться.

Эпилог. В конце своей жизни функция закрывает кадр стека, перемещая указатель вершины стека «вниз», и восстанавливает прежнее значение EBP (если только оптимизирующий компилятор не адресовал локальные переменные через ESP, используя EBP как обычный регистр общего назначения). Эпилог функции может выглядеть двояко: либо ESP увеличивается на нужное значение командой ADD, либо в него копируется значение EBP, указывающее на низ кадра стека:

Листинг 14. Обобщенный код эпилога функции

pop	ebp	mov	esp, ebp
add	esp, 64h	pop	ebp
retn		retn	
Эпилог 1		Эпилог 2	

Важно отметить: между командами POP EBP/ADD ESP, xxx и MOV ESP, EBP/POP EBP могут находиться и другие команды — они необязательно должны следовать вплотную друг к другу. Поэтому для поиска эпилогов контекстный поиск непригоден — требуется применять поиск *по маске*.

Если функция написана с учетом соглашения PASCAL, то ей приходится самостоятельно очищать стек от аргументов. В подавляющем большинстве случаев это осуществляется инструкцией RET n, где n — количество байтов, снимаемых из стека после возврата. Функции же, соблюдающие C-соглашение, предоставляют очистку стека вызывающему их коду и всегда оканчиваются командой RET. API-функции Windows представляют собой комбинацию соглашений C и PASCAL — аргументы заносятся в стек справа налево, но очищает стек сама функция (подробнее обо всем этом см. *раздел «Идентификация аргументов функций»*).

Таким образом, RET может служить достаточным признаком эпилога функции, но не всякий эпилог — это конец. Если функция имеет в своем теле несколько операторов return (как часто и бывает), компилятор в общем случае генерирует для каждого из них свой собственный эпилог. Посмотрите, находится ли за концом эпилога новый пролог или продолжается код старой функции? Не забывайте и о том, что компиляторы обычно не помещают в исполняемый файл код, никогда не получающий управления. Иначе говоря, у функции будет всего один эпилог, а все находящееся после первого return будет выброшено как ненужное:

Листинг 15. Пример, демонстрирующий выбрасывание компилятором кода, расположенного за безусловным оператором return

int func(int a)	push ebp
{	mov ebp, esp
	mov eax, [ebp+arg_0]
return a++;	mov ecx, [ebp+arg_0]
a=1/a;	add ecx, 1
return a;	mov [ebp+arg_0], ecx
	pop ebp
}	retn

Напротив, если внеплановый выход из функции происходит при срабатывании некоторого условия, такой return будет сохранен компилятором и «окаймлен» условным переходом, прыгающим через эпилог.

Листинг 16. Пример, демонстрирующий функцию с несколькими эпилогами

```
int func(int a)
{
    if (!a) return a++;
    return 1/a;
}
```

Листинг 17

```
push    ebp
mov     ebp, esp
cmp     [ebp+arg_0], 0
jnz     short loc_0_401017
mov     eax, [ebp+arg_0]
mov     ecx, [ebp+arg_0]
add     ecx, 1
mov     [ebp+arg_0], ecx
pop     ebp
ret     0
```

; Да, это 0000000000000000 явно эпилог функции, но
 ; смотрите: следом идет продолжение кода функции, а
 ; вовсе не новый пролог!

loc_0_401017: ; CODE XREF: sub_0_401000+7↑j
 ; Данная перекрестная ссылка, приводящая нас к условному переходу,
 ; говорит о том, что этот код – продолжение прежней функции, а отнюдь не
 ; начало новой, ибо «нормальные» функции вызываются не `jump`, а `CALL`!
 ; А если это «ненормальная» функция? Что ж, это легко проверить – достаточно
 ; выяснить, лежит ли адрес возврата на вершине стека или нет? Смотрим,
 ; нет, не лежит, следовательно, наше предположение относительно продолжения
 ; кода функции верно.

```
mov     eax, 1
cdq
idiv    [ebp+arg_0]
```

loc_0_401020: ; CODE XREF: sub_0_401000+15↑j
 pop ebp
 ret 0

Специальное замечание: начиная с 80286 процессора, в наборе команд появились две инструкции `ENTER` и `LEAVE`, предназначенные специально для открытия и закрытия кадра стека. Однако они практически никогда не используются современными компиляторами. Почему? Причина в том, что `ENTER` и `LEAVE` очень медлительны, намного медлительнее `PUSH EBP/MOV EBP, ESP/SUB ESB, xxx` и `MOV ESP, EBP/POP EBP`. Так, на Pentium `ENTER` выполняется за десять тактов, а приведенная последовательность команд — за семь. Аналогично `LEAVE` требует пять тактов, хотя ту же операцию можно выполнить за два (и даже быстрее, если разделить `MOV ESP, EBP/POP EBP` какой-нибудь командой). Поэтому современный читатель никогда не столкнется ни с `ENTER`, ни с `LEAVE`. Хотя помнить об их назначении будет нелишне (мало ли, вдруг придется дизассемблировать древние программы

или программы, написанные на ассемблере, — не секрет, что многие пишущие на ассемблере очень плохо знают тонкости работы процессора и их «ручная оптимизация» заметно уступает компилятору по производительности).

«Голые» (naked) функции. Компилятор Microsoft Visual C++ поддерживает нестандартный квалификатор `naked`, позволяющий программистам создавать функции без пролога и эпилога. Без пролога и эпилога *вообще!* Компилятор даже не помещает в конце функции `RET`, и это приходится делать «вручную», прибегая к ассемблерной вставке `__asm{ret}` (Использование `return` не приводит к желаемому результату).

Вообще-то поддержка `naked`-функций задумывалась исключительно для написания драйверов на чистом C (ну, почти чистом, с небольшой примесью ассемблерных включений), но она нашла неожиданное признание и среди разработчиков защитных механизмов. Действительно, приятно иметь возможность «ручного» создания функций, не беспокоясь, что их непредсказуемым образом «изуродует» компилятор.

Для нас же, кодокопателей, в первом приближении это означает, что в программе может встретиться одна или несколько функций, не содержащих ни пролога, ни эпилога. Ну и что в этом страшного? Оптимизирующие компиляторы также выкидывают пролог, а от эпилога оставляют один лишь `RET`, но функции элементарно идентифицируются по вызывающей их инструкции `CALL`.

Идентификация встраиваемых (inline) функций. Самый эффективный способ избавиться от накладных расходов на вызов функций — не вызывать их. В самом деле, почему бы не встроить код функции непосредственно в саму вызывающую функцию? Конечно, это ощутимо увеличит размер (и тем ошутимее, чем из больших мест функция вызывается), но зато значительно увеличит скорость выполнения программы (и тем значительнее, чем чаще развернутая функция вызывается).

Чем плоха развертка функций для исследования программы? Прежде всего, она увеличивает размер материнской функции и делает ее код менее наглядным — вместо `CALL\TEST EAX, EAX\JZ xxx` с бросающимся в глаза условным переходом, мы видим кучу ничего не напоминающих инструкций, в логике работы которых еще предстоит разобраться!

Вспомним, мы уже сталкивались с таким приемом при анализе `sgasckme02`:

Листинг 18

```
mov     ebp, ds:SendMessageA
push    esi
push    edi
mov     edi, ecx
push    eax
push    666h
mov     ecx, [edi+80h]
push    0Dh
push    ecx
call    ebp ; SendMessageA
```



```

        lea     esi, [esp+678h+var_668]
        mov     eax, offset aMygoodpassword ; "MyGoodPassword"

loc_0_4013F0:                                ; CODE XREF: sub_0_4013C0+52j
        mov     dl, [eax]
        mov     bl, [esi]
        mov     cl, dl
        cmp     dl, bl
        jnz     short loc_0_401418
        test    cl, cl
        jz      short loc_0_401414
        mov     dl, [eax+1]
        mov     bl, [esi+1]
        mov     cl, dl
        cmp     dl, bl
        jnz     short loc_0_401418
        add     eax, 2
        add     esi, 2
        test    cl, cl
        jnz     short loc_0_4013F0

loc_0_401414:                                ; CODE XREF: sub_0_4013C0+3Cj
        xor     eax, eax
        jmp     short loc_0_40141D

loc_0_401418:                                ; CODE XREF: sub_0_4013C0+38j
        sbb     eax, eax
        sbb     eax, 0FFFFFFFh

loc_0_40141D:                                ; CODE XREF: sub_0_4013C0+56j
        test    eax, eax
        push    0
        push    0
        jz      short loc_0_401460

```

Встроенные функции не имеют ни собственного пролога, ни эпилога, их код и локальные переменные (если таковые имеются) полностью вживлены в вызывающую функцию, результат компиляции выглядит в точности так, как будто бы никакого вызова функции и не было. Единственная зацепка — встраивание функции неизбежно приводит к дублированию ее кода во всех местах вызова, а это хоть и с трудом, но можно обнаружить. С трудом потому, что встраиваемая функция, становясь частью вызывающей функции, в сквозную оптимизируется в контексте последней, что приводит к значительным вариациям кода. Рассмотрим такой пример:

Листинг 19. Пример, демонстрирующий, сквозную оптимизацию встраиваемых функций

```

#include <stdio.h>
__inline int max( int a, int b )
{
    if( a > b ) return a;
    return b;
}

```

```
int main(int argc, char **argv)
{
    printf("%x\n",max(0x666,0x777));
    printf("%x\n",max(0x666,argc));
    printf("%x\n",max(0x666,argc));
    return 0;
}
```

Результат его компиляции в общем случае должен выглядеть так:

Листинг 20

```
push esi
push edi
push 777h ; ← код 1-го вызова max
; Компилятор вычислил значение функции max еще на этапе компиляции и
; вставил его в программу, избавившись от лишнего вызова функции

push offset aProc ; "%x\n"
call printf
mov esi, [esp+8+arg_0]
add esp, 8

cmp esi, 666h ; ← код 2-го вызова max
mov edi, 666h ; ← код 2-го вызова max
jl short loc_0_401027 ; ← код 2-го вызова max
mov edi, esi ; ← код 2-го вызова max

loc_0_401027:                ; CODE XREF: sub_0_401000+23|j
push edi
push offset aProc ; "%x\n"
call printf
add esp, 8

cmp esi, 666h ; ← код 3-го вызова max
jge short loc_0_401042 ; ← код 2-го вызова max
mov esi, 666h ; ← код 2-го вызова max
; Смотрите, как изменился код функции! Во-первых, нарушилась очередность
; выполнения инструкций – было CMP -> MOV – Jx, а стало CMP -> Jx, MOV
; А во-вторых, условный переход JL загадочным образом превратился в JGE!
; Впрочем, ничего загадочного тут нет, просто идет сквозная оптимизация!
; Поскольку после 3-го вызова функции max переменная argc, размещенная
; компилятором в регистре ESI, более не используется, у компилятора появляется
; возможность непосредственно модифицировать этот регистр, а не вводить
; временную переменную, выделяя под нее регистр EDI
; (см. «Идентификация регистровых и временных переменных»)

loc_0_401042:                ; CODE XREF: sub_0_401000+3B|j
push esi
push offset aProc ; "%x\n"
call printf
add esp, 8
mov eax, edi
pop edi
pop esi
retn
```

Смотрите, при первом вызове компилятор вообще выкинул весь код функции, вычислив результат ее работы еще на стадии компиляции (действительно, `0x777` всегда больше `0x666`, и незачем тратить процессорные такты на их сравнение). А 2-й вызов очень мало похож на 3-й, несмотря на то что в обоих случаях функции передавали одни и те же аргументы! Тут не то что поиск по маске (не говоря уже о контекстном поиске), человек не разберется — вызывается одна и та же функция или нет!



Модели памяти и 16-разрядные компиляторы. Под адресом функции в данной главе до настоящего момента подразумевалось исключительно ее *смещение* в кодовом сегменте. Плоская (*flat*) модель памяти 32-разрядной Windows 9x\NT упаковывает все три сегмента — сегмент кода, сегмент стека и сегмент данных — в единое четырехгигабайтное адресное пространство, позволяя вообще забыть о существовании сегментов.

Иное дело 16-разрядные приложения для MS-DOS и Windows 3.x. В них максимально допустимый размер сегментов составляет всего лишь 64 килобайта, чего явно недостаточно для большинства приложений. В крошечной (*tiny*) модели памяти сегменты кода, стека и данных также расположены в одном адресном пространстве, но в отличие от плоской модели это адресное пространство чрезвычайно ограничено в размерах, и мало-мальски серьезное приложение приходится рассовывать по нескольким сегментам.

Теперь для вызова функции уже недостаточно знать ее смещение, требуется указать еще и сегмент, в котором она расположена. Однако сегодня об этом рудименте старины можно со спокойной совестью забыть. На фоне грядущей 64-разрядной версии Windows подробно описывать 16-разрядный код просто смешно!

Идентификация стартовых функций

...чтобы не наделать ошибок в работе, Богу понадобился свет. Судя по этому, в предшествовавшие века он сидел в полной темноте. К счастью, он не рисковал обо что-либо стукнуться, ибо вокруг ничего не было.

Л. Таксиль. Забавная Библия

Если первого встречного программиста спросить: «С какой функции начинается выполнение Windows-программы?» — вероятнее всего, мы услышим в ответ — «С *WinMain*», и это будет ошибкой. На самом же деле первым управление получает *стартовый код*, скрыто вставляемый компилятором, — выполнив необходимые инициализационные процедуры, в какой-то момент он вызывает *WinMain*, а после ее завершения вновь получает управление и выполняет капитальную деинициализацию.

В подавляющем большинстве случаев стартовый код не представляет никакого интереса, и первой задачей анализирующего становится поиск функции *WinMain*. Если компилятор входит в число «знакомых» IDA, она опознает *WinMain* авто-

матически, в противном же случае это приходится делать руками и головой. Обычно в штатную поставку компилятора входят исходные тексты его библиотек, в том числе и процедуры стартового кода. Например, у Microsoft Visual C++ стартовый код расположен в файлах CRT\STC\CRT0.C — версия для статичной компоновки, CRT\SRC\CRTEXE.C — версия для динамичной компоновки (т. е. библиотечный код не пристыкуется к файлу, а вызывается из DLL), CRT\SRC\wincmdln.c — версия для консольных приложений. У Borland C++ все файлы со start-up кодом хранятся в отдельной одноименной директории, в частности, стартовый код для Windows-приложений содержится в файле c0w.asm. Разобравшись с исходными текстами, понять дизассемблерный листинг будет намного легче!

А как быть, если для компиляции исследуемой программы использовался неизвестный или недоступный вам компилятор? Прежде чем приступить к утомительному ручному анализу, давайте вспомним, какой прототип имеет функция WinMain:

```
int WINAPI WinMain(
    HINSTANCE hInstance,      // handle to current instance
    HINSTANCE hPrevInstance,  // handle to previous instance
    LPSTR lpCmdLine,          // pointer to command line
    int nCmdShow              // show state of window
);
```

Во-первых, четыре аргумента (см. «Идентификация аргументов функций») — это достаточно много, и в большинстве случаев WinMain оказывается самой «богатой» на аргументы функцией стартового кода. Во-вторых, последний заносимый в стек аргумент — hInstance — чаще всего вычисляется на лету вызовом функции GetModuleHandleA, т. е., встретив конструкцию типа CALL GetModuleHandleA, можно с высокой степенью уверенности утверждать, что следующая функция и есть WinMain. Наконец, вызов WinMain обычно расположен практически в самом конце кода стартовой функции. За ней бывает не более двух-трех «закрывающих» строй функций, таких, как exit и XcptFilter.

Рассмотрим следующий фрагмент кода. Сразу бросается в глаза множество инструкций PUSH, заталкивающих в стек аргументы, последний из которых передает результат завершения GetModuleHandleA. Значит, перед нами не что иное, как вызов WinMain (и IDA подтверждает, что это именно так):

Листинг 21. Идентификация функции WinMain по роду и количеству передаваемых ей аргументов

```
.text:00401804 push    eax
.text:00401805 push    esi
.text:00401806 push    ebx
.text:00401807 push    ebx
.text:00401808 call    ds:GetModuleHandleA
.text:0040180E push    eax
.text:0040180F call    _WinMain@16
.text:00401814 mov     [ebp+var_68], eax
.text:00401817 push    eax
.text:00401818 call    ds:exit
```

Но не всегда все так просто, многие разработчики, пользуясь наличием исходных текстов startup-кода, модифицируют его (подчас весьма значительно). В результате выполнение программы может начинаться не с WinMain, а с любой другой функции, к тому же теперь стартовый код может содержать критические для понимания алгоритма операции (например, расшифровщик основного кода)! Поэтому *всегда* хотя бы мельком следует изучить startup-код: не содержит ли он чего-нибудь необычного?

Аналогичным образом обстоят дела и с динамическими библиотеками — их выполнение начинается вовсе не с функции DllMain (если она, конечно, вообще присутствует в DLL), а с `__DllMainCRTStartup` (по умолчанию). Впрочем, разработчики подчас изменяют умолчания, назначая ключом /ENTRY ту стартовую функцию, которая им нужна.

Строго говоря, неправильно называть DllMain *стартовой* функцией, она вызывается не только при загрузке DLL, но так же и при выгрузке, и при создании/уничтожении подключившим ее процессом нового потока. Получая уведомления об этих событиях, разработчик может предпринимать некоторые действия (например, подготавливать код к работе в многопоточной среде). Весьма актуален вопрос: имеет ли все это значение для анализа программы? Ведь чаще всего требуется проанализировать не всю динамическую библиотеку целиком, а исследовать работу некоторых экспортируемых ею функций. Если DllMain выполняет какие-то действия, скажем инициализирует переменные, то остальные функции, на которых распространяется влияние этих переменных, будут содержать на них прямые ссылки, ведущие прямоком к DllMain. Таким образом, не стоит «вручную» искать DllMain — она сама себя обнаружит! Хорошо, если бы *всегда* это было так! Но жизнь сложнее всяких правил. Вдруг в DllMain находится некий деструктивный код, или библиотека помимо основной своей деятельности шпионит за потоками, отслеживая их появление? Тогда без непосредственного анализа ее кода не обойтись!

Обнаружить DllMain на порядок труднее, чем WinMain; если ее не найдет IDA — пиши пропало. Во-первых, прототип DllMain достаточно незамысловат и не содержит ничего характерного:

```
BOOL WINAPI DllMain(
    HINSTANCE hinstDLL,    // handle to DLL module
    DWORD fdwReason,      // reason for calling function
    LPVOID lpvReserved    // reserved
);
```

А во-вторых, ее вызов идет из самой гущи довольно внушительной функции `__DllMainCRTStartup`, и можно легко убедиться, что это именно тот CALL, который нам нужен, — нет никакой возможности. Впрочем, некоторые зацепки все-таки есть. Так, при неудачной инициализации DllMain возвращает FALSE, и код `__DllMainCRTStartup` обязательно проверит это значение, в случае чего прыгая аж к концу функции.

Подробных ветвлений в теле стартовой функции не так уж много, и обычно только одно из них связано с функцией, принимающей три аргумента.

Листинг 22. Идентификация DllMain по коду неудачной инициализации

```
.text:1000121C      push     edi
.text:1000121D      push     esi
.text:1000121E      push     ebx
.text:1000121F      call     _DllMain@12
.text:10001224      cmp      esi, 1
.text:10001227      mov      [ebp+arg_4], eax
.text:1000122A      jnz      short loc_0_10001238
.text:1000122C      test     eax, eax
.text:1000122E      jnz      short loc_0_10001267
```

Прокрутив экран немного вверх, нетрудно убедиться, что регистры EDI, ESI и EBX содержат lpvReserved, fdwReason и hinstDLL соответственно. А значит, перед нами и есть функция DllMain. (Для справки: исходный текст __DllMainCRTStartup содержится в файле dllicrt0.c, который настоятельно рекомендуется изучить.)

Наконец мы добрались и до функции main консольных Windows-приложений. Как всегда, выполнение программы начинается не с нее, а с функции mainCRTStartup, инициализирующей *кучу*, систему ввода-вывода, подготавливающую аргументы командной строки и только потом предающей управление main. Функция main принимает всего два аргумента: *int main(int argc, char **argv)*, — этого слишком мало, чтобы выделить ее среди остальных. Однако приходит на помощь тот факт, что ключи командной строки доступны не только через аргументы, но и через глобальные переменные — __argc и __argv соответственно. Поэтому вызов main обычно выглядит так:

Листинг 23. Идентификация main

```
.text:00401293      push     dword_0_407D14
.text:00401299      push     dword_0_407D10
.text:0040129F      call     _main
.text:0040129F      ; Смотрите, оба аргумента функции - указатели на глобальные переменные
.text:0040129F      ; (см. "Идентификация глобальных переменных")
.text:0040129F
.text:004012A4      add      esp, 0Ch
.text:004012A7      mov      [ebp+var_1C], eax
.text:004012AA      push     eax
.text:004012AA      ; Смотрите, возвращаемое функцией значение передается функции exit
.text:004012AA      ; как код завершения процесса
.text:004012AA      ; Значит, это main и есть!
.text:004012AA
.text:004012AB      call     _exit
```

Обратите внимание и на то, что результат завершения main передается следующей за ней функции (это, как правило, библиотечная функция exit).

Вот мы и разобрались с идентификацией основных типов стартовых функций. Конечно, в жизни все бывает не так просто, как в теории, но в любом случае описанные выше приемы заметно упростят анализ.

Идентификация виртуальных функций

А мы летим орбитами, путями неизбитыми,
Прошит метеоритами простор.
Оправдан риск и мужество, космическая музыка
Вплывает в деловой наш разговор.

Трава у дома

Виртуальная функция по определению обозначает *«определяемая во время выполнения программы»*. При вызове виртуальной функции выполняемый код должен соответствовать динамическому типу объекта, из которого вызывается функция. Поэтому адрес виртуальной функции не может быть определен на стадии компиляции, это приходится делать непосредственно в момент ее вызова. Вот почему вызов виртуальной функции всегда *косвенный* вызов (исключение составляют лишь виртуальные функции статических объектов, см. раздел «Статическое связывание»).

В то время как неvirtуальные функции вызываются в точности так же, как и обычные Си-функции, вызов виртуальных функций происходит с кардинальными отличиями. Конкретная схема вызова зависит от реализации конкретного компилятора, но в общем случае ссылки на все виртуальные функции помещаются в специальный массив — *виртуальную таблицу (virtual table — сокращенно VTBL)*, а в каждый экземпляр объекта, использующий хотя бы одну виртуальную функцию, помещается *указатель на виртуальную таблицу (virtual table pointer — сокращенно VPRT)*. Причем независимо от числа виртуальных функций каждый объект имеет только один указатель.

Вызов виртуальных функций всегда происходит косвенно, через ссылку на виртуальную таблицу, например: CALL [EBX+0x10], где EBX — регистр, содержащий смещение виртуальной таблицы в памяти, а 0x10 — смещение указателя на виртуальную функцию внутри виртуальной таблицы.

Анализ вызова виртуальных функций наталкивается на ряд сложностей, самая коварная из которых — необходимость обратной трассировки кода для отслеживания значения регистра, используемого для косвенной адресации. Хорошо, если он инициализируется непосредственным значением типа MOV EBX, offset VTBL недалеко от места использования, но значительно чаще указатель на VTBL передается функции как неявный аргумент или (что еще хуже) один и тот же указатель используется для вызова двух различных виртуальных функций и тогда возникает неопределенность, какое именно значение (значения) он имеет в данной ветке программы.

Разберем следующий пример (предварительно вспомнив, что, если одна и та же неvirtуальная функция присутствует и в базовом, и в производном классе, всегда вызывается функция базового класса):

Листинг 24. Демонстрация вызова виртуальных функций

```
#include <stdio.h>

class Base{
public:
    virtual void demo(void)
    {
        printf("BASE\n");
    };

    virtual void demo_2(void)
    {
        printf("BASE DEMO 2\n");
    };

    void demo_3(void)
    {
        printf("Non virtual BASE DEMO 3\n");
    };
};

class Derived: public Base{
public:
    virtual void demo(void)
    {
        printf("DERIVED\n");
    };

    virtual void demo_2(void)
    {
        printf("DERIVED DEMO 2\n");
    };

    void demo_3(void)
    {
        printf("Non virtual DERIVED DEMO 3\n");
    };
};

main()
{
    Base *p = new Base;
    p->demo();
    p->demo_2();
    p->demo_3();

    p = new Derived;
    p->demo();
    p->demo_2();
    p->demo_3();
}
```

Результат ее компиляции в общем случае должен выглядеть так:

Листинг 25

```

main    proc near                ; CODE XREF: start+AFp
        push esi
        push 4
        call ??2@YAPAXI@Z      ; operator new(uint)
        ; EAX с - указатель на выделенный блок памяти.
        ; Выделяем четыре байта памяти для экземпляра нового объекта.
        ; Объект состоит из одного лишь указателя на VTBL.

        add esp, 4
        test eax, eax
        jz  short loc_0_401019 ; -> Ошибка выделения памяти,
        ; проверка успешности выделения памяти

        mov dword ptr [eax], offset BASE_VTBL
        ; Вот здесь в только что созданный экземпляр объекта копируется
        ; указатель на виртуальную таблицу класса BASE.
        ; То, что это именно виртуальная таблица класса BASE, можно узнать,
        ; проанализировав элементы этой таблицы, - они указывают на члены
        ; класса BASE, следовательно, сама таблица - виртуальная таблица
        ; класса BASE

        mov esi, eax            ; ESI = **BASE_VTBL
        ; заносим в ESI указатель на экземпляр объекта (указатель на указатель
        ; на BASE_VTBL
        ; Зачем? Дело в том, что на самом деле в ESI заносится указатель на
        ; экземпляр объекта (см. «Идентификация объектов, структур и массивов»),
        ; но нам на данном этапе все эти детали ни к чему, поэтому мы просто
        ; говорим, что в ESI - указатель на указатель на виртуальную таблицу
        ; базового класса, не вникая, для чего понадобился этот двойной указатель.

        jmp short loc_0_40101B

loc_0_401019:                    ; CODE XREF: sub_0_401000+Dj
        xor esi, esi
        ; принудительно обнуляем указатель на экземпляр объекта (эта ветка получает управление
        ; только в случае неудачного выделения памяти для объекта) нулевой указатель
        ; словит обработчик структурных исключений при первой же попытке обращения

loc_0_40101B:                    ; CODE XREF: sub_0_401000+17j
        mov eax, [esi]          ; EAX = *BASE_VTBL == *BASE_DEMO
        ; заносим в EAX указатель на виртуальную таблицу класса BASE,
        ; не забывая о том, что указатель на виртуальную таблицу одновременно
        ; является указателем и на первый элемент этой таблицы.
        ; А первый элемент виртуальной таблицы, содержащий указатель
        ; на первую (в порядке объявления) виртуальную функцию класса,

        mov ecx, esi            ; ECX = this
        ; заносим в ECX указатель на экземпляр объекта, передавая вызываемой функции
        ; неявный аргумент - указатель this (см. «Идентификация аргументов функций»)

        call dword ptr [eax]     ; CALL BASE_DEMO
        ; Вот он, вызов виртуальной функции! Чтобы понять, какая именно функция
        ; вызывается, мы должны знать значение регистра EAX. Прокручивая экран
        ; дизассемблера вверх, мы видим - EAX указывает на BASE_VTBL, а первый

```

```

; член BASE_VTBL (см. ниже) указывает на функцию BASE_DEMO. Следовательно:
; а) этот код вызывает именно функцию BASE_DEMO
; б) функция BASE_DEMO - это виртуальная функция

mov  edx, [esi]    ; EDX = *BASE_DEMO
; заносим в EDX указатель на первый элемент виртуальной таблицы класса BASE

mov  ecx, esi      ; ECX = this
; заносим в ECX указатель на экземпляр объекта
; Это неявный аргумент функции - указатель this (см. "Идентификация this")

call dword ptr [edx+4] ; CALL [BASE_VTBL+4] (BASE_DEMO_2)
; Еще один вызов виртуальной функции! Чтобы понять, какая именно функция
; вызывается, мы должны знать содержимое регистра EDX. Прокручивая экран
; дизассемблера вверх, мы видим, что он указывает на BASE_VTBL, а EDX+4,
; стало быть, указывает на второй элемент виртуальной таблицы класса BASE.
; Он же, в свою очередь, указывает на функцию BASE_DEMO_2

push offset aNonVirtualBase ; "Non virtual BASE DEMO 3\n"
call printf
; а вот вызов не виртуальной функции. Обратите внимание - он происходит,
; как и вызов обычной Си-функции. (Обратите внимание, что эта функция -
; встроенная, так как объявленная непосредственно в самом классе и вместо ее
; вызова осуществляется подстановка кода.)

push 4
call ??2@YAPAXI@Z ; operator new(uint)
; Далее идет вызов функций класса DERIVED. Не будем здесь подробно
; его комментировать, сделайте это самостоятельно. Вообще же, класс
; DERIVED понадобился только для того, чтобы показать особенности компоновки
; виртуальных таблиц.

add  esp, 8 ; Очистка после printf & new
test eax, eax
jz   short loc_0_40104A ; Ошибка выделения памяти
mov  dword ptr [eax], offset DERIVED_VTBL
mov  esi, eax ; ESI == **DERIVED_VTBL
jmp  short loc_0_40104C

loc_0_40104A: ; CODE XREF: sub_0_401000+3Ej
xor  esi, esi

loc_0_40104C: ; CODE XREF: sub_0_401000+48j
mov  eax, [esi] ; EAX = *DERIVED_VTBL
mov  ecx, esi ; ECX = this
call dword ptr [eax] ; CALL [DERIVED_VTBL] (DERIVED_DEMO)
mov  edx, [esi] ; EDX = *DERIVED_VTBL
mov  ecx, esi ; ECX=this
call dword ptr [edx+4] ; CALL [DERIVED_VTBL+4] (DERIVED_DEMO_2)

push offset aNonVirtualBase ; "Non virtual BASE DEMO 3\n"
call printf
; Обратите внимание - вызывается функция BASE_DEMO базового,
; а не производного класса!!!

add  esp, 4
pop  esi

```

```

        retn
main    endp

BASE_DEMO    proc near                ; DATA XREF: .rdata:004050B0o
        push    offset aBase          ; "BASE\n"
        call    printf
        pop     ecx
        retn
BASE_DEMO    endp

BASE_DEMO_2  proc near                ; DATA XREF: .rdata:004050B4o
        push    offset aBaseDemo2     ; "BASE DEMO 2\n"
        call    printf
        pop     ecx
        retn
BASE_DEMO_2  endp

DERIVED_DEMO proc near                ; DATA XREF: .rdata:004050A8o
        push    offset aDerived        ; "DERIVED\n"
        call    printf
        pop     ecx
        retn
DERIVED_DEMO endp

DERIVED_DEMO_2 proc near            ; DATA XREF: .rdata:004050ACo
        push    offset aDerivedDemo2   ; "DERIVED DEMO 2\n"
        call    printf
        pop     ecx
        retn
DERIVED_DEMO_2 endp

DERIVED_VTBL dd offset DERIVED_DEMO  ; DATA XREF: sub_0_401000+40o
              dd offset DERIVED_DEMO_2
BASE_VTBL    dd offset BASE_DEMO      ; DATA XREF: sub_0_401000+Fo
              dd offset BASE_DEMO_2

```

; Обратите внимание, виртуальные таблицы «растут» снизу вверх в порядке
 ; объявления классов в программе, а элементы виртуальных таблиц «растут»
 ; сверху вниз в порядке объявления виртуальных функций в классе.
 ; Конечно, так бывает не всегда (порядок размещения таблиц и их элементов
 ; нигде не декларирован и целиком лежит на «совести» компилятора, но на
 ; практике большинство из них ведут себя именно так). Сами же виртуальные
 ; функции располагаются вплотную друг к другу в порядке их объявления.

Идентификация чистой виртуальной функции. Если функция объявляется в базовом, а реализуется в производном классе, такая функция называется *чистой виртуальной функцией*, а класс, содержащий хотя бы одну такую функцию, называется *абстрактным классом*. Язык Си++ запрещает создание экземпляров абстрактного класса, да и как они могут создаваться, если, по крайней мере, одна из функций класса неопределенна?

На первый взгляд не определена, и ладно, какая в этом беда? Ведь на анализ программы это не влияет. На самом деле это не так — чистая виртуальная функция в виртуальной таблице замещается указателем на библиотечную функцию `__purecall`. Зачем она нужна? Дело в том, что на стадии компиляции программы

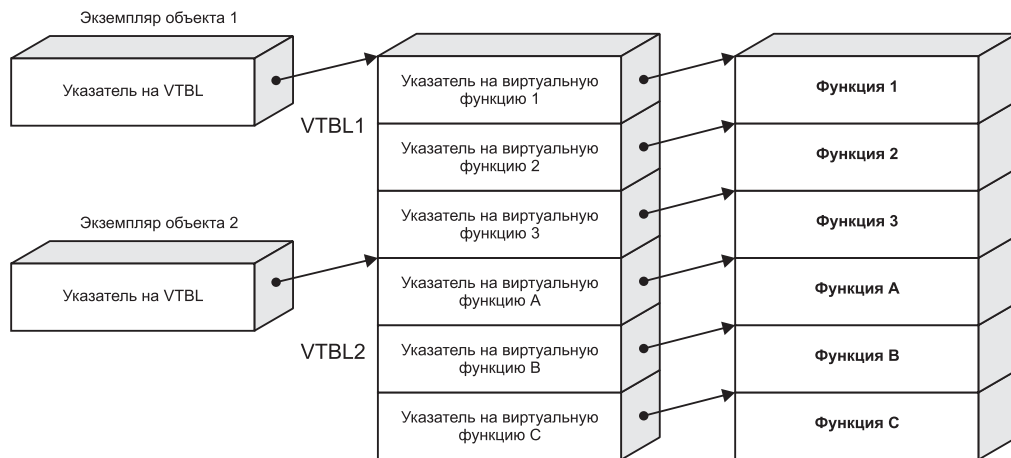


Рис. 11. Реализация вызова виртуальных функций

невозможно гарантированно отловить все попытки вызова чисто виртуальных функций, но если такой вызов и произойдет, управление получит заранее подставленная сюда `__purecall`, которая выведет на экран «ругательство» по поводу запрета на вызов чисто виртуальных функций и завершит работу приложения. Подробнее об этом можно прочитать в технической заметке MSDN Q120919, датированной 27 июня 1997 года.

Таким образом, встретив в виртуальной таблице указатель на `__purecall`, можно с уверенностью утверждать, что мы имеем дело с чисто виртуальной функцией. Рассмотрим следующий пример:

Листинг 26. Демонстрация вызова чистой виртуальной функции

```
#include <stdio.h>

class Base{
public:
    virtual void demo(void)=0;
};

class Derived:public Base {
public:
    virtual void demo(void)
    {
        printf("DERIVED\n");
    };
};

main()
{
    Base *p = new Derived;
    p->demo();
}
```

Результат его компиляции в общем случае должен выглядеть так:

Листинг 27

```

main    proc    near                ; CODE XREF: start+AF|p
        push    4
        call    ??2@YAPAXI@Z
        add     esp, 4
        ; Выделение памяти для нового экземпляра объекта

        test    eax, eax
        ; Проверка успешности выделения памяти

        jz      short loc_0_401017
        mov     ecx, eax
        ; ECX = this

        call    GetDERIVED_VTBL
        ; занесение в экземпляр объекта указателя на виртуальную таблицу класса
        ; DERIVED

        jmp     short loc_0_401019

loc_0_401017:                ; CODE XREF: main+C|j
        xor     eax, eax
        ; EAX = NULL

loc_0_401019:                ; CODE XREF: main+15|j
        mov     edx, [eax]
        ; тут возникает исключение по обращению к нулевому указателю

        mov     ecx, eax
        jmp     dword ptr [edx]
main    endp

GetDERIVED_VTBL proc near      ; CODE XREF: main+10p
        push    esi
        mov     esi, ecx
        ; Через регистр ECX-функции передается неявный аргумент - this

        call    SetPointToPure
        ; функция заносит в экземпляр объекта указатель на __purecall
        ; специальную функцию - заглушку на случай незапланированного вызова
        ; чисто виртуальной функции

        mov     dword ptr [esi], offset DERIVED_VTBL
        ; занесение в экземпляр объекта указателя на виртуальную таблицу производного
        ; класса, с затиранием предыдущего значения (указателя на __purecall)

        mov     eax, esi
        pop     esi
        retn

GetDERIVED_VTBL endp

DERIVED_DEMO proc near         ; DATA XREF: .rdata:004050A8|o
        push    offset aDerived ; "DERIVED\n"
        call    printf
        pop     ecx
        retn

DERIVED_DEMO endp

```

```

SetPointToPure proc near                ; CODE XREF: GetDERIVED_VTBL+3|p
    mov     eax, ecx
    mov     dword ptr [eax], offset PureFunc
    ; Заносим по [EAX] (в экземпляр нового объекта) указатель на специальную
    ; функцию - __purecall, которая предназначена для отслеживания попыток
    ; вызова чисто виртуальной функции в ходе выполнения программы -
    ; если такая попытка произойдет, __purecall выведет на экран матюгательство,
    ; дескать, вызывать чисто виртуальную функцию нельзя, и завершит работу

    retn
SetPointToPure endp

DERIVED_VTBL    dd offset DERIVED_DEMO    ; DATA XREF: GetDERIVED_VTBL+8|o
PureFunc        dd offset __purecall      ; DATA XREF: SetPointToPure+2|o
    ; указатель на функцию-заглушку __purecall. Следовательно, мы имеем дело
    ; с чисто виртуальной функцией.

```

Совместное использование виртуальной таблицы несколькими экземплярами объекта. Сколько бы экземпляров объекта ни существовало, все они пользуются одной и той же виртуальной таблицей. Виртуальная таблица принадлежит самому объекту, но не экземпляру (экземплярам) этого объекта. Впрочем, из этого правила существуют и исключения (см. раздел «Копии виртуальных таблиц»).

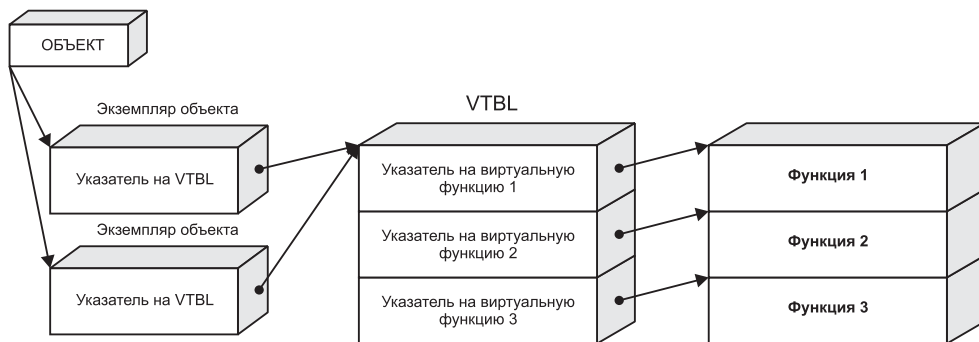


Рис. 12. Все экземпляры объекта используют одну и ту же виртуальную таблицу

Для подтверждения сказанного рассмотрим следующий пример:

Листинг 28. Демонстрация совместного использования одной копии виртуальной таблицы несколькими экземплярами класса

```

#include <stdio.h>

class Base{
public:
    virtual demo ()
    {
        printf("Base\n");
    }
};

class Derived:public Base{
public:

```

```

        virtual demo()
        {
            printf("Derived\n");
        }
};

main()
{
    Base * obj1 = new Derived;
    Base * obj2 = new Derived;

    obj1->demo();
    obj2->demo();
}

```

Результат его компиляции в общем случае должен выглядеть так:

Листинг 29

```

main          proc near                ; CODE XREF: start+AFp
    push     esi
    push     edi
    push     4
    call     ??2@YAPAXI@Z              ; operator new(uint)
    add      esp, 4
    ; Выделяем память под первый экземпляр объекта

    test     eax, eax
    jz       short loc_0_40101B
    mov      ecx, eax                  ; EAX указывает на первый экземпляр объекта

    call     GetDERIVED_VTBL
    ; в EAX - указатель на виртуальную таблицу класса DERIVED

    mov      edi, eax                  ; EDI = *DERIVED_VTBL
    jmp      short loc_0_40101D

loc_0_40101B:                ; CODE XREF: main+E|j
    xor      edi, edi

loc_0_40101D:                ; CODE XREF: main+19|j
    push     4
    call     ??2@YAPAXI@Z ; operator new(uint)
    add      esp, 4
    ; выделяем память под второй экземпляр объекта

    test     eax, eax
    jz       short loc_0_401043
    mov      ecx, eax                  ; ECX - this

    call     GetDERIVED_VTBL
    ; Обратите внимание, второй экземпляр использует ту же самую
    ; виртуальную таблицу

DERIVED_VTBL   dd offset DERIVED_DEMO ; DATA XREF: GetDERIVED_VTBL+80
BASE_VTBL      dd offset BASE_DEMO    ; DATA XREF: GetBASE_VTBL+20
; Обратите внимание, виртуальная таблица одна на все экземпляры класса

```

Копии виртуальных таблиц. ОК, для успешной работы, понятное дело, вполне достаточно и одной виртуальной таблицы, однако на практике приходится сталкиваться с тем, что исследуемый файл прямо-таки кишит копиями этих виртуальных таблиц. Что же это за напасть такая, откуда она берется и как с ней бороться?

Если программа состоит из нескольких файлов, компилируемых в самостоятельные obj-модули (а такой подход используется практически во всех мало-мальски серьезных проектах), компилятор, очевидно, должен поместить в каждый obj свою собственную виртуальную таблицу для каждого используемого модулем класса. В самом деле, откуда компилятору знать о существовании других obj и наличии в них виртуальных таблиц? Вот так и возникают никому не нужные дубли, отъедающие память и затрудняющие анализ. Правда, на этапе компоновки линкер может обнаружить копии и удалить их, да и сами компиляторы используют различные эвристические приемы для повышения эффективности генерируемого кода. Наибольшую популярность завоевал следующий алгоритм: виртуальная таблица помещается в тот модуль, в котором содержится реализация первой не-встроенной неvirtуальной функции класса. Обычно каждый класс реализуется в одном модуле, и в большинстве случаев такая эвристика срабатывает. Хуже, если класс состоит из одних виртуальных или встраиваемых функций, в этом случае компилятор «ложится» и начинает запихивать виртуальные таблицы во все модули, где этот класс используется. Последняя надежда на удаление «мусорных» копий возлагается на линкер, но и линкер не панацея. Собственно, эти проблемы должны больше заботить разработчиков программы (если их волнует количество занимаемой программой памяти), для анализа лишние копии всего лишь досадная помеха, но отнюдь не непреодолимое препятствие!

Связанный список. В большинстве случаев виртуальная таблица представляет собой обыкновенный массив, но некоторые компиляторы представляют ее в виде связанного списка, каждый элемент виртуальной таблицы содержит указатель на следующий элемент, а сами элементы размещены не вплотную друг к другу, а рассеяны по всему исполняемому файлу.

На практике подобное, однако, встречается крайне редко, поэтому не будем подробно на этом останавливаться, — достаточно лишь знать, что такое бывает, — если встретитесь со списками (впрочем, навряд ли вы с ними встретитесь) — разберетесь по обстоятельствам, благо это несложно.

Вызов через шлюз. Будьте также готовы и к тому, чтобы встретить в виртуальной таблице указатель не на виртуальную функцию, а на код, который модифицирует этот указатель, занося в него смещение вызываемой функции. Этот прием был предложен самим разработчиком языка Бьерном Страуструпом, позаимствовавшим его из ранних реализаций Алгола-60. В Алголе код, корректирующий указатель вызываемой функции, называется *шлюзом (thunk)*, а сам вызов — вызовом через шлюз. Вполне справедливо употреблять эти терминологию и по отношению к C++.

Однако в настоящее время вызов через шлюз чрезвычайно мало распространен и не используется практически ни одним компилятором. Несмотря на то, что

он обеспечивает более компактное хранение виртуальных таблиц, модификация указателя приводит к излишним накладным расходам на процессорах с конвейерной архитектурой (а Pentium — наиболее распространенный процессор — как раз и построен по такой архитектуре). Поэтому использование шлюзовых вызовов оправданно лишь в программах, критических к размеру, но не к скорости.

Подробнее обо всем этом можно прочесть в руководстве по Алголу-60 (шутка) или у Бьерна Страуструпа в *«Дизайне и эволюции языка C++»*.

Сложный пример, или Когда неvirtуальные функции попадают в виртуальные таблицы. До сих пор мы рассматривали лишь простейшие примеры использования виртуальных функций. В жизни же порой встречается такое... Рассмотрим сложный случай наследования с конфликтом имен:

Листинг 30. Демонстрация помещения не виртуальных функций в виртуальные таблицы

```
#include <stdio.h>

class A{
public:
    virtual void f() { printf("A_F\n");};
};

class B{
public:
    virtual void f() { printf("B_F\n");};
    virtual void g() { printf("B_G\n");};
};

class C:public A, public B {
public:
    void f(){ printf("C_F\n");}
}

main()
{
    A *a = new A;
    B *b = new B;
    C *c = new C;
    a->f();
    b->f();
    b->g();
    c->f();
}
```

Как будет выглядеть виртуальная таблица класса C? Так, давайте подумаем: раз класс C — производный от классов A и B, то он наследует функции обоих, но виртуальная функция f() класса B перекрывает одноименную виртуальную функцию класса A, поэтому из класса A она не наследуется. Далее, поскольку неvirtуальная функция f() присутствует и в производном классе C, она перекрывает виртуальную функцию производного класса (да, именно так, а вот неvirtуальная функция неvirtуальную не перекрывает, и она всегда вызывается из базового, а

не производного класса). Таким образом, виртуальная таблица класса C должна содержать только один элемент — указатель на виртуальную функцию g(), унаследованную от B, а не виртуальная функция f() вызывается как обычная Си-функция. Правильно? Нет!

Это как раз тот случай, когда *невиртуальная функция вызывается через указатель как виртуальная функция*. Более того, виртуальная таблица класса будет содержать не два, а три элемента! Третий элемент — это ссылка на виртуальную функцию f(), унаследованную от B, но тут же замещенная компилятором на «переходник» к C::f(). Уф... Как все непросто! Может, после изучения дизассемблерного листинга это станет понятнее?

Листинг 31

```
main          proc near          ; CODE XREF: start+AFp
    push      ebx
    push      esi
    push      edi
    push      4
    call      ??2@YAPAXI@Z      ; operator new(uint)
    add esp, 4
    ; Выделяем память для экземпляра объекта A

    test      eax, eax
    jz        short loc_0_40101C
    mov       ecx, eax          ; ECX = this
    call      Get_A_VTBL        ; a[0]=*A_VTBL
    ; Помещаем в экземпляр объекта указатель на его виртуальную таблицу

    mov       ebx, eax          ; EBX = *a
    jmp       short loc_0_40101E

loc_0_40101C:  ; CODE XREF: main+Fj
    xor       ebx, ebx

loc_0_40101E:  ; CODE XREF: main+1Aj
    push      4
    call      ??2@YAPAXI@Z      ; operator new(uint)
    add       esp, 4
    ; Выделяем память для экземпляра объекта B

    test      eax, eax
    jz        short loc_0_401037
    mov       ecx, eax          ; ECX = this
    call      Get_B_VTBL        ; b[0] = *B_VTBL
    ; Помещаем в экземпляр объекта указатель на его виртуальную таблицу

    mov       esi, eax ; ESI = *b
    jmp       short loc_0_401039

loc_0_401037:  ; CODE XREF: main+2Aj
    xor       esi, esi

loc_0_401039:  ; CODE XREF: main+35j
    push      8
    call      ??2@YAPAXI@Z      ; operator new(uint)
```

```

    add     esp, 4
    ; Выделяем память для экземпляра объекта B

    test    eax, eax
    jz      short loc_0_401052
    mov     ecx, eax          ; ECX = this
    call    GET_C_VTBLS      ; ret: EAX=*c
    ; Помещаем в экземпляр объекта указатель на его виртуальную таблицу
    ; (внимание: загляните в функцию GET_C_VTBLS)

    mov     edi, eax ; EDI = *c
    jmp     short loc_0_401054

loc_0_401052:                ; CODE XREF: main+45j
    xor     edi, edi

loc_0_401054:                ; CODE XREF: main+50j
    mov     eax, [ebx]       ; EAX = a[0] = *A_VTBL
    mov     ecx, ebx         ; ECX = *a
    call    dword ptr [eax]   ; CALL [A_VTBL] (A_F)
    mov     edx, [esi]       ; EDX = b[0]
    mov     ecx, esi         ; ECX = *b
    call    dword ptr [edx]   ; CALL [B_VTBL] (B_F)
    mov     eax, [esi]       ; EAX = b[0] = B_VTBL
    mov     ecx, esi         ; ECX = *b
    call    dword ptr [eax+4] ; CALL [B_VTBL+4] (B_G)
    mov     edx, [edi]       ; EDX = c[0] = C_VTBL
    mov     ecx, edi         ; ECX = *c
    call    dword ptr [edx]   ; CALL [C_VTBL] (C_F)
    ; Внимание! Вызов неvirtуальной функции происходит как виртуальной!

    pop     edi
    pop     esi
    pop     ebx
    retn

main                         endp

GET_C_VTBLS proc near        ; CODE XREF: main+49p
    push    esi              ; ESI = *b
    push    edi              ; ECX = *c
    mov     esi, ecx         ; ESI = *c
    call    Get_A_VTBL       ; c[0]=*A_VTBL
    ; помещаем в экземпляр объекта C указатель на виртуальную таблицу класса A

    lea     edi, [esi+4]     ; EDI = *c[4]
    mov     ecx, edi         ; ECX = **_C_F
    call    Get_B_VTBL       ; c[4]=*B_VTBL
    ; добавляем в экземпляр объекта C указатель на виртуальную таблицу класса B,
    ; т. е. теперь объект C содержит два указателя на две виртуальные таблицы
    ; базовых классов. Посмотрим далее, как компилятор справится с конфликтом
    ; имен...

    mov     dword ptr [edi], offset C_VTBL_FORM_B ; c[4]=*_C_VTBL
    ; Ага! Указатель на виртуальную таблицу класса B замещается указателем
    ; на виртуальную таблицу класса C (смотри комментарии в самой таблице)

```

```

    mov     dword ptr [esi], offset C_VTBL    ; c[0]=C_VTBL
    ; Ага, еще раз, теперь указатель на виртуальную таблицу класса А замещается
    ; указателем на виртуальную таблицу класса С. Какой неоптимальный код, ведь это
    ; можно было сократить еще на стадии компиляции!

    mov     eax, esi                        ; EAX = *c
    pop     edi
    pop     esi
    retn

GET_C_VTBLs    endp

Get_A_VTBL     proc near                  ; CODE XREF: main+13p GET_C_VTBLs+4p
    mov     eax, ecx
    mov     dword ptr [eax], offset A_VTBL
    ; помещаем в экземпляр объекта указатель на виртуальную таблицу класса В

    retn

Get_A_VTBL     endp

A_F            proc near                  ; DATA XREF: .rdata:004050A8o
    ; виртуальная функция f() класса А

    push    offset aA_f                   ; "A_F\n"
    call    printf
    pop     ecx
    retn

A_F            endp

Get_B_VTBL     proc near                  ; CODE XREF: main+2Ep GET_C_VTBLs+Ep
    mov     eax, ecx
    mov     dword ptr [eax], offset B_VTBL
    ; помещаем в экземпляр объекта указатель на виртуальную таблицу класса В

    retn

Get_B_VTBL     endp

B_F            proc near                  ; DATA XREF: .rdata:004050A0o
    ; виртуальная функция f() класса В

    push    offset aB_f                   ; "B_F\n"
    call    printf
    pop     ecx
    retn

B_F            endp

B_G            proc near                  ; DATA XREF: .rdata:004050B0o
    ; виртуальная функция g() класса В

    push    offset aB_g                   ; "B_G\n"
    call    printf
    pop     ecx
    retn

B_G            endp

C_F            proc near                  ; CODE XREF: _C_F+3j
    ; Не виртуальная функция f() класса С выглядит и вызывается как виртуальная!

    push    offset aC_f                   ; "C_F\n"
    call    printf

```

```

        pop     ecx
        retn
C_F      endp

_C_F     proc near                ; DATA XREF: .rdata:004050B8o
        sub     ecx, 4
        jmp     C_F
        ; Смотрите, какая странная функция! Во-первых, она никогда не вызывается, а
        ; во-вторых, это переходник к функции C_F.
        ; Зачем уменьшается ECX? В ECX компилятор поместил указатель this, который
        ; до уменьшения пытался указывать на виртуальную функцию f(), унаследованную
        ; от класса B. Но на самом же деле this указывал на этот переходник.
        ; А после уменьшения он стал указывать на предыдущий элемент виртуальной
        ; таблицы, т.е. функцию f() класса C, вызов которой и осуществляет JMP
_C_F     endp

A_VTBL   dd offset A_F           ; DATA XREF: Get_A_VTBL+2o
        ; виртуальная таблица класса A

B_VTBL   dd offset B_F           ; DATA XREF: Get_B_VTBL+2o
        dd offset B_G
        ; виртуальная таблица класса B содержит указатели на две виртуальные функции

C_VTBL   dd offset C_F           ; DATA XREF: GET_C_VTBLs+19o
        ; виртуальная таблица класса C - содержит указатель на не виртуальную функцию f()

C_VTBL_FORM_B dd offset _C_F     ; DATA XREF: GET_C_VTBLs+13o
        dd offset B_G
        ; виртуальная таблица класса C, скопированная компилятором из класса B. Первоначально
        ; состояла из двух указателей на функции f() и g(), но еще на стадии
        ; компиляции компилятор разобрался в конфликте имен и заменил указатель на B::f()
        ; указателем на переходник к C::f()

```

Таким образом, на самом деле виртуальная таблица производного класса включает в себя виртуальные таблицы всех базовых классов (во всяком случае, всех, откуда она наследует виртуальные функции). В данном случае виртуальная таблица класса C содержит указатель на неvirtуальную функцию C и виртуальную таблицу класса B. Задача: как определить, что функция C::f() неvirtуальная? И как найти все базовые классы класса C?

Начнем с последнего. Да, виртуальная таблица класса C не содержит никакого намека на его родственные отношения с классом A, но взгляните на содержимое функции GET_C_VTBLs, видите: предпринимается попытка внедрить в C указатель на виртуальную таблицу A, следовательно, класс C — производный от A. Автору могут возразить, дескать, это не слишком надежный путь, компилятор мог бы оптимизировать код, выкинув обращение к виртуальной таблице класса A, которое все равно не нужно. Это верно, *мог бы*, но на практике большинство компиляторов так не делают, а если и делают, все равно оставляют достаточно избыточной информации, позволяющей установить базовые классы. Другой вопрос: так ли необходимо устанавливать «родителей», от которых не наследуется ни одной функции? (Если хоть одна функция наследуется, никаких сложностей в поиске не возникает.) В общем-то для анализа это действительно не критично, но, чем

точнее будет восстановлен исходный код программы, тем нагляднее он будет и тем легче в нем разобраться.

Теперь перейдем к неvirtуальной функции `f()`. Подумаем, что было бы, будь она на самом деле виртуальной. Тогда она бы перекрыла одноименную функцию базовых классов и никакой «дикости» наподобие «переходников» в откомпилированной программе не встретилось бы. А так они говорят, что тут не все гладко и функция не виртуальная, хоть и стремится казаться такой. Опять-таки умный компилятор теоретически может выкинуть переходник и дублирующий элемент виртуальной таблицы класса `C`, но на практике этой интеллектуальности не наблюдается...

Статическое связывание. Есть ли разница, как создавать экземпляр объекта — `MyClass zzz;` или `MyClass *zzz=new MyClass?` Разумеется, в первом случае компилятор может определить адреса виртуальных функций еще на стадии компиляции, тогда как во втором это приходится вычислять в ходе выполнения программы. Другое различие: статические объекты размещаются в стеке (сегменте данных), а динамические — в куче. Таблица виртуальных функций упорно создается компиляторами в обоих случаях, а при вызове каждой функции (включая неvirtуальные) подготавливается указатель `this` (как правило, помещаемый в один из регистров общего назначения — подробнее см. раздел «Идентификация аргументов функций»), содержащий адрес экземпляра объекта.

Таким образом, если мы встречаем функцию, вызываемую непосредственно по ее смещению, но в то же время присутствующую в виртуальной таблице класса, можно с уверенностью утверждать, что это виртуальная функция статичного экземпляра объекта.

Рассмотрим следующий пример:

Листинг 32. Демонстрация вызова статической виртуальной функции

```
#include <stdio.h>

class Base{
public:

    virtual void demo(void)
    {
        printf("BASE DEMO\n");
    };

    virtual void demo_2(void)
    {
        printf("BASE DEMO 2\n");
    };

    void demo_3(void)
    {
        printf("Non virtual BASE DEMO 3\n");
    };
};
```

```

class Derived: public Base{
public:

    virtual void demo(void)
    {
        printf("DERIVED DEMO\n");
    };

    virtual void demo_2(void)
    {
        printf("DERIVED DEMO 2\n");
    };

    void demo_3(void)
    {
        printf("Non virtual DERIVED DEMO 3\n");
    };
};

main()
{
    Base p;
    p.demo();
    p.demo_2();
    p.demo_3();

    Derived d;
    d.demo();
    d.demo_2();
    d.demo_3();
}

```

Результат ее компиляции в общем случае должен выглядеть так:

Листинг 33

```

main          proc near          ; CODE XREF: start+AFp
var_8         = byte ptr -8      ; derived
var_4         = byte ptr -4      ; base
; часто (но не всегда!) экземпляры объектов в стеке расположены снизу вверх,
; т. е. в обратном порядке их объявления в программе

push    ebp
mov     ebp, esp
sub     esp, 8

lea     ecx, [ebp+var_4]         ; base
call    GetBASE_VTBL           ; p[0]=*BASE_VTBL
; Обратите внимание, экземпляр объекта размещается в стеке,
; а не в куче! Это, конечно, еще не свидетельствует о статичной
; природе экземпляра объекта (динамичные объекты тоже могут размещаться в стеке),
; но намеком на «статичку» все же служит

lea     ecx, [ebp+var_4]         ; base
; Подготавливаем указатель this (на тот случай, если он понадобится функции).

```

```

call    BASE_DEMO
; непосредственный вызов функции! Вот вкупе с ее наличием в виртуальной таблице
; свидетельство статичности объявления экземпляра объекта!

lea     ecx, [ebp+var_4]      ; base
; Вновь подготавливаем указатель this на экземпляр base

call    BASE_DEMO_2
; непосредственный вызов функции. Она есть в виртуальной таблице? Есть!
; Значит, это виртуальная функция, а экземпляр объекта объявлен статичным

lea     ecx, [ebp+var_4]      ; base
; готовим указатель this для невиртуальной функции demo_3

call    BASE_DEMO_3
; этой функции нет в виртуальной таблице (см. виртуальную таблицу),
; значит, она не виртуальная

lea     ecx, [ebp+var_8]      ; derived
call    GetDERIVED_VTBL      ; d[0]=*DERIVED_VTBL

lea     ecx, [ebp+var_8]      ; derived
call    DERIVED_DEMO
; аналогично предыдущему...

lea     ecx, [ebp+var_8]      ; derived
call    DERIVED_DEMO_2
; аналогично предыдущему...

lea     ecx, [ebp+var_8]      ; derived
call    BASE_DEMO_3_
; Внимание! Указатель this указывает на объект DERIVED, в то время как
; вызывается функция объекта BASE!!! Значит, функция BASE - производная

mov     esp, ebp
pop     ebp
retn

main endp

BASE_DEMO    proc near          ; CODE XREF: main+11p
; функция demo класса BASE

    push    offset aBase        ; "BASE\n"
    call    printf
    pop     ecx
    retn

BASE_DEMO    endp

BASE_DEMO_2  proc near          ; CODE XREF: main+19p
; функция demo_2-го класса BASE

    push    offset aBaseDemo2   ; "BASE DEMO 2\n"
    call    printf
    pop     ecx
    retn

BASE_DEMO_2  endp

```



```

BASE_DEMO_3    proc near                ; CODE XREF: main+21p
; функция demo_3-го класса BASE
        push    offset aNonVirtualBase ; "Non virtual BASE DEMO 3\n"
        call    printf
        pop     ecx
        retn
BASE_DEMO_3    endp

DERIVED_DEMO   proc near                ; CODE XREF: main+31p
; функция demo класса DERIVED
        push    offset aDerived        ; "DERIVED\n"
        call    printf
        pop     ecx
        retn
DERIVED_DEMO   endp

DERIVED_DEMO_2 proc near                ; CODE XREF: main+39p
; функция demo класса DERIVED
        push    offset aDerivedDemo2 ; "DERIVED DEMO 2\n"
        call    printf
        pop     ecx
        retn
DERIVED_DEMO_2 endp

BASE_DEMO_3_   proc near                ; CODE XREF: main+41p
; функция demo_3-го класса BASE
; Внимание! Смотрите, функция демо_3 дважды присутствует в программе! Первый раз она
; входила в объект класса BASE, а второй - в объект класса DERIVED, который
; унаследовал ее от базового класса и сделал копию. Глупо, да? Ведь лучше бы он
; обратился к оригиналу! Зато это упрощает анализ программы...
        push    offset aNonVirtualDeri ; "Non virtual DERIVED DEMO 3\n"
        call    printf
        pop     ecx
        retn
BASE_DEMO_3_   endp

GetBASE_VTBL   proc near                ; CODE XREF: main+9p
; занесение в экземпляр объекта BASE смещения его виртуальной таблицы
        mov     eax, ecx
        mov     dword ptr [eax], offset BASE_VTBL
        retn
GetBASE_VTBL   endp

GetDERIVED_VTBL proc near                ; CODE XREF: main+29p
; занесение в экземпляр объекта DERIVED смещения его виртуальной таблицы
        push    esi
        mov     esi, ecx
        call    GetBASE_VTBL
        ; ага! Значит, наш объект - производный от BASE!

        mov     dword ptr [esi], offset DERIVED_VTBL
        ; занесение указателя на виртуальную таблицу DERIVED

```

```

        mov     eax, esi
        pop     esi
        retn
GetDERIVED_VTBL endp

BASE_VTBL      dd offset BASE_DEMO      ; DATA XREF: GetBASE_VTBL+20
                dd offset BASE_DEMO_2
DERIVED_VTBL   dd offset DERIVED_DEMO   ; DATA XREF: GetDERIVED_VTBL+80
                dd offset DERIVED_DEMO_2
; Обратите внимание на наличие виртуальной таблицы даже там, где она не нужна!

```

Идентификация производных функций. Идентификация производных *невиртуальных* функций весьма тонкий момент. На первый взгляд, коль они вызываются, как и обычные С-функции, распознать, в каком классе была объявлена функция, невозможно — компилятор уничтожает эту информацию еще на стадии компиляции. Уничтожает, да не всю! Перед каждым вызовом функции (не важно, производной или нет) в обязательном порядке формируется указатель `this` — на тот случай, если он понадобится функции, указывающей на объект, из которого вызывается эта функция.

Для производных функций указатель `this` хранит смещение производного, а не базового объекта. Вот оно! Если функция вызывается с различными указателями `this` — это производная функция.

Сложнее выяснить, от какого объекта она происходит. Универсальных решений нет, но если выделить объект А с функциями `f1()`, `f2()`... и объект В с функциями `f1()`, `f3()`, `f4()`... то можно смело утверждать, что `f1()` — функция, производная от класса А. Правда, если из экземпляра класса функция `f1()` не вызывалась ни разу, определить, производная она или нет, не удастся.

Рассмотрим все это на следующем примере:

Листинг 34. Демонстрация идентификации производных функций

```

#include <stdio.h>

class Base{
public:
    void base_demo(void)
    {
        printf("BASE DEMO\n");
    };

    void base_demo_2(void)
    {
        printf("BASE DEMO 2\n");
    };
};

class Derived: public Base{
public:
    void derived_demo(void)
    {
        printf("DERIVED DEMO\n");
    };
};

```

```

void derived_demo_2(void)
{
    printf("DERIVED DEMO 2\n");
};
};

```

Результат компиляции в общем случае должен выглядеть так:

Листинг 35

```

main                proc near                ; CODE XREF: start+AFp
    push    esi
    push    1
    call    ??2@YAPAXI@Z                    ; operator new(uint)
    ; Создаем новый экземпляр некоторого объекта. Пока мы еще не знаем какого,
    ; пусть это будет объект A

    mov     esi, eax                        ; ESI = *a
    add     esp, 4
    mov     ecx, esi                       ; ECX = *a (this)
    call    BASE_DEMO
    ; вызываем BASE_DEMO, обращая внимание на то, что this указывает на 'a'

    mov     ecx, esi                       ; ECX = *a (this)
    call    BASE_DEMO_2
    ; вызываем BASE_DEMO_2, обращая внимание на то, что this указывает на 'a'

    push    1
    call    ??2@YAPAXI@Z                    ; operator new(uint)
    ; создаем еще один экземпляр некоторого объекта, назовем его b

    mov     esi, eax                        ; ESI = *b
    add     esp, 4
    mov     ecx, esi                       ; ECX = *b (this)
    call    BASE_DEMO
    ; Ага! Вызываем BASE_DEMO, но на этот раз this указывает на b,
    ; значит, BASE_DEMO связана родственными отношениями и с 'a', и с 'b'

    mov     ecx, esi
    call    BASE_DEMO_2
    ; Ага! Вызываем BASE_DEMO_2, но на этот раз this указывает на b,
    ; значит, BASE_DEMO_2 связана родственными отношениями и с 'a', и с 'b'

    mov     ecx, esi
    call    DERIVED_DEMO
    ; вызываем DERIVED_DEMO. Указатель this указывает на b, и никаких родственных
    ; связей DERIVED_DEMO с 'a' не замечено. this никогда не указывал на 'a'
    ; при ее вызове

    mov     ecx, esi
    call    DERIVED_DEMO_2
    ; аналогично...

    pop     esi
    retn
main                endp

```

ОК, идентификация неvirtуальных производных функций вполне реальное дело. Единственная сложность — отличить экземпляры двух различных объектов от экземпляров одного и того же объекта.

Что же касается идентификации производных виртуальных функций — об этом уже рассказывалось выше. Производные виртуальные функции вызываются в два этапа — на первом в экземпляр объекта заносится смещение виртуальной таблицы базового класса, а затем оно замещается смещением виртуальной таблицы производного класса. Даже если компилятор оптимизирует код, оставшейся избыточности все равно с лихвой хватит для отличия производных функций от остальных.

Идентификация виртуальных таблиц. Теперь, основательно освоившись с виртуальными таблицами и функциями, рассмотрим очень коварный вопрос: всякий ли массив указателей на функции есть виртуальная таблица? Разумеется, нет! Ведь косвенный вызов функции через указатель частое дело в практике программиста. Массив указателей на функции... хм, конечно, типичным его не назовешь, но и такое в жизни встречается!

Рассмотрим следующий пример — кривой и наигранный, конечно, — но чтобы продемонстрировать ситуацию, где массив указателей жизненно необходим, пришлось бы написать не одну сотню строк кода:

Листинг 36. Демонстрация имитации виртуальных таблиц

```
#include <stdio.h>

void demo_1(void)
{
    printf("Demo 1\n");
}

void demo_2(void)
{
    printf("Demo 2\n");
}

void call_demo(void **x)
{
    ((void (*)(void)) x[0])();
    ((void (*)(void)) x[1])();
}

main()
{
    static void* x[2] =
        { (void*) demo_1, (void*) demo_2};

    // Внимание! Если инициализировать массив не при его объявлении,
    // а по ходу программы, т. е. x[0]=(void *) demo_1...
    // то компилятор сгенерирует адекватный код, заносащий
    // смещения функций в ходе выполнения программы, что будет
    // совсем непохоже на виртуальную таблицу!
    // Напротив, инициализация при объявлении помещает уже
```

```

    // готовые указатели в сегмент данных, смахивая на настоящую
    // виртуальную таблицу (и экономя такты процессора к тому же)

    call_demo(&x[0]);
}

```

А теперь посмотрим, сможем ли мы отличить рукотворную таблицу указателей от настоящей:

Листинг 37

```

main          proc near          ; CODE XREF: start+AFp
    push      offset Like_VTBL
    call      demo_call
    ; Ага, функции передается указатель на нечто очень похожее на виртуальную
    ; таблицу. Но мы-то, уже умудренные опытом, с легкостью раскалываем эту
    ; грубую подделку. Во-первых, указатели на VTBL так просто не передаются
    ; (там не такой тривиальный код), во-вторых, они передаются не через стек,
    ; а через регистр. В-третьих, указатель на виртуальную таблицу ни одним
    ; существующим компилятором не используется непосредственно, а помещается
    ; в объект. Тут же нет ни объекта, ни указателя this, в-четвертых, -
    ; словом, это не виртуальная таблица, хотя на беглый, нетренированный
    ; взгляд очень на нее похожа...

    pop       ecx
    retn
main          endp

demo_call     proc near          ; CODE XREF: sub_0_401030+5p
arg_0         = dword ptr 8
    ; Вот-с! Указатель - аргумент, а к виртуальным таблицам идет обращение
    ; через регистр...

    push      ebp
    mov       ebp, esp
    push      esi
    mov       esi, [ebp+arg_0]
    call      dword ptr [esi]
    ; происходит двухуровневый вызов функции - по указателю на массив
    ; указателей на функцию, что характерно для вызова виртуальных функций,
    ; но опять-таки слишком тривиальный код - вызов виртуальных функций
    ; сопряжен с большой избыточностью, а кроме того, опять нет указателя this

    call      dword ptr [esi+4]
    ; аналогично - слишком просто для вызова виртуальной функции

    pop       esi
    pop       ebp
    retn
demo_call     endp

Like_VTBL     dd offset demo_1    ; DATA XREF:main
              dd offset demo_2
              ; массив указателей внешне похож на виртуальную таблицу, но
              ; расположен "не там", где обычно располагаются виртуальные таблицы.

```

Обобщая выводы, разбросанные по комментариям, повторим основные признаки подделки еще раз:

- слишком тривиальный код, минимум используемых регистров и никакой избыточности, обращение к виртуальным таблицам происходит куда витиеватее;
- указатель на виртуальную функцию заносится в экземпляр объекта, и передается он не через стек, а через регистр (точнее см. раздел «Идентификация *this*»);
- отсутствует указатель *this*, всегда подготавливаемый перед вызовом виртуальной функции;
- виртуальные функции и статические переменные располагаются в различных местах сегмента данных, поэтому сразу можно отличить одни от других.

А можно ли так организовать вызов функции по ссылке, чтобы компиляция программы давала код, идентичный вызову виртуальной функции? Как сказать... Теоретически да, но практически едва ли такое удастся осуществить (а уж непреднамеренно — тем более). Код вызова виртуальных функций в связи с большой избыточностью очень специфичен и легко различим на глаз. Легко симитировать общую технику работы с виртуальными таблицами, но без ассемблерных вставок невозможно воспроизвести ее в точности.

Заключение. Вообще же, как мы видим, работа с виртуальными функциями сопряжена с огромной избыточностью и «тормозами», а их анализ связан с большими трудозатратами — приходится постоянно держать в голове множество указателей и помнить, какой из них на что указывает. Но как бы там ни было, никаких принципиально неразрешимых преград перед исследователем не стоит.

Идентификация конструктора и деструктора

То, что не существует в одном тексте (одном возможном мире), может существовать в других текстах (возможных мирах).

Тезис семантики возможных миров

Конструктор в силу своего автоматического вызова при создании нового экземпляра объекта — первая по счету вызываемая функция объекта. Так какие же могут возникнуть сложности в его идентификации? Камень преткновения в том, что конструктор *факультативен*, т. е. может присутствовать в объекте, а может и не присутствовать. Поэтому совсем не факт, что первая вызываемая функция — конструктор!

Заглянув в описание языка Си++, можно обнаружить, что конструктор не возвращает никакого значения, что нехарактерно для обычных функций, однако все же не настолько редко встречается, чтобы однозначно идентифицировать конструктор. Как же тогда быть?

Выручает то обстоятельство, что по стандарту конструктор не должен автоматически вызывать исключения, даже если отвести память под объект не уда-

лось. Реализовать это требование можно множеством различных способов, но все виденные автором компиляторы просто помещают перед вызовом конструктора проверку на нулевой указатель, передавая ему управление *только* при удачном выделении памяти для объекта. Напротив, все остальные функции объекта вызываются всегда, даже при неуспешном выделении памяти. Точнее, *пытаются вызываться*, но нулевой указатель (возвращенный при ошибке отведения памяти) при первой же попытке обращения вызывает исключение, передавая бразды правления обработчику соответствующей исключительной ситуации.

Таким образом, функция, окольцованная проверкой нулевого указателя, и есть конструктор, а не что-либо иное. Теоретически, впрочем, подобная проверка может присутствовать и при вызове других функций, конструктором не являющихся, но... во всяком случае, автору на практике с таким еще не приходилось встречаться.

Деструктор, как и конструктор, факультативен, т. е. последняя вызываемая функция объекта может и не быть деструктором. Тем не менее отличить деструктор от любой другой функции очень просто — он вызывается только при результативном создании объекта (т. е. успешном выделении памяти) и игнорируется в противном случае. Это документированное свойство языка, следовательно, обязательное к реализации всеми компиляторами. Таким образом, в код помещается такое же «кольцо», как и у конструктора, но никакой путаницы не возникает, так как конструктор вызывается всегда первым (если он есть), а деструктор — последним.

Особый случай представляет объект, целиком состоящий из одного конструктора (или деструктора), — попробуй разберись, с чем мы имеем дело. Но разобратся можно! За вызовом конструктора практически всегда присутствует код, обнуляющий `this` в случае неудачного выделения памяти, а у деструктора этого нет! Далее, деструктор обычно вызывается не непосредственно из материнской процедуры, а из функции-обертки, вызывающей помимо деструктора и оператор `delete`, освобождающий занятую объектом память. Так что отличить конструктор от деструктора вполне можно!

Давайте для лучшего уяснения сказанного рассмотрим следующий пример:

Листинг 38. Демонстрация конструктора и деструктора

```
#include <stdio.h>

class MyClass{
public:
    MyClass(void);
    void demo(void);
    ~MyClass(void);
};

MyClass::MyClass()
{
    printf("Constructor\n");
}
```

```

MyClass::~MyClass()
{
    printf("Destructor\n");
}

void MyClass::demo(void)
{
    printf("MyClass\n");
}

main()
{
    MyClass *zzz = new MyClass;
    zzz->demo();
    delete zzz;
}

```

Результат его компиляции в общем случае должен выглядеть так:

Листинг 39

```

Constructor    proc near                ; CODE XREF: main+11p
; функция конструктора. То, что это именно конструктор, можно понять из реализации
; его вызова (см. main):
    push     esi
    mov     esi, ecx
    push     offset aConstructor      ; "Constructor\n"
    call     printf
    add     esp, 4
    mov     eax, esi
    pop     esi
    retn
Constructor    endp

Destructor     proc near                ; CODE XREF: __destructor+6p
; функция деструктора. То, что это именно деструктор, можно понять из реализации
; его вызова (см. main):
    push     offset aDestructor      ; "Destructor\n"
    call     printf
    pop     ecx
    retn
Destructor     endp

demo          proc near                ; CODE XREF: main+1Ep
; обычная функция demo.
    push     offset aMyClass         ; "MyClass\n"
    call     printf
    pop     ecx
    retn
demo          endp

main          proc near                ; CODE XREF: start+AFp
    push     esi
    push     1
    call     ??2@YAPAXI@Z            ; operator new(uint)

```



```

    add     esp, 4
    ; выделяем память для нового объекта, точнее, пытаемся это сделать

    test    eax, eax
    jz      short loc_0_40105A
    ; Проверка успешности выделения памяти для объекта.
    ; Обратите внимание, куда направлен jump.
    ; Он направлен на инструкцию XOR ESI,ESI, обнуляющую указатель на объект, -
    ; при попытке использования нулевого указателя возникнет исключение,
    ; но конструктор не должен вызывать исключение, даже если память под объект
    ; отвести не удалось.
    ; Поэтому конструктор получает управление только при успешном отводе памяти!
    ; Следовательно, функция, находящаяся до XOR ESI,ESI, и есть конструктор!!!
    ; И мы сумели надежно идентифицировать ее.

    mov     ecx, eax
    ; Готовим указатель this

    call    Constructor
    ; эта функция - конструктор, так как вызывается только при удачном отводе памяти

    mov     esi, eax
    jmp     short loc_0_40105C

loc_0_40105A:                                ; CODE XREF: main+Dj
    xor     esi, esi
    ; обнуляем указатель на объект, чтобы вызвать исключение при попытке его
    ; использования
    ; Внимание: конструктор никогда не вызывает исключения, поэтому
    ; нижележащая функция гарантированно не является конструктором

loc_0_40105C:                                ; CODE XREF: main+18j
    mov     ecx, esi
    ; готовим указатель this

    call    demo
    ; вызываем обычную функцию объекта

    test    esi, esi
    jz      short loc_0_401070
    ; проверка указателя this на NULL. Деструктор вызывается только в том случае,
    ; если память под объект была отведена (если же она не была отведена,
    ; освободить особо нечего).
    ; Таким образом, следующая функция именно деструктор, а не что-нибудь еще

    push    1
    ; количество байтов для освобождения (необходимо для delete)

    mov     ecx, esi
    ; готовим указатель this

    call    __destructor
    ; вызываем деструктор

loc_0_401070:                                ; CODE XREF: main+25j
    pop     esi
    retn

main     endp

```

```

__destructor proc near                ; CODE XREF: main+2Bp
; функция деструктора. Обратите внимание, что деструктор обычно вызывается
; из той же функции, что и delete (хотя так бывает и не всегда, но очень часто)

arg_0      = byte ptr 8
    push    ebp
    mov     ebp, esp
    push    esi
    mov     esi, ecx
    call    Destructor
    ; Вызываем функцию деструктора, определенную пользователем

    test    [ebp+arg_0], 1
    jz      short loc_0_40109A
    push    esi
    call    ???@YAXPAX@Z              ; operator delete(void *)
    add     esp, 4
    ; освобождаем память, ранее выделенную объекту,

loc_0_40109A:                          ; CODE XREF: __destructor+Fj
    mov     eax, esi
    pop     esi
    pop     ebp
    retn    4

__destructor endp

```

Объекты в автоматической памяти, или Когда конструктор/деструктор идентифицировать невозможно. Если объект размещается в стеке (автоматической памяти), то никаких проверок успешности ее выделения не выполняется и вызов конструктора становится неотличим от вызова остальных функций. Аналогичная ситуация и с деструктором — стековая память автоматически освобождается по завершении функции, а вместе с ней умирает и сам объект безо всякого вызова delete (delete применяется только для удаления объектов из кучи).

Чтобы убедиться в этом, модифицируем функцию main нашего предыдущего примера следующим образом:

Листинг 40

```

main()
{
    MyClass zzz;
    zzz.demo();
}

```

Результат компиляции в общем случае должен выглядеть так:

Листинг 41

```

main proc near                        ; CODE XREF: start+AFp

var_4      = byte ptr -4
; локальная переменная zzz - экземпляр объекта MyClass

    push    ebp
    mov     ebp, esp

```

```
    push    ecx
    lea     ecx, [ebp+var_4]
    ; подготавливаем указатель this

    call    constructor
    ; вызываем конструктор, как и обычную функцию!
    ; догадаться, что это конструктор, можно разве что по его содержимому
    ; (обычно конструктор инициализирует объект), да и то неуверенно

    lea     ecx, [ebp+var_4]
    call    demo
    ; вызываем функцию demo. Обратите внимание, ее вызов ничем не отличается
    ; от вызова конструктора!

    lea     ecx, [ebp+var_4]
    call    destructor
    ; вызываем деструктор - его вызов, как мы уже поняли, ничем
    ; характерным не отмечен

    mov     esp, ebp
    pop     ebp
    retn

main      endp
```

Идентификация конструктора/деструктора в глобальных объектах. Глобальные объекты (также называемые *статическими объектами*) размещаются в сегменте данных еще на стадии компиляции. Стало быть, ошибки выделения памяти в принципе невозможны, и выходит, что по аналогии со стековыми объектами надежно идентифицировать конструктор/деструктор и здесь нельзя? А вот и нет!

Глобальный объект, в силу своей глобальности, доступен из многих мест программы, но его конструктор должен вызываться лишь *однажды*. Как можно это обеспечить? Конечно, возможны самые различные варианты реализации, но большинство компиляторов идут по простейшему пути, используя для этой цели глобальную переменную-флаг, изначально равную нулю, а перед первым вызовом конструктора увеличивающуюся на единицу (в более общем случае устанавливающуюся в TRUE). При повторных итерациях остается проверить, равен ли флаг нулю, и, если нет — пропустить вызов конструктора. Таким образом, конструктор вновь «окольцовывается» условным переходом, что позволяет безошибочно отличить его от всех остальных функций.

С деструктором еще проще — раз объект глобальный, то он уничтожается только при завершении программы. А кто это может отследить, кроме поддержки времени исполнения? Специальная функция, такая, как `_atexit`, принимает на вход указатель на конструктор, запоминает его и затем вызывает при возникновении в этом необходимости. Интересный момент — функция `_atexit` (или что там используется в вашем конкретном случае) должна быть вызвана лишь однократно (надеюсь, понятно почему?). И чтобы не вводить еще один флаг, она вызывается сразу же после вызова конструктора! На первый взгляд объект может показаться состоящим из одних конструктора/деструктора, но это не так! Не забывайте, что `_atexit` не передает немедленно управление на код деструктора, а только запоминает его указатель для дальнейшего использования!

Таким образом, конструктор/деструктор глобального объекта очень просто идентифицировать, что и доказывает следующий пример:

Листинг 42

```
main()
{
    static MyClass zzz;
    zzz.demo();
}
```

Результат его компиляции в общем случае должен выглядеть так:

Листинг 43

```
main          proc near          ; CODE XREF: start+AFp
    mov       cl, byte_0_4078E0  ; флаг инициализации экземпляра объекта
    mov       al, 1
    test      al, cl
    ; объект инициализирован?

    jnz       short loc_0_40106D
    ; -> да, инициализирован, не вызываем конструктор

    mov       dl, cl
    mov       ecx, offset unk_0_4078E1 ; экземпляр объекта
    ; готовим указатель this

    or        dl, al
    ; устанавливаем флаг инициализации в TRUE
    ; и вызываем конструктор

    mov       byte_0_4078E0, dl    ; флаг инициализации экземпляра объекта
    call      constructor
    ; Вызов конструктора.
    ; Обратите внимание, что, если экземпляр объекта уже инициализирован
    ; (см. проверку выше), конструктор не вызывается.
    ; Таким образом, его очень легко отождествить!

    push      offset thunk_destructo
    call      _atexit
    add       esp, 4
    ; Передаем функции _atexit указатель на деструктор,
    ; который она должна вызвать по завершении программы

loc_0_40106D:          ; CODE XREF: main+Aj
    mov       ecx, offset unk_0_4078E1 ; экземпляр объекта
    ; готовим указатель this

    jmp       demo
    ; вызываем demo

main            endp

thunk_destructo:      ; DATA XREF: main+200
    ; переходник к функции-деструктору
```

```

        mov     ecx, offset unk_0_4078E1 ; экземпляр объекта
        jmp     destructor

byte_0_4078E0 db 0                      ; DATA XREF: mainr main+15w
                                         ; флаг инициализации экземпляра объекта
unk_0_4078E1 db 0 ;                      ; DATA XREF: main+E0 main+2Do ...
                                         ; экземпляр объекта

```

Аналогичный код генерирует и Borland C++. Единственное отличие — более хитрый вызов деструктора. Вызовы всех деструкторов помещены в специальную процедуру, которая выдает себя тем, что обычно располагается перед библиотечными функциями или в непосредственной близости от них, так что идентифицировать ее очень легко. Смотрите сами:

Листинг 44

```

_main      proc near                      ; DATA XREF: DATA:00407044o
        push   ebp
        mov    ebp, esp
        cmp    ds:byte_0_407074, 0      ; флаг инициализации объекта
        jnz    short loc_0_4010EC
        ; Если объект уже инициализирован - конструктор не вызывается

        mov    eax, offset unk_0_4080B4 ; Экземпляр объекта
        call   constructor
        inc    ds:byte_0_407074        ; флаг инициализации объекта
        ; Увеличиваем флаг на единицу, возводя его в TRUE

loc_0_4010EC:                      ; CODE XREF: _main+Aj
        mov    eax, offset unk_0_4080B4 ; Экземпляр объекта
        call   demo
        ; Вызов функции demo

        xor     eax, eax
        pop     ebp
        retn

_main      endp

        call_destruct proc near          ; DATA XREF: DATA:004080A4o
; Эта функция содержит в себе вызовы всех деструкторов глобальных объектов,
; поскольку вызов каждого деструктора "окольцован" проверкой флага инициализации,
; эту функцию легко идентифицировать - только она содержит подобный "колечный код"
; (вызовы конструкторов обычно разбросаны по всей программе)

        push   ebp
        mov    ebp, esp
        cmp    ds:byte_0_407074, 0      ; флаг инициализации объекта
        jz     short loc_0_401117
        ; объект был инициализирован?

        mov    eax, offset unk_0_4080B4 ; Экземпляр объекта
        ; готовим указатель this

        mov     edx, 2
        call   destructor
        ; вызываем деструктор

```

```
loc_0_401117:                ; CODE XREF: call_destruct+Aj  
    pop     ebp  
    retn  
call_destruct endp
```

Виртуальный деструктор. Деструктор тоже может быть виртуальным! А почему бы и нет? Это бывает полезно, когда экземпляр производного класса удаляется через указатель на базовый объект. Поскольку виртуальные функции связаны с классом объекта, а не с классом указателя, то вызывается виртуальный деструктор, связанный с типом объекта, а не с типом указателя. Впрочем, эти тонкости относятся к непосредственному программированию, а исследователей в первую очередь интересует, как идентифицировать виртуальный деструктор. О, это просто — виртуальный деструктор совмещает в себе свойства обычного деструктора и виртуальной функции (см. раздел «Идентификация виртуальных функций»).

Виртуальный конструктор. Виртуальный конструктор?! А что, разве есть такой? Ничего подобного стандартный C++ не поддерживает. *Непосредственно не поддерживает.* И когда виртуальный конструктор позарез требуется программистам (впрочем, это бывает лишь в весьма экзотических случаях), они прибегают к ручной эмуляции некоторого его подобия. В специально выделенную для этих целей виртуальную функцию (не конструктор!) помещается приблизительно следующий код: `return new имя класса (*this)` или. Этот трюк кривее, чем бумеранг, но... он работает. Разумеется, существуют и другие решения.

Подробное их обсуждение далеко выходит за рамки данной книги и требует глубокого знания C++ (гораздо более глубокого, чем у рядового разработчика), к тому же это заняло бы слишком много места, но едва ли оказалось бы интересно рядовому читателю.

Итак, идентификация виртуального конструктора в силу отсутствия самого понятия в принципе *невозможна*. Его эмуляция насчитывает десятки (если не больше) решений, попробуй-ка перечисли их все! Впрочем, этого и не нужно делать, в большинстве случаев виртуальные конструкторы представляют собой виртуальные функции, принимающие в качестве аргумента указатель `this` и возвращающие указатель на новый объект. Не слишком-то надежно для идентификации, но все же лучше, чем ничего.

Конструктор раз, конструктор два... Количество конструкторов объекта может быть и более одного (и очень часто не только может, но и бывает). Однако это никак не влияет на анализ. Сколько бы конструкторов ни присутствовало, для каждого экземпляра объекта всегда вызывается только один, выбранный компилятором в зависимости от формы объявления объекта. Единственная деталь — различные экземпляры объекта могут вызывать различные конструкторы. Будьте внимательны!

Зачем козе баян, или внимание: пустой конструктор. Некоторые ограничения конструктора (в частности, отсутствие возвращаемого значения) привели к появлению стиля программирования «пустой конструктор». Конструктор умышленно оставляется пустым, а весь код инициализации помещается в

специальную функцию-член, как правило называемую `Init`. Обсуждение сильных и слабых сторон такого стиля — предмет отдельного разговора, никаким образом не относящегося к данной книге. Исследователям достаточно знать — такой стиль есть и активно используется не только индивидуальными программистами, но и крупнейшими компаниями-гигантами (например, той же `Microsoft`). Поэтому, встретив вызов пустого конструктора, не удивляйтесь, это нормально, и ищите функцию инициализации среди обычных членов.

Идентификация объектов, структур и массивов

Для целого поколения Эйнштейн был глашатаем передовой науки, пророком разума и мира. А сам он в глубине своей кроткой и невозмутимой души без всякой горечи оставался скептиком... Он хотел затеряться и как бы раствориться в окружающем его мире, а оказался одним из самых разрекламированных людей нашего века, и его лицо, вдохновенное и отрешенное от всех грехов мира, стало таким же широко известным, как фотография какой-нибудь кинозвезды.

Ч. П. Сноу. Эйнштейн

Внутренне представление объектов очень похоже на представление структур в языке `C` (по большому счету объекты и есть структуры), поэтому рассмотрим их идентификацию в одной главе.

Структуры очень популярны среди программистов. Позволяя объединить под одной крышей родственные данные, они делают листинг программы более наглядным, упрощая его понимание. Соответственно идентификация структур при дизассемблировании облегчает анализ кода. К великому сожалению исследователей, структуры, как таковые, существует только в исходном тексте программы и практически полностью «перемалываются» при ее компиляции, становясь неотличимыми от обычных, никак не связанных друг с другом переменных.

Рассмотрим следующий пример:

Листинг 45. Пример, демонстрирующий уничтожение структур на стадии компиляции

```
#include <stdio.h>
#include <string.h>

struct zzz
{
    char s0[16];
    int a;
    float f;
};

func(struct zzz y)
// Понятное дело, передачи структуры по значению лучше избегать,
// но здесь это сделано умышленно для демонстрации скрытого создания
// локальной переменной
```

```

{
    printf("%s %x %f\n", &y.s0[0], y.a, y.f);
}

main()
{
    struct zzz y;
    strcpy(&y.s0[0], "Hello, Sailor!");
    y.a=0x666;
    y.f=6.6;
    func(y);
}

```

Результат его компиляции в общем случае должен выглядеть так:

Листинг 46

```

main          proc near          ; CODE XREF: start+AFp

var_18        = byte ptr -18h
var_8         = dword ptr -8
var_4         = dword ptr -4
; члены структуры неотличимы от обычных локальных переменных

    push      ebp
    mov       ebp, esp
    sub       esp, 18h
    ; резервирование места в стеке для структуры

    push      esi
    push      edi
    push      offset aHelloSailor ; "Hello,Sailor!"

    lea       eax, [ebp+var_18]
    ; Указатель на локальную переменную var_18
    ; следующая за ней переменная расположена по смещению 8,
    ; следовательно, 0x18-0x8=0x10 - шестнадцать байтов - именно столько
    ; занимает var_18, что намекает на то, что она - строка
    ; (см. раздел «Идентификация литералов и строк»)

    push      eax
    call      strcpy
    ; копирование строки из сегмента данных в локальную переменную-член структуры

    add       esp, 8
    mov       [ebp+var_8], 666h
    ; занесение в переменную типа DWORD значения 0x666

    mov       [ebp+var_4], 40D33333h
    ; а это значение в формате float равно 6.6
    ; (см. "Идентификация аргументов функций")

    sub       esp, 18h
    ; резервируем место для скрытой локальной переменной, которая используется
    ; компилятором для передачи функции экземпляра структуры по значению
    ; (см. раздел "Идентификация регистровых и временных переменных")

```



```

mov     ecx, 6
; будет скопировано 6 двойных слов, т. е. 24 байта
; 16 - на строку и по четыре на float и int

lea     esi, [ebp+var_18]
; получаем указатель на копируемую структуру

mov     edi, esp
; получаем указатель на только что созданную скрытую локальную переменную

repe    movsd
; копируем!

call    func
; вызываем функцию
; передачи указателя на скрытую локальную переменную - она - не происходит
; и так находится на верху стека.

add     esp, 18h
pop     edi
pop     esi
mov     esp, ebp
pop     ebp
retn

main    endp

```

А теперь заменим структуру последовательным объявлением тех же самых переменных:

Листинг 47. Пример, демонстрирующий сходство структур с обычными локальными переменными

```

main()
{
    char s0[16];
    int a;
    float f;

    strcpy(&s0[0], "Hello, Sailor!");
    a=0x666;
    f=6.6;
}

```

И сравним результат компиляции с предыдущим:

Листинг 48

```

main          proc near          ; CODE XREF: start+AFp

var_18        = dword ptr -18h
var_14        = byte ptr -14h
var_4         = dword ptr -4
; Ага, кажется, есть какое-то различие! Действительно, локальные переменные помещены
; в стек не в том порядке, в котором они были объявлены в программе, а как это
; захотелось компилятору. Напротив, члены структуры обязательно должны помещаться
; в порядке их объявления.
; Но поскольку при дизассемблировании оригинальный порядок следования переменных

```

; неизвестен, определить, "правильно" они расположены или нет, увы,
 ; не представляется возможным.

```

push    ebp
mov     ebp, esp
sub     esp, 18h
; резервируем 0x18 байтов стека (как и в предыдущем примере)

push    offset aHelloSailor ; "Hello,Sailor!"
lea     eax, [ebp+var_14]
push    eax
call    strcpy
add     esp, 8
mov     [ebp+var_4], 666h
mov     [ebp+var_18], 40D33333h
; Смотрите, код аккуратно совпадает байт в байт! Следовательно, невозможно
; автоматически отличить структуру от простого скопища локальных переменных

mov     esp, ebp
pop     ebp
retn
main    endp

```

```
func      proc near          ; CODE XREF: main+36p
```

```
var_8     = qword ptr -8
arg_0     = byte ptr 8
arg_10    = dword ptr 18h
arg_14    = dword ptr 1Ch

```

; Смотрите, хотя функции передается только один аргумент - экземпляр структуры, -
 ; в дизассемблерном тексте он не отличим от последовательной засылки в стек
 ; нескольких локальных переменных! Поэтому восстановить подлинный прототип
 ; функции невозможно!

```

push    ebp
mov     ebp, esp
fld     [ebp+arg_14]
; загрузить в стек FPU вещественное целое, находящееся по смещению
; 0x14 относительно указателя eax

sub     esp, 8
; зарезервировать 8 байтов под локальные переменные

fstp    [esp+8+var_8]
; перепихнуть считанное вещественное значение в локальную переменную

mov     eax, [ebp+arg_10]
push    eax
; прочитать только что "перепихнутую" вещественную переменную
; и затолкать ее в стек

lea     ecx, [ebp+arg_0]
; получить указатель на первый аргумент

push    ecx
push    offset aSXF          ; "%s %x %f\n"
call    printf

```

```
add     esp, 14h
pop     ebp
ret     4
func    endp
```

Выходит, отличить структуру от обычных переменных невозможно? Неужто исследователю придется самостоятельно распознавать «родство» данных и связывать их «брачными узами», порой ошибаясь и неточно воспроизводя исходный текст программы?

Как сказать... И да и нет одновременно. «Да» — экземпляр структуры, использующийся в той же единице трансляции, в которой он был объявлен, «развертывается» еще на стадии компиляции в самостоятельные переменные, обращение к которым происходит индивидуально по их фактическим адресам (возможно, косвенным). «Нет» — если в области видимости находится один лишь указатель на экземпляр структуры. Тогда обращение ко всем членам структуры происходит через указатель на этот экземпляр структуры (так как структура не присутствует в области видимости, например, передается другой функции по ссылке, вычислить фактические адреса ее членов на стадии компиляции невозможно).

Постойте, но ведь точно так происходит обращение и к элементам массива: базовый указатель указывает на начало массива, к нему добавляется смещение искомого элемента относительно начала массива (индекс элемента, умноженный на его размер), результат вычислений и будет фактическим указателем на искомый элемент!

Единственное фундаментальное отличие массивов от структур состоит в том, что массивы **гомогенны** (т. е. состоят из элементов одинакового типа), а структуры **могут** быть как гомогенными, так и **гетерогенными** (состоящими из элементов различных типов). Таким образом, задача идентификации структур и массивов сводится, во-первых, к выделению ячеек памяти, адресуемых через общий для всех них базовый указатель, и, во-вторых, определению типа этих переменных (см. раздел «Идентификация типов данных»). Если удастся выделить более одного типа, скорее всего, перед нами структура, в противном случае это с равным успехом может быть и структурой, и массивом — тут уж придется смотреть по обстоятельствам и самой программе.

С другой стороны, если программисту вздумается подсчитать зависимость выпитого количества пива от дня недели, он может выделить для учета либо массив `day[7]`, либо завести структуру `struct week{int Monday; int Tuesday; ...}`. И в том и в другом случае сгенерированный компилятором код будет одинаков, да не только код, но и смысл! В этом контексте структура неотличима от массива и физически и логически, выбор той или иной конструкции — дело вкуса. Также возьмите себе на заметку, что массивы, как правило, длинны, а обращение к их элементам часто сопровождается различными математическими операциями, совершаемыми над указателем. Далее, обработка элементов массива, как правило, осуществляется в цикле, а члены структуры, по обыкновению, «разбираются» индивидуально (хотя некоторые программисты позволяют себе вольность обращаться со структурой как с массивом). Еще неприятнее, что языки C/C++ допускают (если не ска-

зять, провоцируют) явное преобразование типов и... ой, а ведь в этом случае при дизассемблировании не удастся установить, имеем ли мы дело с объединенными под одну крышу разнотипными данными (т. е. структуру), или же это массив с ручным преобразованием типа своих элементов. Хотя, строго говоря, после подобных преобразований массив превращается в самую настоящую структуру! (Массив по определению гомогенен и данные разных типов хранить не может.)

Модифицируем предыдущий пример, передав функции не саму структуру, а указатель на нее, и посмотрим, что за код сгенерировал компилятор.

Листинг 49

```

funct      proc near          ; CODE XREF: sub_0_401029+29p
var_8      = qword ptr -8
arg_0      = dword ptr 8
; Aga! Функция принимает только один аргумент!

    push    ebp
    mov     ebp, esp
    mov     eax, [ebp+arg_0]
    ; Загружаем переданный функции аргумент в EAX

    fld     dword ptr [eax+14h]
    ; Загружаем в стек FPU вещественное значение, находящееся по смещению
    ; 0x14 относительно указателя EAX.
    ; Таким образом, во-первых, EAX (аргумент, переданный функции) – это указатель.
    ; Во-вторых, это не просто указатель, а базовый указатель, использующийся
    ; для доступа к элементам структуры или массива.
    ; Запомним тип первого элемента (вещественное значение) и продолжим анализ

    sub     esp, 8
    ; резервируем 8 байтов под локальные переменные

    fstp    [esp+8+var_8]
    ; перепихиваем считанное вещественное значение в локальную переменную var_8

    mov     ecx, [ebp+arg_0]
    ; Загружаем в ECX значение переданного функции указателя

    mov     edx, [ecx+10h]
    ; загружаем в EDX значение, лежащее по смещению 0x10.
    ; Aga! Это явно не вещественное значение, следовательно, мы имеем дело со
    ; структурой

    push    edx
    ; заталкиваем только что считанное значение в стек

    mov     eax, [ebp+arg_0]
    push    eax
    ; получаем указатель на структуру (т. е. на ее первый член)
    ; и запикиваем его в стек. Поскольку ближайший элемент
    ; находится по смещению 0x10, то первый элемент структуры, по-видимому,
    ; занимает все эти 0x10 байтов, хотя это и не обязательно, – возможно, остальные
    ; члены структуры просто не используются. Установить, как все обстоит на самом
    ; деле, можно, обратившись к вызывающей (материнской) функции, которая и

```

```

; инициализировала эту структуру, но и без этого мы можем восстановить
; ее приблизительный вид
; struct xxx{
; char x[0x10] || int x[4] || __int16[8] || __int64[2];
; int y;
; float z;
; }

push    offset aSXF ; "%s %x %f\n"
; строка спецификаторов позволяет уточнить типы данных. Так, первый элемент -
; это, бесспорно, char x[0x10], поскольку он выводится как строка,
; следовательно, наше предварительное предположение о формате структуры
; верное!

call    printf
add     esp, 14h
pop     ebp
retn

funct    endp

main     proc near          ; CODE XREF: start+AFp

var_18   = byte ptr -18h
var_8    = dword ptr -8
var_4    = dword ptr -4
; Смотрите, на первый взгляд мы имеем дело с несколькими локальными переменными,
; но давайте не будем торопиться с их идентификацией!

push     ebp
mov      ebp, esp
sub      esp, 18h
; Открываем кадр стека

push     offset aHelloSailor ; "Hello,Sailor!"
lea      eax, [ebp+var_18]
push     eax
call     unknown_libname_1
; unknown_libmane_1 - это strcpy, и понять это можно, даже не анализируя ее код.
; Функция принимает два аргумента - указатель на локальный буфер из 0x10 байтов
; (размер 0x10 получен вычитанием смещения ближайшей переменной от смещения
; самой этой переменной относительно кадра стека) такой же точно прототип
; и strcmp, но это не может быть strcmp, так как локальный буфер
; не инициализирован и он может быть только буфером-приемником

add      esp, 8
; выталкиваем аргументы из стека

mov      [ebp+var_8], 666h
; инициализируем локальную переменную var_8 типа DWORD

mov      [ebp+var_4], 40D33333h
; инициализируем локальную переменную var_4 типа... нет, не DWORD
; (хотя она и выглядит как DWORD). Проанализировав, как эта переменная
; используется в функции funct, которой она передается, мы распознаем
; в ней вещественное значение размером 4 байта. Стало быть, это float
; (подробнее см. раздел «Идентификация аргументов функций»)

```

```

lea    ecx, [ebp+var_18]
push   ecx
; Вот теперь самое главное! Функции передается указатель на локальную
; переменную var_18 - строковый буфер размером в 0x10 байтов,
; но анализ вызываемой функции позволил установить, что она обращается не
; только к первым 0x10 байтам стека материнской функции, а ко всем 0x18!
; Следовательно, функции передается не указатель на строковый буфер,
; а указатель на структуру
;
; struct x{
; char var_18[10];
; int var_8;
; float var_4
; }
;
; Поскольку типы данных различны, то это именно структура, а не массив.

call   funct
add    esp, 4
mov    esp, ebp
pop    ebp
retn
sub_0_401029  endp

```

Идентификация объектов. Объекты языка C++ — это, по сути дела, структуры, совмещающие в себе данные, методы их обработки (функции то бишь) и атрибуты защиты (типа public, friend...).

Элементы-данные объекта обрабатываются компилятором равно как и обычные члены структуры. Невиртуальные функции вызываются по фактическому смещению и в объекте отсутствуют. Виртуальные функции вызываются через специальный указатель на виртуальную таблицу, помещенный в объект, а атрибуты защиты уничтожаются еще на стадии компиляции. Отличить публичную функцию от защищенной можно только тем, что публичная вызывается и из других объектов, а защищенная — только из своего объекта.

Теперь обо всем этом подробнее. Итак, объект (вернее, экземпляр объекта). Что он собой представляет?

Пусть у нас есть следующий объект:

Листинг 50. Пример, демонстрирующий строение объекта

```

class MyClass{
    void demo_1(void);
    int a;
    int b;

public:
    virtual void demo_2(void);
    int c;
};

MyClass zzz;

```

Экземпляр объекта `zzz` «перемелется» компилятором в следующую структуру (рис. 13):

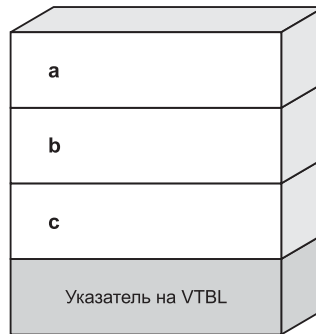


Рис. 13. Представление экземпляра объекта в памяти

Перед исследователем встают следующие проблемы: как отличить объекты от простых структур? Как определить размер объектов? Как определить, какая функция к какому объекту принадлежит? Как... Погодите, погодите, не все сразу! Начнем отвечать на вопросы по порядку согласно социалистической очереди.

Вообще же, строго говоря, отличить объект от структуры невозможно в силу того, что объект и есть структура с членами, приватными по умолчанию. При объявлении объектов можно пользоваться и ключевым словом `struct`, и ключевым словом `class`. Причем для классов, все члены которых открыты, предпочтительнее использовать именно `struct`, так как члены структуры уже публичны по умолчанию. Сравните два следующих примера:

Листинг 51. Классы — это структуры с членами, приватными по умолчанию

<pre>struct MyClass{ void demo(void); int x; private: void demo_private(void); int y; };</pre>	<pre>class MyClass{ void demo_private(void); int y; public: void demo(void); int x; };</pre>
--	--

Одна запись отличается от другой лишь синтаксически, а код, генерируемый компилятором, будет идентичен! Поэтому с надеждой научиться отличать объекты от структур следует как можно скорее расстаться.

ОК, условимся считать объектами структуры, содержащие одну или более функций. Вот только как определить, какая функция какому объекту принадлежит? С виртуальными функциями все просто — они вызываются косвенно, через указатель на виртуальную таблицу, помещаемый компилятором в каждый экземпляр объекта, к которому принадлежит данная виртуальная функция. Невиртуальные функции вызываются по их фактическому адресу, равно как и обычные функции, не принадлежащие никакому объекту. Положение безнадежно? От-

нудь нет! Каждой функции-члену объекта передается неявный аргумент — указатель *this*, ссылающийся на экземпляр объекта, к которому принадлежит данная функция. Экземпляр объекта — это, правда, не сам объект, но нечто очень тесно с ним связанное, поэтому восстановить исходную структуру объектов дизассемблируемой программы вполне реально (подробнее об этом см. раздел «Объекты и экземпляры»).

Размер объектов определяется теми же указателями *this* — как разница соседних указателей (если объекты расположены в стеке или в сегменте данных). Если же экземпляры объектов создаются оператором *new* (как часто и бывает), то в код помещается вызов функции *new*, принимающий в качестве аргумента количество выделяемых байтов, это и есть размер объекта.

Вот, собственно, и все. Остается добавить, что многие компиляторы, создавая экземпляр объекта, не содержащего ни данных, ни виртуальных функций, все равно выделяют под него минимальное количество памяти (обычно один байт), хотя *никак* его не используют. На какой же, извините за грубость, хвост такое делать? Память — она не резиновая, а из кучи одни байты и не выделишь — за счет грануляции отъедается солидный кусок, размер которого варьируется в зависимости от реализации самой кучи от 4 байтов до 4 килобайтов!

Причина в том, что компилятору жизненно необходимо определить указатель *this*, — нулевым, увы, *this* быть не может — это вызвало бы исключение при первой же попытке обращения. Да и оператору *delete* надо что-то удалять, а раз так, это «что-то» надо предварительно выделить...

Эх, хоть разработчики C++ не устают повторять, что их язык не уступает по эффективности чистому C, все известные мне реализации C++ компиляторов генерируют ну очень кривой и тормозной код! Ладно, все это лирика, перейдем к рассмотрению конкретных примеров.

Листинг 52

```
#include <stdio.h>

class MyClass{
public:
    void demo(void);
    int x;
private:
    demo_private(void);
    int y;
};

void MyClass::demo_private(void)
{
    printf("Private\n");
}

void MyClass::demo(void)
{
    printf("MyClass\n");
    this->demo_private();
}
```



```

        this->y=0x666;
    }

main()
{
    MyClass *zzz = new MyClass;
    zzz->demo();
    zzz->x=0x777;
}

```

Результат его компиляции в общем случае должен выглядеть так:

Листинг 53

```

main          proc near          ; CODE XREF: start+AFp
    push      esi
    push      8
    call      ???2@YAPAXI@Z      ; operator new(uint)
    ; Выделяем 8 байтов под экземпляр некоторого объекта оператором new.
    ; Вообще-то вовсе не факт, что память выделяется именно под объект
    ; (может, тут было что-то типа char *x = new char[8]), так что
    ; не будем считать это утверждение догмой, а примем как рабочую гипотезу,
    ; дальнейшие исследования покажут, что к чему

    mov       esi, eax
    add       esp, 4

    mov       ecx, esi
    ; Ухо-хвост Тигра! Готовится указатель this, который передается функции
    ; через регистр. Значит, ECX не что иное, как указатель на экземпляр объекта!
    ; (подробнее об этом см. раздел "Идентификация this").

    call      demo
    ; Вот мы и добрались до вызова функции демо - открываем хвост Тигре!
    ; Пока не ясно, что эта функция делает (символьное имя дано ей для наглядности),
    ; но известно, что она принадлежит экземпляру объекта, на который
    ; указывает ECX. Назовем этот экземпляр "а". Далее, поскольку
    ; функция, вызывающая демо (т. е. функция, в которой мы сейчас находимся), не
    ; принадлежит к "а" (она же его сама и создала - не мог же экземпляр объекта
    ; сам "вытянуть себя за волосы"), то функция demo - это public-функция.
    ; Неплохо для начала?

    mov       dword ptr [esi], 777h
    ; Так, так... мы помним, что ESI указывает на экземпляр объекта, тогда
    ; выходит, что в объекте есть еще один public-член - это переменная
    ; типа int.
    ; По предварительным заключениям объект выглядел так:
    ; class myclass{
    ; public:
    ; void demo(void); // void - так как функция ничего не принимает и не возвращает
    ; int x;
    ; };

    pop       esi
    retn
main          endp

```

```

demo          proc near                ; CODE XREF: main+Fp
; вот мы в функции demo - члене объекта A

    push     esi
    mov      esi, ecx
    ; Загружаем в ECX указатель this, переданный функции

    push     offset aMyclass           ; "MyClass\n"
    call     printf
    add      esp, 4
    ; Выводим строку на экран... это неинтересно, но вот дальше...

    mov      ecx, esi
    call     demo_private
    ; Опля, вот он, наш Тигра! Вызывается еще одна функция! Судя по this,
    ; эта функция нашего объекта, причем, вероятнее всего, имеющая атрибут private,
    ; поскольку вызывается только из функции самого объекта.

    mov      dword ptr [esi+4], 666h
    ; Так, в объекте есть еще одна переменная, вероятно, приватная. Тогда,
    ; по современным воззрениям, объект должен выглядеть так:
    ; class myclass{
    ; void demo_provate(void);
    ; int y;
    ; public:
    ; void demo(void); // void -так как функция ничего не принимает и не возвращает
    ; int x;
    ; }
    ;
    ; Итак, мы не только идентифицировали объект, но даже восстановили его
    ; структуру! Пускай не застрахованную от ошибок (так, предположение
    ; о приватности "demo_private" и "y" базируется лишь на том, что они ни разу
    ; не вызывались извне объекта), но все же не так ООП страшно, как его
    ; малюют, и восстановить если не подлинный исходный текст программы, то хотя бы
    ; какое-то его подобие вполне возможно!

    pop      esi
    retn

demo          endp

demo_private  proc near                ; CODE XREF: demo+12p
; приватная функция demo. - ничего интересного
    push     offset aPrivate           ; "Private\n"
    call     printf
    pop      ecx
    retn

demo_private  endp

```

Объекты и экземпляры. В коде, сгенерированном компилятором, никаких объектов и в помине нет, одни лишь *экземпляры объектов*. Вроде бы, да какая разница-то? Экземпляр объекта разве не есть сам объект? Нет, между объектом и экземпляром существует принципиальная разница. Объект — это *структура*, в то время как экземпляр объекта (в сгенерированном коде!) — подструктура этой структуры. Иными словами, пусть имеется объект А, включающий в себя функ-

ции a_1 и a_2 . Далее, пусть создано два его экземпляра — из одного мы вызываем функцию a_1 , из другого — a_2 . С помощью указателя `this` мы сможем выяснить лишь то, что одному экземпляру принадлежит функция a_1 , другому — a_2 . Но установить, являются ли эти экземпляры экземплярами одного объекта или экземплярами двух разных объектов, невозможно! Ситуация усугубляется тем, что в производных классах наследуемые функции не дублируются (во всяком случае, так поступают «умные» компиляторы, хотя в жизни случается всякое). Возникает двусмысленность: если с одним экземпляром связаны функции a_1 и a_2 , а с другим — a_1 , a_2 и a_3 , то это могут быть либо экземпляры одного класса (просто из первого экземпляра функция a_3 не вызывается), то ли второй экземпляр — экземпляр класса, производного от первого. Код, сгенерированный компилятором, в обоих случаях будет идентичным! Приходится восстанавливать иерархию классов по смыслу и назначению принадлежащих им функций... понятное дело, приблизиться к исходному коду сможет только провидец (ясновидящий).

Словом, как бы там ни было, никогда не путайте экземпляр объекта с самим объектом и не забывайте, что объекты существуют только в исходном тексте и уничтожаются на стадии компиляции.

Мой адрес — не дом и не улица! Где живут структуры, массивы и объекты? Конечно же в памяти! А поконкретнее? Конкретнее, существуют три типа размещения: в *стеке* (автоматическая память), *сегменте данных* (статическая память) и *куче* (динамическая память). И каждый тип со своим «характером». Возьмем стек — выделение памяти неявное, фактически происходящее на этапе компиляции, причем гарантированно определяется только общий объем памяти, выделенный под все локальные переменные, а определить, сколько занимает каждая из них, невозможно в принципе. Не верите? А вот, скажем, пусть будет такой код: `char a1[13]; char a2[17]; char a3[23]`. Если компилятор выравнивает массивы по кратным адресам (а это делают многие компиляторы), то разница смещений ближайших друг к другу массивов может и не быть равна их размеру. Единственная надежда восстановить подлинный размер — найти в коде проверки выходы за границы массива (если они есть — их часто не бывает). Второе (самое неприятное) — если один из массивов не используется, а только объявляется, то неоптимизирующие компиляторы (и даже некоторые оптимизирующие!) могут тем не менее отвести для него стековое пространство. Он вплотную примкнет к предыдущему массиву, и гадай — то ли размер массива такой, то ли в его конец «вбухан» неиспользуемый массив! Ну, с массивами куда бы еще ни шло, а вот со структурами и объектами дела обстоят намного хуже. Никому и в голову не придет помещать в программу код, отслеживающий выход за пределы структуры (объекта). Такое невозможно в принципе (ну разве что программист слишком вольно работает с указателями)!

Ладно, оставим в стороне размер, перейдем к проблемам «разверстки» и поиску указателей. Как уже говорилось выше, если массив (объект, структура) объявляется в непосредственной области видимости единицы трансляции, он «всплывает» на этапе компиляции, и обращение к его членам происходит по фактическому смещению, а не по базовому указателю. К счастью, идентификацию

объектов облегчает наличие в них указателя на виртуальную таблицу, но ведь не факт, что любая таблица указателей на функции есть виртуальная таблица! Может, это просто массив указателей на функции, определенный самим программистом? Вообще-то при наличии опыта такие ситуации можно легко распознать (см. раздел «Идентификация виртуальных функций»), но все-таки они достаточно неприятны.

С объектами, расположенными в статической памяти, дела обстоят намного проще, в силу своей глобальности они имеют специальный флаг, предотвращающий повторный вызов конструктора (подробнее см. раздел «Идентификация конструктора и деструктора»), поэтому отличить экземпляр объекта, расположенный в сегменте данных, от структуры или массива становится очень легко. С определением его размера, правда, все те же неувязки.

Наконец, объекты (структуры, массивы), расположенные в куче, просто сказка для анализа! Отведение памяти осуществляется функцией, явно принимающей количество выделяемых байтов в качестве своего аргумента и возвращающей указатель, гарантированно указывающий на начало экземпляра объекта (структуры, массива). Радует и то, что обращение к элементам всегда происходит через базовый указатель, даже если объявление совершается в области видимости (иначе и быть не может — фактические адреса выделяемых блоков динамической памяти не известны на стадии компиляции).

Идентификация указателя **this**

Не все ли равно, о чем спрашивать, если ответа все равно не получишь, правда?

Льюис Кэрролл. Алиса в стране чудес

Указатель **this** — это настоящий золотой ключик или, если угодно, спасательный круг, позволяющий не утонуть в бурном океане ООП. Именно благодаря **this** возможно определять принадлежность вызываемой функции к тому или иному экземпляру объекта. Поскольку все неvirtуальные функции объекта вызываются непосредственно — по фактическому адресу, объект как бы расщепляется на составляющие его функции еще на стадии компиляции. Не будь указателей **this**, восстановить иерархию функций было бы принципиально невозможно!

Таким образом, правильная идентификация **this** очень важна. Единственная проблема: как отличить его от указателей на массивы и структуры? Ведь идентификация экземпляра объекта осуществляется по указателю **this** (если на выделенную память указывает **this**, это экземпляр объекта), однако сам **this** по определению — это указатель, ссылающийся на экземпляр объекта. Замкнутый круг! К счастью, есть одна лазейка... Код, манипулирующий указателем **this**, весьма специфичен, что и позволяет отличить **this** ото всех остальных указателей.

Вообще-то у каждого компилятора свой почерк, который настоятельно рекомендуется изучить, дизассемблируя собственные C++ программы, но существуют и универсальные рекомендации, приемлемые к большинству реализаций. По-

сколько `this` — это неявный аргумент каждой функции-члена класса, то логично отложить разговор о его идентификации до главы «Идентификация аргументов функций», здесь же мы дадим лишь краткую сводную таблицу, описывающую механизмы передачи `this` различными компиляторами.

Таблица 1. Механизм передачи указателя `this` в зависимости от реализации компилятора и типа функции

Компилятор	Тип функции				
	default	fastcall	cdecl	stdcall	PASCAL
Microsoft Visual C++	ECX		Через стек последним аргументом функции		Через стек первым аргументом
Borland C++	EAX				
WATCOM C					

Идентификация операторов `new` и `delete`

...нет ничего случайного. Самые свободные ассоциации являются самыми надежными.

Тезис классического психоанализа

Операторы `new` и `delete` транслируются компилятором в вызовы библиотечных функций, которые могут быть распознаны точно так, как и обычные библиотечные функции (см. раздел «Идентификация библиотечных функций»). Автоматически распознавать библиотечные функции умеет, в частности, IDA Pro, снимая эту заботу с плеч пользователя. Однако IDA Pro есть не у всех и далеко не всегда в нужный момент находится под рукой, да к тому же не все библиотечные функции она знает, а из тех, что знает, не всегда узнает `new` и `delete`... Словом, причин для их ручной идентификации существует предостаточно...

Реализация `new` и `delete` может быть любой, но Windows-компиляторы в большинстве своем редко реализуют функции работы с кучей самостоятельно. Зачем это? Намного проще обратиться к услугам операционной системы. Однако наивно ожидать увидеть вместо `new` вызов `HeapAlloc`, а вместо `delete` — `HeapFree`. Нет, компилятор не так прост! Разве он может отказать себе в удовольствии «вырезания матрешек»? Оператор `new` транслируется в функцию `new`, вызывающую для выделения памяти `malloc`, `malloc` же, в свою очередь, обращается к `heap_alloc` (или ее подобию — в зависимости от реализации библиотеки работы с памятью, см. раздел «Подходы к реализации кучи») — своеобразной «обертке» одноименной Win32 API-процедуры. Картина с освобождением памяти аналогична.

Углубляться в дебри вложенных вызовов слишком утомительно. Нельзя ли `new` и `delete` идентифицировать как-нибудь иначе, с меньшими трудозатратами и без большой головной боли? Разумеется, можно! Давайте вспомним все, что мы знаем о `new`:

- *new* принимает единственный аргумент — количество байтов выделяемой памяти, причем этот аргумент в подавляющем большинстве случаев вычисляется еще на стадии компиляции, т. е. является константой;
- если объект не содержит ни данных, ни виртуальных функций, его размер равен единице (минимальный блок памяти, выделяемый только для того, чтобы было на что указывать указателю *this*); отсюда будет очень много вызовов типа `PUSH 01\CALL xxx`, где *xxx* и есть адрес *new*! Вообще же типичный размер объектов составляет менее сотни байтов... ищите часто вызываемую функцию с аргументом-константой меньшей ста байтов;
- функция *new* — одна из самых популярных библиотечных функций, ищите функцию с «толпой» перекрестных ссылок;
- самое характерное: *new* возвращает указатель *this*, а *this* очень легко идентифицировать даже при беглом просмотре кода (см. раздел «Идентификация *this*»);
- возвращенный *new* результат всегда проверяется на равенство нулю, и, если он действительно равен нулю, конструктор (если он есть, см. раздел «Идентификация конструктора и деструктора») не вызывается;

«Родимых пятен» у *new* более чем достаточно для быстрой и надежной идентификации, тратить время на анализ ее кода совершенно ни к чему! Единственное, о чем следует помнить: *new* используется не только для создания новых экземпляров объектов, но и для выделения памяти под массивы (структуры) и изредка — под одиночные переменные (типа `int *x = new int`, что вообще маразм, но некоторые так делают). К счастью, отличить два этих способа очень просто — ни у массивов, ни у структур, ни у одиночных переменных нет указателя *this*!

Давайте для закрепления всего вышесказанного рассмотрим фрагмент кода, сгенерированного компилятором WATCOM (IDA PRO не распознает его «родного» оператора *new*):

Листинг 54

```
main_      proc near          ; CODE XREF: __CMain+40p
            push    10h
            call    __CHK
            push    ebx
            push    edx
            mov     eax, 4
            call    W?$_wnn_ui_pnv
            ; Это, как мы узнаем позднее, функция new. IDA вообще-то распознала ее имя, но
            ; чтобы узнать в этой "абракадабре" оператор выделения памяти, надо быть
            ; провидцем!
            ; Пока же обратим внимание, что она принимает один аргумент-константу,
            ; очень небольшую по значению, т. е. заведомо не являющуюся смещением
            ; (см. раздел "Идентификация констант и смещений").
            ; Передача аргумента через регистр ни о чем не говорит - Watcom так поступает
            ; со многими библиотечными функциями, напротив, другие компиляторы всегда
            ; заталкивают аргумент в стек...

            mov     edx, eax
```

```

test    eax, eax
; Проверка результата, возвращенного функцией на нулевое значение
; (что характерно для new)

jz      short loc_41002A
mov     dword ptr [eax], offset BASE_VTBL
; Ага, функция возвратила указатель, и по нему записывается указатель на
; виртуальную таблицу (или, по крайней мере, массив функций).
; EAX уже очень похож на this, но чтобы окончательно убедиться в этом,
; требуются дополнительные признаки...

loc_41002A:                                ; CODE XREF: main+1Aj
mov     ebx, [edx]
mov     eax, edx
call    dword ptr [ebx]
; Вот теперь можно не сомневаться, что EAX - указатель this, а этот код
; и есть вызов виртуальной функции!
; Следовательно, функция W?$_pnm_ui_pnv и есть new
;(а кто бы еще мог вернуть this?).

```

Сложнее идентифицировать *delete*. Каких-либо характерных признаков эта функция не имеет. Да, она принимает единственный аргумент — указатель на освобождаемый регион памяти, причем в подавляющем большинстве случаев этот указатель *this*. Но помимо нее *this* принимают десятки, если не сотни других функций! Правда, между ними существует одно тонкое различие — *delete* в большинстве случаев принимает указатель *this* через стек, а остальные функции — через регистр. К сожалению, некоторые компиляторы (тот же WATCOM — не к ночи он будет упомянут) передают многим библиотечным функциям аргументы через регистры, скрывая тем самым все различия! Еще *delete* ничего не возвращает, но мало ли функций поступают точно так же? Единственная зацепка — вызов *delete* следует за вызовом деструктора (если он есть), но, ввиду того что конструктор как раз и идентифицируется как функция, предшествующая *delete*, образуется замкнутый круг!

Ничего не остается, как анализировать ее содержимое, — *delete* рано или поздно вызывает *HeapFree* (хотя тут возможны и варианты: так, Borland содержит библиотеки, работающие с кучей на низком уровне и освобождающие память вызовом *VirtualFree*). К счастью, IDA Pro в большинстве случаев опознает *delete* и самостоятельно напрягаться не приходится.

Подходы к реализации кучи. В некоторых, между прочим достаточно многих, руководствах по программированию на C++ (например, Джеффри Рихтер «*Windows для профессионалов*») встречаются призывы всегда выделять память именно *new*, а не *malloc*, поскольку *new* опирается на эффективные средства управления памятью самой операционной системы, а *malloc* реализует собственный (и достаточно тормозной) менеджер кучи. Все это грубые натяжки! Стандарт вообще ничего не говорит о реализации кучи, и какая функция окажется эффективнее, наперед неизвестно. Все зависит от конкретных библиотек конкретного компилятора.

Рассмотрим, как происходит управление памятью в штатных библиотеках трех популярных компиляторов: *Microsoft Visual C++*, *Borland C++* и *Watcom C++*.

В *Microsoft Visual C++* и *malloc*, и *new* представляют собой переходники к одной и той же функции *__nh_malloc*, поэтому можно с одинаковым успехом пользоваться и той и другой. Сама же *__nh_malloc* вызывает *__heap_alloc*, в свою очередь вызывающую API-функцию Windows *HeapAlloc*. (Стоит отметить, что в *__heap_alloc* есть «хук» — возможность вызвать собственный менеджер куч, если по каким-то причинам системный менеджер будет недоступен, впрочем, в *Microsoft Visual C++ 6.0* от хука осталась одна лишь обертка, а собственный менеджер куч был исключен.)

Все не так в *Borland C++*! Во-первых, этот зверь напрямую работает с виртуальной памятью Windows, реализуя собственный менеджер кучи, основанный на функциях *VirtualAlloc / VirtualFree*. Профилировка показывает, что он серьезно проигрывает в производительности на Windows 2000 (другие системы не проверял), не говоря уже о том, что помещение лишнего кода в программу увеличивает ее размер. Во-вторых, *new* вызывает функцию *malloc*, причем вызывает не напрямую, а через несколько слоев «оберточного» кода! Поэтому, вопреки всем рекомендациям, под *Borland C++* вызов *malloc* эффективнее, чем *new*!

Товарищ *Watcom* (во всяком случае, его одиннадцатая версия — последняя, до которой мне удалось дотянуться) реализует *new* и *malloc* практически идентичным образом — обе они ссылаются на *_nmalloc* — очень «толстую» обертку от *LocalAlloc*. Да, да, 16-разрядной функции Windows, которая сама является переходником к *HeapAlloc*!

Таким образом, Джеффри Рихтер лопухнулся по полной программе: ни в одном из популярных компиляторов *new* не быстрее *malloc*, а вот наоборот — таки да! Уж не знаю, какой он такой редкоземельный компилятор имел в виду (точнее, не сам компилятор, а библиотеки, поставляемые вместе с ним, но это не суть важно), или, скорее всего, просто писал не думая. Отсюда мораль — все умозаклочения, прежде чем переносить на бумагу, необходимо тщательно проверять.

Идентификация библиотечных функций

Сегодня целый день идет снег. Он падает, тихо кружась. Ты помнишь? Тогда тоже все было засыпано снегом — это был снег наших встреч. Он лежал перед нами, белый-белый, как чистый лист бумаги, и мне казалось, что мы напишем на этом листе повесть нашей любви.

Снегопад «Пламя»

Читая текст программы, написанный на языке высокого уровня, мы только в исключительных случаях изучаем реализацию стандартных библиотечных функ-

ций, таких, например, как *printf*. Да и зачем? Ее назначение известно и без того, а если и есть какие неясности — всегда можно заглянуть в описание...

Анализ дизассемблерного листинга — дело другое. Имена функций, за редкими исключениями, в нем отсутствуют, и определить, *printf* это или что-то другое, «на взгляд» невозможно. Приходится вникать в алгоритм... Легко сказать — вникать! Та же *printf* представляет собой сложный интерпретатор строки спецификаторов — с ходу в нем не разберешься! А ведь есть и более монструозные функции. Самое обидное — алгоритм их работы не имеет никакого отношения к анализу исследуемой программы. Тот же *pew* может и выделять память из Windows-кучи, и реализовывать собственный менеджер, но нам-то от этого что? Достаточно знать, что это именно *pew*, т. е. функция выделения памяти, а не *free* или скажем, *forep*.

Доля библиотечных функций в программе в среднем составляет от пятидесяти до девяноста процентов. Особенно она велика у программ, составленных в визуальных средах разработки, использующих автоматическую генерацию кода (например, Microsoft Visual C++, DELPHI). Причем библиотечные функции подчас намного сложнее и запутаннее тривиального кода самой программы. Обидно, львиная доля усилий по анализу вылетает впустую... Как бы оптимизировать этот процесс?

Уникальная способность IDA различать стандартные библиотечные функции множества компиляторов выгодно отличает ее от большинства других дизассемблеров, этого делать не умеющих. К сожалению, IDA (как и все, созданное человеком) далека от идеала: каким бы обширным ни был список поддерживаемых библиотек, конкретные версии конкретных поставщиков или моделей памяти могут отсутствовать. И даже из тех библиотек, что ей известны, распознаются не все функции (о причинах будет рассказано чуть ниже). Впрочем, нераспознанная функция — это полбеда, неправильно распознанная функция — много хуже, ибо это приводит к ошибкам (иногда трудноуловимым) анализа исследуемой программы или ставит исследователя в глухой тупик. Например, вызывается *forep*, и возвращенный ею результат спустя некоторое время передается *free* — с одной стороны: почему бы и нет? Ведь *forep* возвращает указатель на структуру *FILE*, а *free* ее и удаляет. А если *free* никакой не *free*, а, скажем, *fseek*? Пропустив операцию позиционирования, мы не сможем правильно восстановить структуру файла, с которым работает программа.

Распознать ошибки IDA будет легче, если представлять, как именно она выполняет распознавание. Многие почему-то считают, что здесь задействован тривиальный подсчет CRC (контрольной суммы). Что ж, подсчет CRC — заманчивый алгоритм, но, увы, для решения данной задачи он не пригоден. Основной камень преткновения — наличие непостоянных фрагментов, а именно **перемещаемых элементов** (подробнее см. раздел «Шаг четвертый. Знакомство с отладчиком :: Бряк на оригинальный пароль»). И хотя при подсчете CRC перемещаемые элементы можно элементарно игнорировать (не забывая проделывать ту же операцию и в идентифицируемой функции), разработчик IDA пошел другим, более запутанным и витиеватым, но и более производительным путем.

Ключевая идея заключается в том, что незачем тратить время на вычисление CRC, для предварительной идентификации функции вполне сойдет и тривиальное

посимвольное сравнение, за вычетом перемещаемых элементов (они игнорируются и в сравнении не участвуют). Точнее говоря, не сравнение, а поиск заданной последовательности байтов в эталонной базе, организованной в виде двоичного дерева. Время двоичного поиска, как известно, пропорционально логарифму количества записей в базе. Здравый смысл подсказывает, что длина шаблона (иначе говоря, сигнатуры, т. е. сравниваемой последовательности) должна быть достаточной для однозначной идентификации функции. Однако разработчик IDA по непонятным для автора причинам решил ограничиться только первыми тридцатью двумя байтами, что (особенно с учетом вычета пролога, который у всех функций практически одинаков) довольно мало.

И верно! Достаточно многие функции попадают на один и тот же лист дерева, возникает **коллизия** — неоднозначность отождествления. Для разрешения ситуации у всех «коллизионных» функций подсчитывается CRC16 с тридцать второго байта до первого перемещаемого элемента и сравнивается с CRC16 эталонных функций. Чаше всего это срабатывает, но если первый перемещаемый элемент окажется расположенным слишком близко к тридцать второму байту, последовательность подсчета контрольной суммы окажется слишком короткой, а то и вовсе равной нулю (может же быть тридцать второй байт перемещаемым элементом, почему бы и нет?). В случае повторной коллизии находим в функциях байт, в котором все они отличаются, и запоминаем его смещение в базе.

Все это (да простит автора разработчик IDA!) напоминает следующий анекдот: *поймали туземцы немца, американца и хохла и говорят им: мол, или откупайтесь чем-нибудь, или съедим. На откуп предлагается: миллион долларов (только не спрашивайте автора, зачем туземцам миллион долларов, — может, костер жесть), сто щелбанов или съесть мешок соли. Ну, американец достает сотовый, звонит кому-то... Приплывает катер с миллионом долларов, и американца благополучно отпускают. Немец в это время героически съедает мешок соли, и его полумертвого спускают на воду. Хохол же ел соль, ел-ел, две трети съел, не выдержал и говорит: а, ладно, черти, бейте щелбаны. Бьет вождь его и только девяносто ударов отщелкал, хохол не выдержал и говорит: да нате миллион, подавитесь!* Так и с IDA, посимвольное сравнение не до конца, а только тридцати двух байтов, подсчет CRC не для всей функции, а сколько как случай на душу положит, наконец, последний ключевой байт — и тот-то «ключевой», да не совсем. Дело в том, что многие функции совпадают байт в байт, но совершенно различны по названию и назначению. Не верите? Тогда как вам понравится следующее:

Листинг 55

read:	write:
push ebp	push ebp
mov ebp,esp	mov ebp,esp
call _read	call _write
pop ebp	pop ebp
ret	ret

Тут без анализа перемещаемых элементов никак не обойтись! Причем это не какой-то специально надуманный пример — подобных функций очень много. В частности, библиотеки от Borland ими так и кишат. Неудивительно, что IDA часто «спотыкается» и впадает в грубые ошибки. Взять, к примеру, следующую функцию:

```
void demo(void)
{
    printf("DERIVED\n");
};
```

Даже последняя на момент написания этой книги версия IDA 4.17 ошибается, обзывая ее `__pure_error`:

```
CODE:004010F3 __pure_error_ proc near                ; DATA XREF: DATA:004080BC↓o
CODE:004010F3         push     ebp
CODE:004010F4         mov      ebp, esp
CODE:004010F6         push     offset aDerived ; format
CODE:004010FB         call    _printf
CODE:00401100         pop      ecx
CODE:00401101         pop      ebp
CODE:00401102         retn
CODE:00401102 __pure_error_ endp
```

Стоит ли говорить, какие неприятные последствия для анализа эта ошибка может иметь? Бывает, сидишь, тупо уставившись в листинг дизассемблера, и никак не можешь понять: что же этот фрагмент делает? И только потом обнаруживаешь — одна или несколько функций опознаны неправильно!

Для уменьшения количества ошибок IDA пытается по стартовому коду распознать компилятор, подгружая только библиотеку его сигнатур. Из этого следует, что «ослепить» IDA очень просто — достаточно слегка видоизменить стартовый код. Поскольку он, по обыкновению, поставляется вместе с компилятором в исходных текстах, сделать это будет нетрудно. Впрочем, хватит и изменения одного байта в начале `startup`-функции. И все, хакер скинет ласты! К счастью, в IDA предусмотрена возможность ручной загрузки базы сигнатур (`FILE\Load file\FLIRT signature file`), но попробуй-ка вручную определить, сигнатуры какой именно версии библиотеки требуется загружать! Наугад — слишком долго... Хорошо, если удастся визуально опознать компилятор (опытным исследователям это обычно удается, т. к. каждый из них имеет свой уникальный «почерк»). К тому же существует принципиальная возможность использования библиотек из поставки одного компилятора в программе, скомпилированной другим компилятором. Словом, будьте готовы к тому, что в один прекрасный момент вы можете столкнуться с необходимостью самостоятельно опознавать библиотечные функции. Решение этой задачи состоит из трех этапов. Первое — определение самого факта «библиотечности» функции, второе — определение происхождения библиотеки и третье — идентификации функции по этой библиотеке.

Используя тот факт, что линкер обычно располагает функции в порядке перечисления `obj`-модулей и библиотек, а большинство программистов указывают сначала собственные `obj`-модули, а библиотеки — потом (кстати, так же поступают и компиляторы, самостоятельно вызывающие линкер по окончании своей работы),

можно заключить: библиотечные функции помещаются в конце программы, а собственно ее код — в начале. Конечно, из этого правила есть исключения, но все же срабатывает оно достаточно часто.



Рис. 14. Структура pkzip.exe. Обратите внимание, все библиотечные функции (белые) в одном месте — в конце сегмента кода перед началом сегмента данных

Рассмотрим, к примеру, структуру общеизвестной программы pkzip.exe на диаграмме, построенной IDA 4.17, видно, что все библиотечные функции сосредоточены в одном месте — в конце сегмента кода, вплотную примыкая к сегменту данных. Самое интересное — start-up функция в подавляющем большинстве случаев расположена в самом начале региона библиотечных функций или находится в непосредственной близости от него. Найти же саму start-up не проблема — она совпадает с точкой входа в файл!

Таким образом, можно с высокой долей уверенности утверждать, что все функции, расположенные «ниже» Start-up (т. е. в более старших адресах), — библиотечные. Посмотрите, распознала их IDA или переложила эту заботу на вас? Грубо — возможны две ситуации: вообще никакие функции не распознаны и не распознана только часть функций.

Если не распознана ни одна функция, скорее всего, IDA не сумела опознать компилятор или использовались неизвестные ей версии библиотек. Техника распознавания компиляторов — разговор особый, а вот распознавание версий библиотек — это то, чем мы сейчас и займемся.

Прежде всего, многие из них содержат копирайты с названием имени производителя и версии библиотеки, просто поищите текстовые строки в бинарном файле. Если их нет, не беда — ищем любые другие текстовые строки (как правило, сообщения об ошибках) и простым контекстным поиском пытаемся найти их во всех библиотеках, до которых удастся дотянуться (хакер должен иметь большую библиотеку компиляторов и библиотек на своем жестком диске). Возможные варианты: никаких других текстовых строк вообще нет; строки есть, но они не уникальны — обнаруживаются во многих библиотеках; наконец, искомый фрагмент нигде не обнаружен. В первых двух случаях следует выделить из одной (нескольких) библиотечных функций характерную последовательность байтов, не содержащую перемещаемых элементов, и вновь попытаться отыскать ее во всех доступных вам библиотеках. Если же это не поможет, то увы, искомой библиотеки у вас в наличии и нет и положение — ласты.

Ласты, да не совсем! Конечно, автоматически восстановить имена функций уже не удастся, но надежда на быстрое выяснение назначения функций все же есть. Имена API-функций Windows, вызываемые из библиотек, позволяют иденти-

фицировать, по крайней мере, категорию библиотеки (например, работа с файлами, памятью, графикой и т. д.). Математические же функции, по обыкновению, богаты командами сопроцессора.

Дизассемблирование очень похоже на разгадывание кроссворда (хотя не факт, что хакеры любят разгадывать кроссворды) — неизвестные слова угадываются за счет известных. Применительно к данной ситуации в некоторых контекстах название функции вытекает из ее использования. Например, запрашиваем у пользователя пароль, передаем ее функции *X* вместе с эталонным паролем, если результат завершения нулевой — пишем «пароль ОК», и соответственно наоборот. Не подсказывает ли ваша интуиция, что функция *X* не что иное, как *strcmp*? Конечно, это простейший случай, но, по любому, столкнувшись с незнакомой подпрограммой, не спешите впадать в отчаяние, приходя в ужас от ее «монстроузости», просмотрите все вхождения, обращая внимания, **кто** вызывает ее, **когда** и **сколько раз**.

Статистический анализ проливает свет на очень многое (функции, как и буквы алфавита, встречаются каждая со своей частотой), а контекстная зависимость дает пищу для размышлений. Так, функция чтения из файла не может предшествовать функции открытия!

Другие зацепки: аргументы и константы. Ну, с аргументами все более или менее ясно. Если функция получает строку, то это, очевидно, функция из библиотеки работы со строками, а если вещественное значение — возможно, функция математической библиотеки. Количество и тип аргументов (если их учитывать) весьма сужают круг возможных кандидатов. С константами же еще проще — очень многие функции принимают в качестве аргумента флаг, допускающий ограниченное количество значений. За исключением битовых флагов, которые все похожи друг на друга, довольно часто встречаются уникальные значения, пускай не однозначно идентифицирующие функцию, но все равно сужающие круг «подозреваемых». Да и сами функции могут содержать характерные константы, скажем, встретив стандартный полином для подсчета CRC, можно быть уверенным, что «подследственная» вычисляет контрольную сумму...

Мне могут возразить: мол, все это частности. Возможно. Но, опознав часть функций, назначения остальных можно вычислить «от противного» и, уж по крайней мере, понять, что это за библиотека такая и где ее искать.

Напоследок, идентификацию алгоритмов (т. е. назначения функции) очень сильно облегчает знание этих самих алгоритмов. В частности, код, осуществляющий LZ-сжатие (распаковку), настолько характерен, что узнается с беглого взгляда, достаточно знать этот механизм упаковки. Напротив, если не иметь о нем никакого представления, ох, и нелегко же будет анализировать программу! Зачем изобретать колесо, если можно взять уже готовое? Хотя и бытует мнение, что хакер в первую очередь хакер, а уж потом программист (да и зачем ему уметь программировать?), в жизни все наоборот — программист, не умеющий программировать, проживет — вокруг уйма библиотек, воткни — и заработает! Хакеру же знание информатики необходимо, без этого далеко не уплывешь (разумеется, отломать серийный номер можно и без высшей математики).

Понятное дело, библиотеки как раз на то и создавались, чтобы избавить разработчиков от необходимости вникать в те предметные области, без которых им и так хорошо. Увы, у исследователей программ нет простых путей, приходится думать и руками, и головой, и даже... пятой точкой опоры вкупе со спинным мозгом, только так и дизассемблируются серьезные программы. Бывает, готовое решение приходит в поезде или во сне...

Анализ библиотечных функций — это самый сложный аспект дизассемблирования, и просто замечательно, когда есть возможность идентифицировать их имена по сигнатурам.

Идентификация аргументов функций

То, что пугает зверя, не пугает человека.

Ф. Херберт. Ловец душ

Идентификация аргументов функций — ключевое звено в исследовании дизассемблерных программ, поэтому приготовьтесь, к тому, что эта глава будет длинной и, возможно, скучной, но ничего не поделаешь — хакерство требует жертв!

Существует три способа передачи аргументов функции: через **стек**, **регистры** и **комбинированный** (через стек и регистры одновременно). К этому списку вплотную примыкает и неявная передача аргументов через глобальные переменные, описание которой вынесено в отдельную главу «Идентификация глобальных переменных».

Сами же аргументы могут передаваться либо по **значению**, либо по **ссылке**. В первом случае функции передается копия соответствующей переменной, а во втором — **указатель** на саму переменную.

Соглашения о передаче параметров. Для успешной совместной работы вызывающая функция должна не только знать прототип вызываемой, но и «договориться» с ней о способе передачи аргументов: по ссылке или по значению, через регистры или через стек? Если через регистры — оговорить, какой аргумент в какой регистр помещен, а если через стек — определить порядок занесения аргументов и выбрать «ответственного» за очистку стека от аргументов после завершения вызываемой функции.

Неоднозначность механизма передачи аргументов — одна из причин несовместимости различных компиляторов. Казалось, почему бы не заставить всех производителей компиляторов придерживаться какой-то одной схемы? Увы, это решение принесет больше проблем, чем решит.

Каждый механизм имеет свои достоинства и недостатки и, что еще хуже, тесно связан с самим языком. В частности, «Сишные» вольности в отношении соблюдения прототипов функций возможны именно потому, что аргументы из стека выталкивает не вызываемая, а вызывающая функция, которая наверняка помнит, что она передавала. Например, функции `main` передаются два аргумента — количество ключей командной строки и указатель на содержащий их массив. Однако

если программа не работает с командной строкой (или получает ключ каким-то иным путем), прототип `main` может быть объявлен и так: `main()`.

На Паскале подобная выходка привела бы либо к ошибке компиляции, либо к краху программы, так как в нем стек очищает непосредственно вызываемая функция и, если она этого не сделает (или сделает неправильно, вытолкнув не то же самое количество машинных слов, которое ей было передано), стек окажется несбалансированным и все рухнет (точнее, у материнской функции «слетит» вся адресация локальных переменных, а вместо адреса возврата в стеке окажется, что глюк на душу положит).

Минусом «Сишного» решения является незначительное увеличение размера генерируемого кода, ведь после каждого вызова функции приходится вставлять машинную команду (и порой не одну) для выталкивания аргументов из стека, а у Паскаля эта команда внесена непосредственно в саму функцию и потому встречается в программе один-единственный раз.

Не найдя золотой середины, разработчики компиляторов решили использовать все возможные механизмы передачи данных, а чтобы справиться с проблемой совместимости, стандартизировали каждый из механизмов, введя ряд *соглашений*.

С-соглашение (обозначаемое `__cdecl`) предписывает засылать аргументы в стек справа налево в порядке их объявления, а очистку стека возлагает на плечи вызывающей функции. Имена функций, следующих С-соглашению, предваряются символом подчеркика «`_`», автоматически вставляемого компилятором. Указатель `this` (в С++ программах) передается через стек последним по счету аргументом.

Паскаль-соглашение (обозначаемое `PASCAL`) (в настоящее время ключевое слово `PASCAL` считается устаревшим и выходит из употребления, вместо него можно использовать аналогичное соглашение `WINAPI`) предписывает засылать аргументы в стек слева направо в порядке их объявления и возлагает очистку стека на саму вызывающую функцию.

Стандартное соглашение (обозначаемое `__stdcall`) является гибридом С-и Паскаль-соглашений. Аргументы засылаются в стек справа налево, но очищает стек сама вызываемая функция. Имена функций, следующих стандартному соглашению, предваряются символом прочерка «`_`», а заканчиваются суффиксом `@`, за которым следует количество байтов, передаваемых функции. Указатель `this` (в С++ программах) передается через стек последним по счету аргументом.

Соглашение быстрого вызова. Предписывает передавать аргументы через регистры. Компиляторы от Microsoft и Borland поддерживают ключевое слово `__fastcall`, но интерпретируют его по-разному, а WATCOM С++ вообще не понимает ключевого слова `__fastcall`, но имеет в «арсенале» своего лексикона специальную прагму `aux`, позволяющую вручную выбрать регистры для передачи аргументов (подробнее см. раздел «Соглашения о быстрых вызовах — `fastcall`»). Имена функций, следующих соглашению `__fastcall`, предваряются символом `@`, автоматически вставляемым компилятором.

Соглашение по умолчанию. Если явное объявление типа вызова отсутствует, компилятор обычно использует собственные соглашения, выбирая их по своему усмотрению. Наибольшему влиянию подвергается указатель `this`, большинство компиляторов при вызове по умолчанию передают его через регистр. У Microsoft это ECX, у Borland — EAX, у WATCOM либо EAX, либо EDX, либо и то и другое разом. Остальные аргументы также могут передаваться через регистры, если оптимизатор посчитает, что так будет лучше. Механизм передачи и логика выборки аргументов у всех разная и наперед непредсказуемая, разбирайтесь по ситуации.

Цели и задачи. При исследовании функции перед исследователем стоят следующие задачи: определить, *какое соглашение используется для вызова*; подсчитать *количество аргументов*, передаваемых функции (и/или используемых функцией); наконец, выяснить *тип и назначение* самих аргументов. Начнем?

Тип соглашения грубо идентифицируется по способу вычистки стека. Если его очищает вызываемая функция, мы имеем дело с `cdecl`, в противном случае либо с `stdcall`, либо с `PASCAL`. Такая неопределенность в отождествлении вызвана тем, что подлинный прототип функции неизвестен и, стало быть, порядок занесения аргументов в стек определить невозможно. Единственная зацепка: зная компилятор и предполагая, что программист использовал тип вызовов по умолчанию, можно уточнить тип вызова функции. Однако в программах под Windows широко используются оба типа вызовов: и `PASCAL` (он же `WINAPI`), и `stdcall`, поэтому неопределенность по-прежнему остается. Впрочем, порядок передачи аргументов ничего не меняет: имея в наличии и вызывающую, и вызываемую функцию между передаваемыми и принимаемыми аргументами, всегда можно установить взаимную однозначность. Или, проще говоря, если действительный порядок передачи аргументов известен (а он и будет известен, см. вызывающую функцию), то знать очередность расположения аргументов в прототипе функции уже ни к чему.

Другое дело — библиотечные функции, прототип которых известен. Зная порядок занесения аргументов в стек, по прототипу можно автоматически восстановить тип и назначение аргументов!

Определение количества и типа передачи аргументов. Как уже было сказано выше, аргументы могут передаваться либо через стек, либо через регистры, либо и через стек, и через регистры сразу, а также неявно через глобальные переменные.

Если бы стек использовался только для передачи аргументов, подсчитать их количество было бы относительно легко. Увы, стек активно используется и для временного хранения регистров с данными. Поэтому, встретив инструкцию заталкивания `PUSH`, не торопитесь идентифицировать ее как аргумент. Узнать количество байтов, переданных функции в качестве аргументов, невозможно, но достаточно легко определить количество байтов, выталкиваемых из стека после завершения функции!

Если функция следует соглашению `stdcall` (или `PASCAL`), она наверняка очищает стек командой `RET n`, где `n` и есть искомое значение в байтах. Хуже с `cdecl`-функциями. В общем случае за их вызовом следует инструкция `ADD ESP, n`, где `n` —

искомое значение в байтах, но возможны и вариации — *отложенная очистка стека* или выталкивание аргументов в какой-нибудь свободный регистр. Впрочем, отложим головоломки оптимизации на потом, пока ограничившись лишь кругом неоптимизирующих компиляторов.

Логично предположить, что количество занесенных в стек байтов равно количеству выталкиваемых, иначе после завершения функции стек окажется несбалансированным и программа рухнет (о том, что оптимизирующие компиляторы допускают дисбаланс стека на некотором участке, мы помним, но поговорим об этом потом). Отсюда: количество аргументов равно количеству переданных байтов, деленному на размер машинного слова. Под машинным словом понимается не только два байта, но и размер операндов по умолчанию, в 32-разрядном режиме машинное слово равно четырем байтам. Верно ли это? Нет! Далеко не всякий аргумент занимает ровно один элемент стека. Взять тот же тип `double`, отъедающий восемь байтов, или символьную строку, переданную не по ссылке, а по непосредственному значению, она «скушает» столько байтов, сколько захочет... К тому же засылаться в стек строка (как и структура данных, массив, объект) может не командой `PUSH`, а с помощью `MOVS`! (Кстати, наличие `MOVS` — явное свидетельство передачи аргумента по значению.)

Если я не успел окончательно вас запутать, то попробуем разложить по полочкам тот кавардак, что образовался в нашей голове. Итак, анализом кода вызываемой функции установить количество переданных через стек аргументов невозможно. Даже количество переданных байтов определяется весьма неуверенно. С типом передачи полный мрак. Позже (см. раздел «Идентификация констант и смещений») мы к этому еще вернемся, а пока вот пример: `PUSH 0x404040/CALL MyFunc: 0x404040` — что это: аргумент, передаваемый по значению (т. е. константа `0x404040`), или указатель на нечто, расположенное по смещению `0x404040` (и тогда, стало быть, передача происходит по ссылке)? Определить невозможно, не правда ли?

Но не волнуйтесь, нам не пришли кранты — мы еще повоюем! Большую часть проблем решает анализ вызываемой функции. Выяснив, как она манипулирует переданными ей аргументами, мы установим и их тип, и количество! Для этого нам придется познакомиться с адресацией аргументов в стеке, но, прежде чем приступить к работе, рассмотрим в качестве небольшой разминки следующий пример:

Листинг 56. Демонстрация механизма передачи аргументов

```
#include <stdio.h>
#include <string.h>

struct XT{
    char s0[20];
    int x; };

void MyFunc(double a, struct XT xt)
{
    printf("%f,%x,%s\n", a, xt.x, &xt.s0[0]);
}
```

```
main()
{
    struct XT xt;
    strcpy(&xt.s0[0], "Hello,World!");
    xt.x=0x777;
    MyFunc(6.66, xt);
}
```

Результат его компиляции компилятором Microsoft Visual C++ с настройками по умолчанию выглядит так:

Листинг 57

```
main                proc near                ; CODE XREF: start+AFp
var_18              = byte ptr -18h
var_4               = dword ptr -4

    push    ebp
    mov     ebp, esp
    sub     esp, 18h
    ; Первый PUSH явно относится к прологу функции, а не к передаваемым аргументам

    push    esi
    push    edi
    ; Отсутствие явной инициализации регистров говорит о том, что, скорее всего,
    ; они просто сохраняются в стеке, а не передаются как аргументы,
    ; однако если данной функции аргументы передавались не только через стек,
    ; но и через регистры ESI и EDI, то их засылка в стек вполне может
    ; преследовать цель передачи аргументов следующей функции

    push    offset aHelloWorld ; "Hello,World!"
    ; Ага, а вот здесь явно имеет место передача аргумента - указателя на строку
    ; (строго говоря, предположительно имеет место, см. "Идентификация констант")
    ; Хотя теоретически возможно временное сохранение константы в стеке для ее
    ; последующего выталкивания в какой-нибудь регистр или же непосредственному
    ; обращению к стеку, ни один из известных мне компиляторов не способен на такие
    ; хитрости, и засылка константы в стек всегда является передаваемым аргументом

    lea     eax, [ebp+var_18]
    ; в EAX заносится указатель на локальный буфер

    push    eax
    ; EAX (указатель на локальный буфер) сохраняется в стеке.
    ; Поскольку ряд аргументов непрерывен, то после распознавания первого аргумента
    ; можно не сомневаться, что все последующие заносы чего бы то ни было в стек -
    ; так же аргументы

    call    strcpy
    ; Прототип функции strcpy(char *, char *) не позволяет определить порядок
    ; занесения аргументов, однако поскольку все библиотечные Си-функции
    ; следует соглашению cdecl, то аргументы заносятся справа налево
    ; и исходный код выглядел так: strcpy(&buff[0], "Hello,World!")
    ; Но, может быть, программист использовал преобразование, скажем, в stdcall?
    ; Крайне маловероятно, для этого пришлось бы перекомпилировать и саму
    ; strcpy - иначе откуда бы она узнала, что порядок занесения аргументов
```

```
; изменился? Хотя обычно стандартные библиотеки поставляются с исходными
; текстами, их перекомпиляцией практически никто и никогда не занимается

add     esp, 8
; Вытаскиваем 8 байтов из стека. Из этого мы заключаем, что функции передавалось
; два машинных слова аргументов и, следовательно, PUSH ESI и PUSH EDI не были
; аргументами функции!

mov     [ebp+var_4], 777h
; Заносим в локальную переменную константу 0x777. Это явно константа, а не
; указатель, так как у Windows в этой области памяти не могут храниться никакие
; пользовательские данные

sub     esp, 18h
; Резервирование памяти для временной переменной. Временные переменные,
; в частности, создаются при передаче аргументов по значению, поэтому
; будем готовы к тому, что следующий "товарищ" - аргумент
; (см. "Идентификация регистровых и временных переменных")

mov     ecx, 6
; Заносим в ECX константу 0x6. Пока еще неизвестно зачем.

lea     esi, [ebp+var_18]
; Загружаем в ESI указатель на локальный буфер, содержащий скопированную
; строку "Hello, World!"

mov     edi, esp
; Копируем в EDI указатель на вершину стека

rep     movsd
; Вот она, передача строки по значению. Строка целиком копируется в стек,
; отъедая от него 6*4 байтов
; (6 - значение счетчика ECX, 4 - размер двойного слова - movsd),
; следовательно, этот аргумент занимает 20 (0x14) байтов стекового пространства -
; эта цифра нам пригодится при определении количества аргументов по количеству
; вытаскиваемых байтов.
; В стек копируются данные с [ebp+var_18] до [ebp+var_18-0x14], т. е.
; с var_18 до var_4. Но ведь в var_4 содержится константа 0x777!
; Следовательно, она будет передана функции вместе со строкой.
; Это позволяет нам воссоздать исходную структуру:
; struct x{
;     char s0[20]
;     int x
; }
; Да, функции, выходит, передается структура, а не одна строка!

push    401AA3D7h
push    0A3D70A4h
; Заносим в стек еще два аргумента. Впрочем, почему именно два?
; Это вполне может быть и один аргумент типа int64 или double.
; Определить, какой именно, по коду вызывающей функции не представляется
; возможным

call    MyFunc
; Вызов MyFunc. Прототип функции установить, увы, не удастся... Ясно только,
; что первый (слева? справа?) аргумент - структура, а за ним идут либо два int,
```

```

; либо один int64 или double.
; Уточнить ситуацию позволяет анализ вызываемой функции, но мы это отложим
; на потом – до того как изучим адресацию аргументов в стеке.
; Пока же придется прибывать в полной неопределенности

add     esp, 20h
; Вытаскиваем 0x20 байтов. Поскольку 20 байтов (0x14) приходится на структуру
; и 8 байтов – на два следующих аргумента, получаем 0x14+0x8=0x20, что
; и требовалось доказать.

pop     edi
pop     esi
mov     esp, ebp
pop     ebp
ret     0
sub_401022  endp

aHelloWorld  db 'Hello,World!',0 ; DATA XREF: sub_401022+80
               align 4

```

Результат компиляции компилятором Borland C++ будет несколько иным и довольно поучительным. Рассмотрим и его:

Листинг 58

```

_main      proc near          ; DATA XREF: DATA:00407044o

var_18     = byte ptr -18h
var_4      = dword ptr -4

    push    ebp
    mov     ebp, esp
    add     esp, 0FFFFFFE8h
; Ага! Это сложение со знаком минус. Жмем в IDA <-> и получаем ADD ESP, -18h

    push    esi
    push    edi
; Пока все идет, как в предыдущем случае

    mov     esi, offset aHelloWorld ; "Hello,World!"
; А вот тут начинаются различия! Вызов strcpy как корова языком слизала –
; нету его, и все! Причем компилятор даже не развернул функцию,
; подставляя ее на место вызова, а просто исключил сам вызов!

    lea     edi, [ebp+var_18]
; Заносим в EDI указатель на локальный буфер

    mov     eax, edi
; Заносим тот же самый указатель в EAX

    mov     ecx, 3
    repe    movsd
    movsb
; Обратите внимание, копируется 4*3+1=13 байтов. Тринадцать, а вовсе не
; двадцать, как следует из объявления структуры. Это компилятор так
; оптимизировал код, копируя в буфер лишь саму строку и игнорируя ее
; неинициализированный "хвост"

```

```

mov     [ebp+var_4], 777h
; Заносим в локальную переменную константу 0x777

push    401AA3D7h
push    0A3D70A4h
; Аналогично. Мы не можем определить, чем являются эти два числа -
; одним или двумя аргументами.

lea     ecx, [ebp+var_18]
; Заносим в ECX указатель на начало строки

mov     edx, 5
; Заносим в EDX константу 5 (пока не понятно зачем)

loc_4010D3:                                ; CODE XREF: _main+37j
push    dword ptr [ecx+edx*4]
; Ой, что это за кошмарный код? Давайте подумаем, начав раскручивать его
; с самого конца. Прежде всего, чему равно ECX+EDX*4? ECX - указатель на
; буфер, и с этим все ясно, а вот EDX*4 == 5*4 == 20.
; Ага, значит, мы получаем указатель не на начало строки, а на конец, вернее,
; даже не на конец, а на переменную ebp+var_4 (0x18-0x14=0x4).
; Подумаем: если это указатель на саму var_4, то зачем его вычислять таким
; закрученным макаром? Вероятнее всего, мы имеем дело со структурой...
; Далее, смотрите, команда push засылает в стек двойное слово,
; хранящееся по этому указателю

dec     edx
; Уменьшаем EDX... Вы уже почувствовали, что мы имеем дело с циклом?

jns     short loc_4010D3
; Вот этот переход, срабатывающий, пока EDX не отрицательное число,
; подтверждает наше предположение о цикле.
; Да, такой вот извращенной конструкцией Borland передает аргумент - структуру
; функции по значению!

call    MyFunc
; Вызов функции... смотрите, нет очистки стека! Да, это последняя вызываемая
; функция, и очистки стека не требуется - Borland ее и не выполняет...

xor     eax, eax
; Обнуление результата, возвращенного функцией. Borland так поступает с void-
; функциями - они у него всегда возвращают ноль,
; точнее, не они возвращают, а помещенный за их вызовом код, обнуления EAX

pop     edi
pop     esi
; Восстанавливаем ранее сохраненные регистры EDI и ESI

mov     esp, ebp
; Восстанавливаем ESI - вот почему стек не очищался после вызова последней
; функции!

pop     ebp
retn

_main   endp

```

Обратите внимание — по умолчанию Microsoft C++ передает аргументы справа налево, а Borland C++ — слева направо! Среди стандартных типов вызова нет такого, который, передавая аргументы слева направо, поручал бы очистку стека вызывающей функции! Выходит, что Borland C++ использует свой собственный, ни с чем не совместимый тип вызова!

Адресация аргументов в стеке. Базовая концепция стека включает в себя лишь две операции — занесение элемента в стек и снятие последнего занесенного элемента со стека. Доступ к произвольному элементу — это что-то новенькое! Однако такое отступление от канонов существенно увеличивает скорость работы. Если нужен, скажем, третий по счету элемент, почему бы не вытащить из стека напрямую, не снимая первые два? Стек это не только «стопка», как учат популярные учебники по программированию, но еще и массив. А раз так, то, зная положение указателя вершины стека (а не знать его мы не можем, иначе куда прикажете класть очередной элемент?), и размер элементов, мы сможем вычислить смещение любого из элементов, после чего не составит никакого труда его прочитать.

Попутно отметим один из недостатков стека: как и любой другой гомогенный массив, стек может хранить данные лишь одного типа, например, двойные слова. Если же требуется занести один байт (скажем, аргумент типа `char`), то приходится расширять его до двойного слова и заносить его целиком. Аналогично, если аргумент занимает четыре слова (*double*, *int64*), на его передачу расходуется два стековых элемента!

Помимо передачи аргументов стек используется и для сохранения адреса возврата из функции, что требует в зависимости от типа вызова функции (ближнего или дальнего) от одного до двух элементов. Ближний (*near*) вызов действует в рамках одного сегмента — в этом случае достаточно сохранить лишь смещение команды, следующей за инструкцией `CALL`. Если же вызываемая функция находится в одном сегменте, а вызываемая в другом, то помимо смещения приходится запоминать и сам сегмент, чтобы знать, в какое место вернуться. Поскольку адрес возврата заносится после аргументов, то относительно вершины стека аргументы оказываются «за» ним и их смещение варьируется в зависимости от того, один элемент занимает адрес возврата или два. К счастью, плоская модель памяти Windows NT\9x позволяет забыть о моделях памяти как о страшном сне и всюду использовать только ближние вызовы.

Неоптимизирующие компиляторы используют для адресации аргументов специальный регистр (как правило, `EBP`), копируя в него значение регистра-указателя вершины стека в самом начале функции. Поскольку стек растет снизу вверх, т. е. от старших адресов к младшим, смещение всех аргументов (включая адрес возврата) положительно, а смещение *N*-го по счету аргумента вычисляется по следующей формуле:

$$arg_offset = N * size_element + size_return_address,$$

где *N* — номер аргумента, считая от вершины стека, начиная с нуля; *size_element* — размер одного элемента стека, в общем случае равный разрядности сегмента (под Windows NT\9x — четыре байта); *size_return_address* — размер в

байтах, занимаемый адресом возврата (под Windows NT\9x — обычно четыре байта).

Часто приходится решать и обратную задачу: зная смещение элемента, определять, к какому по счету аргументу происходит обращение. В этом нам поможет следующая формула, элементарно выводющаяся из предыдущей:

$$N = \frac{arg_offset - size_return_address}{size_element}.$$

Поскольку перед копированием в EBP текущего значения ESP старое значение EBP приходится сохранять в том же самом стеке, в приведенную формулу приходится вносить поправку, добавляя к размеру адреса возврата еще и размер регистра EBP (BP в 16-разрядном режиме, который все еще жив на сегодняшний день).

С точки зрения хакера, главное достоинство такой адресации аргументов в том, что, увидев где-то в середине кода инструкцию типа *MOV EAX,[EBP+0x10]*, можно мгновенно вычислить, к какому именно аргументу происходит обращение. Однако оптимизирующие компиляторы для экономии регистра EBP адресуют аргументы непосредственно через ESP. Разница принципиальна! Значение ESP не остается постоянным на протяжении выполнения функции и изменяется всякий раз при занесении и снятии данных из стека, следовательно, не остается постоянным и смещение аргументов относительно ESP. Теперь, чтобы определить, к какому именно аргументу происходит обращение, необходимо знать, чему равен ESP в данной точке программы, а для выяснения этого все его изменения придется отслеживать от самого начала функции! Подробнее о такой «хитрой» адресации мы поговорим потом (см. раздел «Идентификация локальных стековых переменных»), а для начала вернемся к предыдущему примеру (надо ж его «добить») и разберем вызываемую функцию:

Листинг 59

```
MyFunc      proc near          ; CODE XREF: main+39p
arg_0       = dword ptr 8
arg_4       = dword ptr 0Ch
arg_8       = byte ptr 10h
arg_1C      = dword ptr 24h
; IDA распознала четыре аргумента, передаваемых функции. Однако
; не стоит безоговорочно этому доверять - если один аргумент (например, int64)
; передается в нескольких машинных словах, то IDA ошибочно примет его не за один,
; а за несколько аргументов!
; Поэтому результат, полученный IDA, надо трактовать так: функции передается не менее
; четырёх аргументов. Впрочем, и здесь не все гладко! Ведь никто не мешает вызываемой
; функции залезать в стек материнской так далеко, как она захочет! Может быть,
; нам не передавали никаких аргументов вовсе, а мы самовольно полезли в стек и
; стянули что-то оттуда. Хотя это случается в основном вследствие программистских
; ошибок из-за путаницы с прототипами, считаться с такой возможностью необходимо.
; (Когда-нибудь вы все равно с этим встретитесь, так что будьте готовы.)
; Число, стоящее после arg, выражает смещение аргумента относительно начала
; кадра стека.
```

; Обратите внимание: сам кадр стека смещен на восемь байтов относительно EBP –
 ; четыре байта занимает сохраненный адрес возврата и еще четыре уходят на сохранение
 ; регистра EBP.

```
push    ebp
mov     ebp, esp
lea     eax, [ebp+arg_8]
; получение указателя на аргумент.
; Внимание: именно указателя на аргумент, а не аргумента-указателя!
; Теперь разберемся, на какой именно аргумент мы получаем указатель.
; IDA уже вычислила, что этот аргумент смещен на восемь байтов относительно
; начала кадра стека. В оригинале выражение, заключенное в скобках, выглядело
; как ebp+0x10 – так его и отображает большинство дизассемблеров.
; Не будь IDA такой умной, нам бы пришлось постоянно
; вручную отнимать по восемь байтов от каждого такого
; адресного выражения (впрочем, с этим мы еще поупражняемся).
;
; Логично: на вершине то, что мы клали в стек в последнюю очередь.
; Смотрим вызывающую функцию – что ж мы клали-то?
; (см. вариант, откомпилированный Microsoft Visual C++).
; Ага, последними были те два непонятных аргумента, а перед ними в стек
; засылалась структура, состоящая из строки и переменной типа int.
; Таким образом, EBP+ARG_8 указывает на строку
```

```
push    eax
; Засылаем в стек полученный указатель.
; Похоже, что он передается очередной функции.
```

```
mov     ecx, [ebp+arg_1C]
; Заносим в ECX содержимое аргумента EBP+ARG_1C. На что он указывает?
; Вспомним, что тип int находится в структуре по смещению 0x14 байтов от начала,
; а ARG_8 и есть ее начало. Тогда 0x8+0x14 == 0x1C.
; То есть в ECX заносится значение переменной типа int, члена структуры
```

```
push    ecx
; Заносим полученную переменную в стек, передавая ее по значению
; (по значению потому, что ECX хранит значение, а не указатель)
```

```
mov     edx, [ebp+arg_4]
; Берем один из тех двух непонятных аргументов, занесенных последними в стек
```

```
push    edx
; ...и вновь заталкиваем в стек, передавая его очередной функции.
```

```
mov     eax, [ebp+arg_0]
push    eax
; Берем второй непонятный аргумент и пишем его в стек.
```

```
push    offset aFXS          ; "%f,%x,%s\n"
call    _printf
; Опа! Вызов printf с передачей строки спецификаторов! Функция printf,
; как известно, имеет переменное число аргументов, тип и количество которых
; как раз и задают спецификаторы.
; Вспомним, сперва в стек мы заносили указатель на строку, и действительно,
; крайний правый спецификатор "%s" обозначает вывод строки.
; Затем в стек заносилась переменная типа int и второй справа спецификатор,
```



```

; есть %x - вывод целого в шестнадцатеричной форме.
; А вот затем... затем идет последний спецификатор %f, в то время как в стек
; заносились два аргумента.
; Заглянув в руководство программиста по Microsoft Visual C++, мы прочтем,
; что спецификатор %f выводит вещественное значение, которое в зависимости от
; типа может занимать и четыре байта (float), и восемь (double).
; В нашем случае оно явно занимает восемь байтов, следовательно, это double.
; Таким образом, мы восстановили прототип нашей функции, вот он:
; cdecl MyFunc(double a, struct B b)
; Тип вызова cdecl - то есть стек вычищала вызывающая функция. Вот только, увы,
; подлинный порядок передачи аргументов восстановить невозможно. Вспомним,
; Borland C++ также вычищал стек вызывающей функцией, но самовольно изменил
; порядок передачи параметров.
; Кажется, если программа компилилась Borland C++, то мы просто изменяем
; порядок аргументов на обратный, вот и все. Увы, это не так просто. Если имело
; место явное преобразование типа функции в cdecl, то Borland C++ без лишней
; самостоятельности поступил бы так, как ему велели, и тогда бы обращение
; порядка аргументов дало бы неверный результат!
; Впрочем, подлинный порядок следования аргументов в прототипе функции
; не играет никакой роли. Важно лишь связать передаваемые и принимаемые
; аргументы, что мы и сделали.
; Обратите внимание: это стало возможно лишь при совместном анализе и вызываемой
; и вызывающей функций! Анализ лишь одной из них ничего бы не дал!
; Примечание: никогда не следует безоговорочно полагаться на достоверность
; строки спецификаторов. Поскольку спецификаторы формируются "вручную" самим
; программистом, тут возможны ошибки, подчас весьма трудноуловимые и дающие
; после компиляции чрезвычайно загадочный код!
add     esp, 14h
pop     ebp
retn
MyFunc  endp

```

Так, кое-какие продвижения уже есть — мы уверенно восстановили прототип нашей первой функции. Но это только начало... Еще много миль предстоит пройти, прежде чем будет достигнут конец главы. Если вы устали — передохните. Тяпните пивка (колы), позвоните своей любимой девушке (а что, у хакеров и любимые девушки есть?), словом, как хотите, но обеспечьте свежую голову. Мы приступаем еще к одной нудной, но очень важной теме — сравнительному анализу различных типов вызовов функций и их реализации в популярных компиляторах.

Начнем с изучения стандартного соглашения о вызове — *stdcall*. Рассмотрим следующий пример:

Листинг 60. Демонстрация stdcall

```

#include <stdio.h>
#include <string.h>

__stdcall MyFunc(int a, int b, char *c)
{
    return a+b+strlen(c);
}

```

```
main()
{
    printf("%x\n", MyFunc(0x666, 0x777, "Hello, World!"));
}
```

Результат его компиляции Microsoft Visual C++ с настройками по умолчанию должен выглядеть так:

Листинг 61

```
main          proc near          ; CODE XREF: start+AFp
    push      ebp
    mov       ebp, esp

    push offset aHelloWorld ; const char *
    ; Заносим в стек указатель на строку aHelloWorld.
    ; Заглянув в исходные тексты (благо они у нас есть), мы обнаружим, что
    ; это самый правый аргумент, передаваемый функции. Следовательно,
    ; перед нами вызов типа stdcall или cdecl, но не PASCAL.
    ; Обратите внимание, строка передается по ссылке, но не по значению.

    push      777h ; int
    ; Заносим в стек еще один аргумент - константу типа int.
    ; (IDA, начиная с версии 4.17, автоматически определяет ее тип).

    push      666h ; int
    ; Передаем функции последний, самый левый аргумент - константу типа int

    call      MyFunc
    ; Обратите внимание, после вызова функции отсутствуют команды очистки стека
    ; от занесенных в него аргументов. Если компилятор не схитрил и не прибегнул
    ; к отложенной очистке, то, скорее всего, стек очищает сама вызываемая функция,
    ; значит, тип вызова - stdcall (что, собственно, и требовалось доказать)

    push      eax
    ; Передаем возвращенное функцией значение следующей функции как аргумент

    push      offset asc_406040 ; "%x\n"
    call      _printf
    ; OK, эта следующая функция printf, и строка спецификаторов показывает,
    ; что переданный аргумент имеет тип int

    add       esp, 8
    ; Выталкивание восьми байтов из стека - четыре приходится на аргумент типа int,
    ; остальные четыре - на указатель на строку спецификаторов

    pop       ebp
    retn

main endp

; int __cdecl MyFunc(int,int,const char *)
MyFunc      proc near ; CODE XREF: sub_40101D+12p
    ; С версии 4.17 IDA автоматически восстанавливает прототипы функций,
    ; но делает это не всегда правильно. В данном случае
    ; она допустила грубую ошибку - тип вызова
    ; никак не может иметь тип cdecl,
    ; так как стек вычищает вызываемая функция!
```

```

; Сдается, что вообще не предпринимает никаких попыток анализа типа вызова,
; а берет его из настроек распознанного компилятора по умолчанию.
; В общем, как бы там ни было, но с результатами работы IDA следует обращаться
; очень осторожно.

arg_0      = dword ptr 8
arg_4      = dword ptr 0Ch
arg_8      = dword ptr 10h

push      ebp
mov       ebp, esp
push      esi
; Это, как видно, сохранение регистра в стеке, а не передача его функции, так как
; регистр явным образом не инициализировался ни вызывающей, ни вызываемой
; функцией.

mov       esi, [ebp+arg_0]
; Заносим в регистр ESI последний занесенный в стек аргумент

add       esi, [ebp+arg_4]
; Складываем содержимое ESI с предпоследним занесенным в стек аргументом

mov       eax, [ebp+arg_8]
; Заносим в EAX пред- предпоследний аргумент и...

push      eax                ; const char *
; засылаем его в стек.

call      _strlen
; Поскольку strlen ожидает указателя на строку, можно с уверенностью
; заключить, что предпредпоследний аргумент - строка, переданная по ссылке.

add       esp, 4
; Вычистка последнего аргумента из стека

add       eax, esi
; Как мы помним, в ESI хранится сумма двух первых аргументов,
; а в EAX - возвращенная длина строки. Таким образом, функция суммирует
; два своих аргумента с длиной строки.

pop       esi
pop       ebp
ret      0Ch
; Стек чистит вызываемая функция, следовательно, тип вызова stdcall или PASCAL.
; Будем считать, что это stdcall, тогда прототип функции выглядит так:
; int MyFunc(int a, int b, char *c)
;
; Порядок аргументов вытекает из того, что на вершине стека были две
; переменные типа int, а под ними строка. Поскольку на верху стека лежит
; всегда то, что заносилось в него в последнюю очередь, а по stdcall
; аргументы заносятся справа налево, мы получаем именно такой порядок
; следования аргументов

MyFunc    endp

```

А теперь рассмотрим, как происходит вызов cdecl-функции. Изменим в предыдущем примере ключевое слово stdcall на cdecl:

Листинг 62. Демонстрация cdecl

```

#include <stdio.h>
#include <string.h>

__cdecl MyFunc(int a, int b, char *c)
{
    return a+b+strlen(c);
}

main()
{
    printf("%x\n", MyFunc(0x666, 0x777, "Hello, World!"));
}

```

Результат компиляции должен выглядеть так:

Листинг 63

```

main                proc near                ; CODE XREF: start+AFp
    push            ebp
    mov             ebp, esp

    push            offset aHelloWorld      ; const char *
    push            777h                    ; int
    push            666h                    ; int
    ; Передаем функции аргументы через стек

    call            MyFunc
    add             esp, 0Ch
    ; Смотрите, стек вычищает вызывающая функция. Значит, тип вызова cdecl,
    ; поскольку все остальные предписывают вычищать стек вызываемой функции.

    push            eax
    push            offset asc_406040       ; "%x\n"
    call            _printf
    add             esp, 8
    pop             ebp
    retn
main                endp

; int __cdecl MyFunc(int,int,const char *)
; А вот сейчас IDA правильно определила тип вызова. Однако, как уже показано выше,
; она могла и ошибиться, поэтому полагаться на нее не стоит.

MyFunc              proc near                ; CODE XREF: main+12p
arg_0               = dword ptr 8
arg_4               = dword ptr 0Ch
arg_8               = dword ptr 10h
; Поскольку, как мы уже выяснили, функция имеет тип cdecl, аргументы передаются
; справа налево и ее прототип выглядит так: MyFunc(int arg_0, int arg_4, char *arg_8)

    push            ebp
    mov             ebp, esp
    push            esi
    ; Сохраняем ESI в стеке

```

```

    mov     esi, [ebp+arg_0]
    ; Заносим в ESI аргумент arg_0 типа int

    add     esi, [ebp+arg_4]
    ; Складываем его с arg_4

    mov     eax, [ebp+arg_8]
    ; Заносим в EAX указатель на строку

    push    eax ; const char *
    ; Передаем его функции strlen через стек

    call    _strlen
    add     esp, 4

    add     eax, esi
    ; Добавляем к сумме arg_0 и arg_4 длину строки arg_8

    pop     esi
    pop     ebp
    retn
MyFunc     endp

```

Прежде чем перейти к вещам по-настоящему серьезным, рассмотрим на закуску последний стандартный тип — PASCAL:

Листинг 64. Демонстрация вызова PASCAL

```

#include <stdio.h>
#include <string.h>

// Внимание! Microsoft Visual C++ уже не поддерживает тип вызова PASCAL, вместо этого
// используйте аналогичный ему тип вызова WINAPI, определенный в файле <windows.h>.
#ifdef _MSC_VER
#include <windows.h>
// включать windows.h, только если мы компилируем Microsoft Visual C++,
// для остальных компиляторов более эффективное решение - использование ключевого
// слова PASCAL, если они, конечно, его поддерживают. (Borland поддерживает)
#endif

// Подобный прием программирования может и делает листинг менее читабельным,
// но зато позволяет компилировать его не только одним компилятором!
#ifdef _MSC_VER
WINAPI
#else
__pascal
#endif

MyFunc(int a, int b, char *c)
{
    return a+b+strlen(c);
}

main()
{
    printf("%x\n", MyFunc(0x666, 0x777, "Hello, World!"));
}

```

Результат компиляции Borland C++ должен выглядеть так:

Листинг 65

```
; int __cdecl main(int argc,const char **argv,const char *envp)
_main          proc near          ; DATA XREF: DATA:00407044o

    push      ebp
    mov       ebp, esp

    push      666h                ; int
    push      777h                ; int
    push      offset aHelloWorld ; s
    ; Передаем функции аргументы. Заглянув в исходный текст, мы заметим, что
    ; аргументы передаются слева направо. Однако если исходных текстов нет,
    ; установить этот факт невозможно! К счастью, подлинный прототип функции
    ; не важен.

    call      MyFunc
    ; Функция не вычищает за собой стек! Если это не результат оптимизации,
    ; ее тип вызова либо PASCAL, либо stdcall. Ввиду того что PASCAL уже вышел
    ; из употребления, будем считать, что имеем дело с stdcall

    push      eax
    push      offset unk_407074    ; format
    call      _printf
    add       esp, 8

    xor       eax, eax
    pop       ebp
retn
_main          endp

; int __cdecl MyFunc(const char *s,int,int)
; Ага! IDA вновь дала неправильный результат! Тип вызова явно не cdecl!
; Однако в остальном прототип функции верен, вернее, не то чтобы он верен
; (на самом деле порядок аргументов обратный), но для использования пригоден

MyFunc         proc near          ; CODE XREF: _main+12p

s              = dword ptr 8
arg_4          = dword ptr 0Ch
arg_8          = dword ptr 10h

    push      ebp
    mov       ebp, esp
    ; Открываем кадр стека

    mov       eax, [ebp+s]
    ; Заносим в EAX указатель на строку

    push      eax                ; s
    call      _strlen
    ; Передаем его функции strlen

    pop       ecx
    ; Очищаем стек от одного аргумента, выталкивая его в неиспользуемый регистр
```

```

mov     edx, [ebp+arg_8]
; Заносим в EDX аргумент arg_8 типа int

add     edx, [ebp+arg_4]
; Складываем его с аргументом arg_4

add     eax, edx
; Складываем сумму arg_8 и arg_4 с длиной строки

pop     ebp
retn    0Ch
; Стек чистит вызываемая функция. Значит, ее тип PASCAL или stdcall

```

```
MyFunc    endp
```

Как мы видим, идентификация базовых типов вызова и восстановление прототипов функции — занятие несложное. Единственное, что портит настроение, — путаница между PASCAL и stdcall, но порядок занесения аргументов в стек не имеет никакого значения, разве что в особых случаях, один из которых перед вами:

Листинг 66. Пример, демонстрирующий тот случай, когда требуется точно отличать PASCAL от stdcall

```

#include <stdio.h>
#include <windows.h>
#include <winuser.h>

// CALLBACK - процедура для приема сообщений от таймера
VOID CALLBACK TimerProc(
    HWND hwnd,      // handle of window for timer messages
    UINT uMsg,      // WM_TIMER message
    UINT idEvent,    // timer identifier
    DWORD dwTime     // current system time
)
{
    MessageBeep((dwTime % 5)*0x10);    // Бибикаем всеми пиками на все голоса

    // Выводим время в секундах, прошедшее с момента пуска системы
    printf("\r:=%d", dwTime / 1000);
}

main()
// Да, это консольное приложение, но оно также может иметь цикл выборки сообщений
// и устанавливать таймер!
{
    int a;
    MSG msg;

    // Устанавливаем таймер, передавая ему адрес процедуры TimerProc
    SetTimer(0, 0, 1000, TimerProc);

    // Цикл выборки сообщений. Когда надоест - жмем Ctrl-Break и прерываем его
    while (GetMessage(&msg, (HWND) NULL, 0, 0))
    {
        TranslateMessage(&msg);
        DispatchMessage(&msg);
    }
}

```

Откомпилируем этот пример так: `cl pascal.callback.c USER32.lib` — и посмотрим, что из этого получилось:

Листинг 67

```
main          proc near          ; CODE XREF: start+AFp
; На сей раз IDA не определила прототип функции. Ну и ладно...

Msg           = MSG ptr -20h
; IDA распознала одну локальную переменную и даже восстановила ее тип,
; что не может не радовать

    push      ebp
    mov       ebp, esp
    sub       esp, 20h

    push      offset TimerProc    ; lpTimerFunc
    ; Передаем указатель на функцию TimerProc

    push      1000                ; uElapse
    ; Передаем время задержки таймера

    push      0                   ; nIDEvent
    ; В консольных приложениях аргумент nIDEvent всегда игнорируется

    push      0                   ; hWnd
    ; Окон нет, передаем NULL

    call      ds:SetTimer
    ; Win32 API-функции вызываются по соглашению stdcall - это дает возможность,
    ; зная их прототип(а он описан в SDK), восстановить тип и назначение аргументов,
    ; в данном случае исходный текст выглядел так:
    ; SetTimer(NULL, BULL, 1000, TimerProc);

loc_401051:    ; CODE XREF: main+42j
    push      0                   ; wParamFilterMax
    ; NULL - нет фильтра

    push      0                   ; wParamFilterMin
    ; NULL - нет фильтра

    push      0                   ; hWnd
    ; NULL - нет окон в консольном приложении

    lea       eax, [ebp+Msg]
    ; Получаем указатель на локальную переменную msg -
    ; тип этой переменной определяется, кстати, только на основе прототипа
    ; функции GetMessageA

    push      eax                 ; lpMsg
    ; Передаем указатель на msg

    call      ds:GetMessageA
    ; Вызываем функцию GetMessageA (&msg, NULL, NULL, NULL);

    test      eax, eax
    jz        short loc_40107B
    ; Проверка на получение WM_QUIT
```



```

    lea    ecx, [ebp+Msg]
    ; В ECX - указатель на заполненную структуру MSG...

    push   ecx                ; lpMsg
    ; ...передаем его функции TranslateMessage

    call   ds:TranslateMessage
    ; Вызываем функцию TranslateMessage(&msg);

    lea    edx, [ebp+Msg]
    ; В EDX - указатель на msg...

    push   edx                ; lpMsg
    ; ...передаем его функции DispatchMessageA

    call   ds:DispatchMessageA
    ; Вызов функции DispatchMessageA

    jmp     short loc_401051
    ; Цикл выборки сообщений

loc_40107B:                    ; CODE XREF: main+2Cj
    ; Выход

    mov    esp, ebp
    pop    ebp
    retn

main                            endp

TimerProc    proc near        ; DATA XREF: main+60
; Прототип TimerProc вследствие ее неявного вызова операционной системой
; не был автоматически восстановлен IDA, этим придется заниматься нам.
; Мы знаем, что TimerProc передается функции SetTimer.
; Заглянув в описание SetTimer (SDK всегда должен быть под рукой!), мы найдем
; ее прототип:
;
; VOID CALLBACK TimerProc(
;   HWND hwnd,      // handle of window for timer messages
;   UINT uMsg,      // WM_TIMER message
;   UINT idEvent,    // timer identifier
;   DWORD dwTime     // current system time
; )
;
; Остается разобраться с типом вызова. На сей раз он принципиален, так как, не имея
; кода вызывающей функции (он расположен глубоко в недрах операционной системы),
; мы разберемся с типами аргументов только в том случае, если будем знать их
; порядок передачи.
; Выше уже говорилось, что все CALLBACK-функции следуют соглашению PASCAL.
; Не путайте CALLBACK-функции с Win32 API-функциями! Первые вызывает сама
; операционная система, а вторые - прикладная программа.
;
; OK, тип вызова этой функции - PASCAL. Значит, аргументы заносятся слева направо,
; а стек чистит вызываемая функция (убедитесь, что это действительно так).

arg_C        = dword ptr 14h
; IDA обнаружила только один аргумент, хотя, судя по прототипу, их передается четыре.
; Почему? Очень просто - функция использовала всего один аргумент, а к остальным и

```

```

; не обращалась. Вот IDA и не смогла их восстановить!
; Кстати, что это за аргумент? Смотрим: его смещение равно 0xC. А на вершине стека то,
; что в него заталкивалось в последнюю очередь. Внизу соответственно наоборот.
; Постой, постой, что за чертовщина?! Выходит, dwTime был занесен в стек в первую
; очередь?! (Мы-то, имея исходный текст, знаем, что arg_C наверняка dwTime.)
; Но ведь соглашение PASCAL диктует противоположный порядок занесения аргументов!
; Что-то здесь не так... но ведь программа работает (запустите ее, чтобы проверить).
; А в SDK написано, что CALLBACK - аналог FAR PASACAL. С FAR'ом понятно, в Win9x\NT
; все вызовы ближние, но вот как объяснить инверсию засылки аргументов?!
; Сдается?(Нет, не сдавайтесь, попытайтесь найти решение сами - иначе какой интерес?)
; Тогда загляните в <windef.h> и посмотрите, как там определен тип PASCAL
;
; #elif (_MSC_VER >= 800) || defined(_STDCALL_SUPPORTED)
; #define CALLBACK    __stdcall
; #define WINAPI      __stdcall
; #define WINAPIV     __cdecl
; #define APIENTRY    WINAPI
; #define APIPRIVATE  __stdcall
; #define PASCAL      __stdcall
;
; Нет, ну кто бы мог подумать!!! Вызов, объявленный как PASCAL, на самом деле
; представляет собой stdcall! И CALLBACK - также определен, как stdcall.
; Наконец-то все объяснилось! Теперь, если вам скажут, что CALLBACK - это PASCAL,
; вы можете усмехнуться и сказать, что еж тоже птица, правда, гордая - пока не пнешь,
; не полетит! (Оказывается, копания в дебрях include-файлов могут приносить пользу.)
; Кстати, это извращения с перекрытием типов создают большую проблему при подключении
; к Си-проекту модулей, написанных в среде, поддерживающей PASCAL-соглашения о вызове
; функций. Поскольку в Windows PASCAL никакой не PASCAL, а stdcall, ничего работать
; соответственно не будет! Правда, есть еще ключевое слово __pascal, которое не
; перекрывается, но и не поддерживается последними версиями Microsoft Visual C++.
; Выход состоит в использовании ассемблерных вставок или переходе на Borland C++
; Он, как и многие другие компиляторы, соглашение PASCAL до сих пор исправно
; поддерживает.
;
; Итак, мы выяснили, что аргументы CALLBACK-функциям передаются справа налево, но
; стек вычищает сама вызываемая функция, как и положено по stdcall-соглашению.

```

```

push    ebp
mov     ebp, esp

mov     eax, [ebp+arg_C]
; заносим в EAX аргумент dwTime.
; Как мы получили его? Смотрим, перед ним в стеке лежат три аргумента,
; каждый из которых размером в 4 байта, тогда 4*3=0xC

xor     edx, edx
; Обнуляем EDX

mov     ecx, 5
; Присваиваем ECX значение 5

div     ecx
; Делим dwTime (он в EAX) на 5

shl     edx, 4

```

```

; В EDX - остаток от деления, циклическим сдвигом умножаем его на 0x10
; точнее, умножаем его на 24

push    edx                ; uType
; Передаем полученный результат функции MessageBeep.
; Заглянув в SDK, мы найдем, что MessageBeep принимает одну из констант:
; NB_OK, MB_ICONASTERISK, MB_ICONHAND и т. д., но там ничего не сказано о том,
; какое непосредственное значение каждое из них принимает.
; Зато сообщается, что MessageBeep описана в файле <WINUSER.h>
; Открываем его и ищем контекстным поиском MB_OK:
;
; #define MB_OK                0x00000000L
; #define MB_OKCANCEL         0x00000001L
; #define MB_ABORTRETRYIGNORE 0x00000002L
; #define MB_YESNOCANCEL      0x00000003L
; #define MB_YESNO            0x00000004L
; #define MB_RETRYCANCEL      0x00000005L
;
; #define MB_ICONHAND          0x00000010L
; #define MB_ICONQUESTION     0x00000020L
; #define MB_ICONEXCLAMATION  0x00000030L
; #define MB_ICONASTERISK     0x00000040L
;
; Есть хвост у Тигры! Смотрите: все интересующие нас константы равны:
; 0x0, 0x10, 0x20, 0x30, 0x40. Теперь становится понятным смысл программы.
; Взяв остаток, полученный делением количества миллисекунд, прошедших с минуты
; включения системы на 5, мы получаем число в интервале от 0 до 4. Умножая его
; на 0x10, - 0x0, 0x0x10 - 0x40.

call    ds:MessageBeep
; Библикаем на все лады

mov     eax, [ebp+arg_C]
; Заносим в EAX dwTime

xor     edx, edx
; Обнуляем EDX

mov     ecx, 3E8h
; В десятичном 0x3E8 равно 1000

div     ecx
; Делим dwTime на 1000, т. е. переводим миллисекунды в секунды и...

push    eax
; передаем его функции printf

push    offset aD           ; "\r:=%d"
call    _printf
add     esp, 8
; printf("\r:=%d")

pop     ebp
retn    10h
; Выходя, гасите свет, т. е. чистите за собой стек!

TimerProc    endp

```

***Важное замечание о типах, определенных в <WINDOWS.H>!** Хотя об этом уже говорилось в комментариях к предыдущему листингу, повторение не будет лишним хотя бы уже потому, что не все читатели вчитываются в разборы дизассемблерных текстов.*

Итак, функции CALLBACK- и WINAPI следуют соглашению о вызовах PASCAL, но сам PASCAL определен в <WINDEF.H> как stdcall (а на некоторых платформах и как cdecl). Таким образом, на платформе INTEL все Windows-функции следуют соглашению: аргументы заносятся справа налево, а стек вычищает вызываемая функция.

Давайте для знакомства с PASCAL-соглашением создадим простенькую PASCAL-программу и дизассемблируем ее (это не означает, что PASCAL-вызовы встречаются только в PASCAL-программах, но так будет справедливо):

Листинг 68. Демонстрация PASCAL-вызова

```
USES WINCRT;

Procedure MyProc(a:Word; b:Byte; c:String);
begin
    WriteLn(a+b, ' ',c);
end;

BEGIN
    MyProc($666,$77, 'Hello, Sailor! ');
END.
```

Результат компиляции компилятором Turbo-Pascal for Windows должен выглядеть так:

Листинг 69

```
PROGRAM      proc near
    call      INITTASK
    ; Вызов INITTASK из KRNL386.EXE для инициализации 16-разрядной задачи

    call      @__SystemInit$qv      ; __SystemInit(void)
    ; Инициализация модуля SYSTEM

    call      @__WINCRTInit$qv      ; __WINCRTInit(void)
    ; Инициализация модуля WinCRT

    push      bp
    mov       bp, sp
    ; Пролог функции в середине функции!
    ; Вот такой он, Turbo-PASCAL!

    xor       ax, ax
    call      @__StackCheck$q4Word ; Stack overflow check (AX)
    ; Проверка стека на переполнение

    push      666h
    ; Обратите внимание, передача аргументов идет слева направо

    push      77h ; 'w'
```

```

mov     di, offset aHelloSailor ; "Hello,Sailor!"
; B DI - указатель на строку "Hello, Sailor"

push    ds
push    di
; Смотрите: передается не ближний (NEAR), а дальний (FAR) указатель,
; т. е. и сегмент, и смещение строки.

call    MyProc
; Стек чистит вызываемая функция.

leave
; Эпилог функции - закрытие кадра стека.

xor     ax, ax
call    @Halt$q4Word           ; Halt(Word)
; Конец программы!

```

PROGRAM endp

```

MyProc      proc near           ; CODE XREF: PROGRAM+23p
; IDA не определила прототип функции. Что ж, сделаем это сами!

```

```

var_100     = byte ptr -100h
; Локальная переменная. Судя по тому, что она находится на 0x100 байтов выше кадра
; стека, сдается, что этот массив их 0x100 байтов. Поскольку максимальная длина строки
; в PASCAL как раз и равна 0xFF байтам. Похоже, это буфер, зарезервированный под
; строку.

```

```

arg_0       = dword ptr  4
arg_4       = byte ptr  8
arg_6       = word ptr  0Ah
; Функция принимает три аргумента

```

```

push    bp
mov     bp, sp
; Открываем кадр стека

mov     ax, 100h
call    @_StackCheck$q4Word ; Stack overflow check (AX)
; Проверяем, есть ли в стеке необходимые нам 100 байтов для локальных переменных

sub     sp, 100h
; Резервируем пространство под локальные переменные

les     di, [bp+arg_0]
; получаем указатель на самый правый аргумент

push    es
push    di
; Смотрите, передаем дальний указатель на аргумент arg_0, причем его
; сегмент из стека даже не извлекался!

lea     di, [bp+var_100]
; Получаем указатель на локальный буфер

push    ss
; Заносим его сегмент в стек

```

```

push    di
; Заносим смещение буфера в стек

push    0FFh
; Заносим максимальную длину строки

call    @$basg$qm6Stringt14Byte ; Store string
; Копируем строку в локальный буфер (значит, arg_0 - это строка).
; Правда, совершенно непонятно зачем. Неужто нельзя пользоваться ссылкой?
; Дурной, дурной этот Turbo-Pascal!
; Да что делать, в самом Паскале строки передаются по значению :-(

mov     di, offset unk_1E18
; Получаем указатель на буфер вывода.
; Тут надобно познакомиться с системой вывода Паскаля - она весьма разительно
; отличается от Си.
; Во-первых, левосторонний порядок засылки аргументов в стек не позволяет
; организовать поддержку процедур с переменным числом аргументов
; (во всяком случае, без дополнительных ухищрений).
; Но ведь WriteLn и есть процедура с переменным числом параметров. Разве нет?!
; Вот именно, что нет!!! Никакая это не процедура, а оператор!
; Компилятор еще на стадии компиляции разбивает ее на множество вызовов
; процедур для вывода каждого аргумента по отдельности. Поэтому
; в откомпилированном коде каждая процедура примет фиксированное количество
; аргументов. В нашем случае их будет три: первая для вывода суммы двух
; чисел - этим занимается процедура WriteLongint, - вторая - для вывода символа
; пробела в символьной форме - этим занимается WriteChar - и, наконец, последняя
; для вывода строки - WriteSting.
; Размышляем далее - под Windows непосредственно вывести строку в окно и тут же
; забыть о ней нельзя, так как окно в любой момент может потребовать перерисовки, -
; операционная система не сохраняет его содержимого - в графической среде
; при высоком разрешении это привело бы к большим затратам памяти.
; Код, выводящий строку, должен уметь повторять свой вывод по запросу.
; Каждый, кто хоть раз программировал под Windows, наверняка помнит, что весь
; вывод приходилось помещать в обработчик сообщения WM_PAINT.
; Turbo-Pascal же позволяет обращаться с Windows-окном точно так,
; как с консолью. А раз так, он должен где-то хранить все, выведенное ранее
; на экран. Поскольку локальные переменные умирают вместе с завершением
; их процедуры, то для хранения буфера они не годятся. Остается либо куча, либо
; сегмент данных. PASCAL использует последнее - указатель на такой буфер мы
; только что получили.
; Далее, для повышения производительности вывода Turbo-Pascal реализует
; простейший кэш. Функции WriteLongint, WriteChar, WriteString сливают
; результат своей деятельности в символьном виде в этот самый буфер, а в конце
; следует вызов WriteLn, выводящий содержимое буфера в окно.
; Run-time systems следит за его перерисовками и при необходимости повторяет
; вывод уже без участия программиста.

push    ds
push    di
; Заносим адрес буфера в стек

mov     al, [bp+arg_4]
; Тип аргумента arg_4 - Byte

```

```

xor    ah, ah
; Обнуляем старший байт регистра ah

add     ax, [bp+arg_6]
; Складываем arg_4 с arg_6. Поскольку al было предварительно расширено до AX,
; то arg_6 имеет тип Word, так как при сложении двух чисел разного типа PASCAL
; расширяет их до большего из них.
; Кроме того, вызывающая процедура передает с этим аргументом значение 0x666,
; что явно не влезло бы в Byte.

xor     dx, dx
; Обнуляем DX...

push    dx
; и заносим его в стек.

push    ax
; Заносим в стек сумму двух левых аргументов

push    0
; Еще один ноль!

call    @Write$qm4Text7Longint4Word ; Write(var f; v: Longint; width: Word)
; Функция WriteLongint имеет следующий прототип
; WriteLongint(Text far &, a: Longint, count:Word); где
; Text far & - указатель на буфер вывода,
; a - выводимое длинное целое
; count - сколько переменных выводить (ноль - одна переменная)
;
; Значит, в нашем случае мы выводим одну переменную - сумму двух аргументов.
; Маленькое дополнение - функция WriteLongint не следует соглашению PASCAL,
; так как не до конца чистит за собой стек, оставляя указать на буфер в стеке.
; На этот шаг разработчики компилятора пошли для увеличения производительности:
; раз указатель на буфер будет нужен и другим функциям (по крайней мере, одной
; из них - WriteLn), зачем его то стягивать, то опять лихорадочно запихивать?
; Если вы заглянете в конец функции WriteLongint, вы обнаружите там RET 6,
; т. е. функция выпихивает два аргумента - два машинных слова на Longint и один
; Word на count.
; Вот такая милая маленькая техническая деталь. Маленькая-то она, маленькая,
; но как сбивает с толку!
; (особенно если исследователь незнаком с системой ввода-вывода Паскаля)

push    20h
;
; Заносим в стек следующий аргумент, передаваемый функции WriteLn
; (указатель на буфер все еще находится в стеке).

push    0
; Нам надо вывести только один символ

call    @Write$qm4Text4Char4Word ; Write(var f;c: Char; width:Word)

lea     di, [bp+var_100]
; Получаем указатель на локальную копию переданной функции строки

push    ss
push    di
; Заносим ее адрес в стек

```

```

push    0
; Выводить только одну строку!

call    @Write$qm4Textm6String4Word ; Write(var f; s: String; width: Word)

call    @WriteLn$qm4Text      ; WriteLn(var f: Text)
; Кажется, функции не передаются никакие параметры, но на самом деле на вершине
; стека лежит указатель на буфер и ждет своего "звездного часа",
; после завершения WriteLn он будет снят со стека

call    @__IOCheck$qv          ; Exit if error
; Проверка операции вывода на успешность

leave
; Закрываем кадр стека

ret     8
; Выталкиваем восемь байтов со стека. OK, теперь мы знаем все необходимое для
; восстановления прототипа нашей процедуры. Он выглядит так:
; MyProc(a:Byte, b:Word, c:String);

```

```
MyProc      endp
```

Да, хитрым оказался Turbo-Pascal! Анализ откомпилированной с его помощью программы преподнес нам хороший урок — никогда нельзя быть уверенным, что функция выталкивает все переданные ей аргументы из стека, и уж тем более нельзя определять количество аргументов по числу снимаемых из стека машинных слов!

Соглашения о быстрых вызовах — *fastcall*. Какой бы непроизводительной передача аргументов через стек ни была, типы вызова *stdcall* и *cdecl* стандартизованы, и хочешь не хочешь, а их надо соблюдать. Иначе модули, скомпилированные одним компилятором (например, библиотеки), окажутся несовместимы с модулями, скомпилированными другими компиляторами. Впрочем, если вызываемая функция компилируется тем же самым компилятором, что и вызывающая, придерживаться типовых соглашений ни к чему и можно воспользоваться более эффективной передачей аргументов через регистры.

Многие начинающие программисты удивляются: а почему передача аргументов через регистры до сих пор не стандартизована и вряд ли когда будет стандартизована вообще? Ответ: кем бы она могла быть стандартизована? Комитетами по стандартизации C и C++? Нет, конечно! Все платформеннозависимые решения оставляются на откуп разработчикам компиляторов — каждый из них волен реализовывать их по-своему или не реализовывать вообще. *«Хорошо, уговорили, — не согласится иной читатель, — но что мешает разработчикам компиляторов одной конкретной платформы договориться об общих соглашениях. Ведь договорились же они передавать возвращенное функцией значение через [E]AX:[E]DX, хотя стандарт о конкретных регистрах вообще никакого понятия не имеет».*

Ну, отчасти разработчики и договорись: большинство производителей 16-рядных компиляторов придерживалось общих соглашений (хотя об этом не сильно трубили вслух), но без претензий на совместимость друг с другом. Быстрый

вызов — он на то и называется быстрым, чтобы обеспечить максимальную производительность. Техника же оптимизации не стоит на месте, и вводить стандарт — это все равно что привязывать гирию к ноге. С другой стороны, средний выигрыш от передачи аргументов через регистры составляет единичные проценты, вот многие разработчики компиляторов и отказываются от быстроты в пользу простоты (реализации). К тому же, если так критична производительность, используйте встраиваемые функции.

Впрочем, все эти рассуждения интересны в первую очередь программистам, исследователей же программ волнует не эффективность, а восстановление прототипов функций. Можно ли узнать, какие аргументы принимает `fastcall`-функция, не анализируя ее код (т. е. смотря только на вызываемую функцию). Чрезвычайно популярный ответ: *«Нет, это невозможно, поскольку компилятор передает аргументы в наиболее «удобных» регистрах»* неправилен, и говорящий наглядно демонстрирует свое полное незнание техники компиляции.

Существует такой термин, как «единица трансляции». В зависимости от реализации компилятор может либо транслировать весь текст программы целиком (что весьма накладно, так как придется хранить в памяти все дерево синтаксического разбора), либо транслировать каждую функцию по отдельности, сохраняя в памяти лишь ее имя и ссылку на сгенерированный для нее код. Компиляторы первого типа крайне редки, во всяком случае, для ОС Windows я не встречал ни одного такого C/C++ компилятора (хотя и слышал о таких). Компиляторы второго типа более производительны, требуют меньше памяти, проще в реализации, словом, всем хороши, за исключением органической неспособности к сквозной оптимизации, — каждая функция оптимизируется персонально и независимо от другой. Поэтому подобрать оптимальные регистры для передачи аргументов компилятор не может, поскольку он не знает, как с ними манипулирует вызываемая функция. Поскольку функции транслируются независимо, им приходится придерживаться общих соглашений, даже если это и невыгодно.

Таким образом, зная «почерк» конкретного компилятора, восстановить прототип функции можно без труда.

Borland C++ 3.x — передача аргументов осуществляется через регистры: AX (AL), DX (DL), BX (BL), а когда регистры кончаются, аргументы начинают засылаться в стек, заносясь в него слева направо и выталкиваясь самой вызываемой функцией (а la `stdcall`).

Схема передачи аргументов довольно интересна — компилятор не закрепляет за каждым аргументом «своих» регистров, вместо этого он предоставляет свободный доступ каждому из них к «стопке» кандидатов, уложенных в порядке предпочтения. Каждый аргумент снимает со стопки столько регистров, сколько ему нужно, а когда стопка исчерпается, тогда придется отправляться в стек. Исключение составляет тип `long int`, всегда передаваемый через DX:AX (причем в DX передается старшее слово), а если это невозможно — то через стек.

Если каждый аргумент занимает не более 16 бит (как обычно и происходит), то первый слева аргумент помещается в AX (AL), второй — в DX (DL), третий — в BX (BL). Если же первый слева аргумент представляет тип `long int`, он снимает со

стопки сразу два регистра — DX:AX, тогда второму аргументу остается регистр BX (BL), а третьему и вовсе ничего (и тогда он передается через стек). Когда же long int передается вторым аргументом, он отправляется в стек, так как необходимый ему регистр AX уже занят первым аргументом, третий же аргумент передается через DX. Наконец, будучи третьим слева аргументом, long int идет в стек, а первые два аргумента передаются через AX (AL) и DX (DL) соответственно.

Передача дальних указателей и вещественных значений всегда осуществляется через основной стек (а не стек сопроцессора, как иногда приходится слышать и как подсказывает здравый смысл).

Таблица 2. Порядок предпочтений Borland C++ 3.x при передаче аргументов по соглашению fastcall

Тип	Предпочтения		
	1-й	2-й	3-й
char	AL	DL	BL
int	AX	DX	BX
long int	DX:AX		
Ближний указатель	AX	DX	BX
Дальний указатель	stack		
float	stack		
double	stack		

Microsoft C++ 6.0 ведет себя аналогично компилятору Borland C++ 3.x, за исключением того, что изменяет порядок предпочтений кандидатов для передачи указателей, выдвигая на первое место BX. И это правильно, ибо ранние микропроцессоры 80x86 не поддерживали косвенную адресацию ни через AX, ни через DX и переданное функции значение все равно приходилось перепихивать либо в BX, либо в SI или DI.

Таблица 3. Порядок предпочтений Microsoft C++ 6.x при передаче аргументов по соглашению fastcall

Тип	Предпочтения		
	1-й	2-й	3-й
char	AL	DL	BL
int	AX	DX	BX
long int	DX:AX		
Ближний указатель	BX	AX	DX
Дальний указатель	stack		
float	stack		
double	stack		

Borland C++ 5.x очень похож на своего предшественника — компилятор Borland C++ 3.x, за исключением того, что вместо регистра BX отдает предпочтение регистру CX и аргументы типа *int* и *long int* помещает в любой из подходящих 32-разрядных регистров, а не DX:AX. Как, впрочем, и следовало ожидать при переводе компилятора с 16- на 32-разрядный режим.

Таблица 4. Порядок предпочтений Borland C++ 5.x при передаче аргументов по соглашению *fastcall*

Тип	Предпочтения		
	1-й	2-й	3-й
char	AL	DL	CL
int	EAX	EDX	ECX
long int	EAX	EDX	ECX
Ближний указатель	EAX	EDX	ECX
Дальний указатель	stack		
float	stack		
double	stack		

Microsoft Visual C++ 4.x — 6.x: при возможности передает первый слева аргумент в регистре ECX, второй — в регистре EDX, а все остальные через стек. Вещественные значения и дальние указатели всегда передаются через стек. Аргумент типа *__int64* (нестандартный тип, 64-разрядное целое, введенный Microsoft) всегда передается через стек.

Если *__int64* — первый слева аргумент, то второй аргумент передается через ECX, а третий — через EDX. Соответственно если *__int64* — второй аргумент, то первый передается через ECX, а третий — через EDX.

Таблица 5. Порядок предпочтений Microsoft Visual C++ 4.x — 6.x при передаче аргументов по соглашению *fastcall*

Тип	Предпочтения		
	1-й	2-й	3-й
char	CL	DL	—
int	ECX	EDX	—
__int64	stack		
long int	ECX		—
Ближний указатель	ECX	EDX	—
Дальний указатель	stack		—
float	stack		—
double	stack		—

WATCOM C. Компилятор от WATCOM сильно отличается от компиляторов от Borland и Microsoft. В частности, он не поддерживает ключевого слова *fastcall* (что, кстати, приводит к серьезным проблемам совместимости), но по умолчанию всегда стремится передавать аргументы через регистры. Вместо общепринятой «стопки предпочтений» WATCOM жестко закрепляет за каждым аргументом свой регистр: за первым — EAX, за вторым — EDX, за третьим — EBX, за четвертым — ECX, причем, если какой-то аргумент в указанный регистр поместить не удастся, он и **все остальные аргументы, находящиеся правее его**, помещает в стек! В частности, типы float и double по умолчанию помещаются в стек основного процессора, что «портит всю малину»!

Таблица 6. Схема передачи аргументов компилятором WATCOM по умолчанию

Тип	Аргумент			
	1-й	2-й	3-й	4-й
char	AL	DL	BL	CL
int	EAX	EDX	EBX	ECX
long int	EAX	EDX	EBX	ECX
Ближний указатель	ECX	EDX	EBX	ECX
Дальний указатель	stack	stack	stack	stack
float	stack CPU	stack CPU	stack CPU	stack CPU
	stack FPU	stack FPU	stack FPU	stack FPU
double	stack CPU	stack CPU	stack CPU	stack CPU
	stack FPU	stack FPU	stack FPU	stack FPU

При желании программист может вручную задать собственный порядок передачи аргументов, прибегнув к прагме `aux`, имеющий следующий формат: `#pragma aux имя функции parm [перечень регистров]`. Список допустимых регистров для каждого типа аргументов приведен в табл. 7.

Несколько пояснений: во-первых, аргументы типа `char` передаются не в 8-, а в 32-разрядных регистрах, во-вторых, бросается в глаза неожиданно большое число возможных пар регистров для передачи дальнего указателя, причем сегмент может передаваться не только в сегментных регистрах, но и 16-разрядных регистрах общего назначения.

Вещественные аргументы могут передаваться через стек сопроцессора — для этого достаточно лишь указать 8087 вместо названия регистра и обязательно скомпилировать программу с ключом `-7` (или `-fpi`, `-fpi87`), показывая компилятору, что инструкции сопроцессора разрешены. В документации по WATCOM сообщается, что аргументы типа `double` могут также передаваться и через пары 32-разрядных регистров общего назначения, но мне, увы, не удалось заставить компилятор генерировать такой код. Может быть, я плохо знаю WATCOM или глюк какой произошел. Также мне не встречалось ни одной программы, в которой ве-

вещественные значения передавались бы через регистры общего назначения. Впрочем, это уже никому не нужные тонкости. (Подробнее о передаче вещественных аргументов рассказывается в одноименном разделе данной главы.)

Таким образом, при исследовании программ, откомпилированных компилятором WATCOM, необходимо быть готовыми к тому, что аргументы могут передаваться практически в любых регистрах, какие заблагорассудится программисту.

Таблица 7. Допустимые регистры для передачи различных типов аргументов в WATCOM C

Тип	Допустимые регистры					
char	EAX	EBX	ECX	EDX	ESI	EDI
int	EAX	EBX	ECX	EDX	ESI	EDI
long int	EAX	EBX	ECX	EDX	ESI	EDI
Ближний указатель	EAX	EBX	ECX	EDX	ESI	EDI
Дальний указатель	DX:EAX	CX:EBX	CX:EAX	CX:ESI	DX:EBX	DI:EAX
	CX:EDI	DX:ESI	DI:EBX	SI:EAX	CX:EDX	DX:EDI
	DI:ESI	SI:EBX	BX:EAX	FS:ECX	FS:EDX	FS:EDI
	FS:ESI	FS:EBX	FS:EAX	GS:ECX	GS:EDX	GS:EDI
	GS:ESI	GS:EBX	GS:EAX	DS:ECX	DS:EDX	DS:EDI
	DS:ESI	DS:EBX	DS:EAX	ES:ECX	ES:EDX	ES:EDI
	ES:ESI	ES:EBX	ES:EAX			
float	8087	???	???	???	???	???
double	8087	EDX:EAX	ECX:EBX	ECX:EAX	ECX:ESI	EDX:EBX
	EDI:EAX	ECX:EDI	EDX:ESI	EDI:EBX	ESI:EAX	ECX:EDX
	EDX:EDI	EDI:ESI	ESI:EBX	EBX:EAX		

Идентификация передачи и приема регистров. Поскольку вызываемая и вызывающая функция вынуждены придерживаться общих соглашений при передаче аргументов через регистры, компилятору приходится помещать аргументы в те регистры, в каких их ожидает вызываемая функция, а не в какие ему «удобно». В результате перед каждой функцией, следующей соглашению `fastcall`, появляется код, «тасующий» содержимое регистров строго определенным образом. Каким — это уже зависит от конкретного компилятора. Наиболее популярные схемы передачи аргументов уже были рассмотрены выше, не будем здесь возвращаться к этому вопросу. Если же ваш компилятор отсутствует в списке (что вполне вероятно, компиляторы сейчас растут как грибы после дождя), попробуйте установить его «характер» экспериментальным путем самостоятельно или взгляните в документацию. Вообще-то разработчики, за редкими исключениями, не раскрывают подобных тонкостей (причем даже не из-за желания сохранить это в тайне, просто если документировать каждый байт компилятора, полный комплект

документации не поместится и на поезд), но, быть может, вам повезет. Если же нет — не беда (см. раздел «Техника исследования характера передачи аргументов компилятором»).

Анализом кода вызывающей функции не всегда можно распознать передачу аргументов через регистры (ну разве что их инициализация будет слишком наглядна), поэтому приходится обращаться непосредственно к вызываемой функции. Регистры, сохраняемые в стеке сразу после получения управления функцией, в подавляющем большинстве случаев не являются регистрами, передающими аргументы, и из списка «кандидатов» их можно вычеркнуть. Смотрим среди оставшихся, есть ли такие, содержимое которых используется без явной инициализации. В первом приближении через эти регистры функция и принимает аргументы. При детальном же рассмотрении проблемы всплывает несколько оговорок. Во-первых, через регистры могут передаваться (и очень часто передаются) неявные аргументы функции — указатель `this`, указатели на виртуальные таблицы объекта и т. д. Во-вторых, если криворукий программист, надеясь, что значение переменной после объявления должно быть равно нулю, забывает об инициализации, а компилятор помещает ее в регистр, то при анализе программы она может быть принята за аргумент функции, передаваемый через регистр. Самое интересное, что этот регистр может по случайному стечению обстоятельств явно инициализироваться вызывающей функцией. Пусть, например, программист перед этим вызывал некоторую функцию, возвращаемого значения которой (помещаемого компилятором в EAX) не использовал, а компилятор поместил неинициализированную переменную в EAX. Причем, если функция при своем нормальном завершении возвращает ноль (как часто и бывает), все может работать... Чтобы выловить такого жука, исследователю придется проанализировать алгоритм, действительно ли в EAX помещается код успешности завершения функции, или же имеет место наложение переменных? Впрочем, если откинуть клинические случаи, передача аргументов через регистры не сильно усложняет анализ, в чем мы сейчас и убедимся.

Практическое исследование механизма передачи аргументов через регистры. Для закрепления всего вышесказанного давайте рассмотрим следующий пример. Обратите внимание на директивы условной компиляции, вставленные для совместимости с различными компиляторами:

Листинг 70

```
#include <stdio.h>
#include <string>

#ifdef __BORLANDC__ || defined (_MSC_VER)
// Эта ветка компилируется только компиляторами Borland C++ и Microsoft C++,
// поддерживающими ключевое слово fastcall
__fastcall
#endif

// Функция MyFunc с различными типами аргументов для демонстрации механизма
// их передачи
```

```

MyFunc(char a, int b, long int c, int d)
{
    #if defined(__WATCOMC__)
        // А эта ветка специально предназначена для WATCOM C.
        // Прагма aux принудительно задает порядок передачи аргументов через
        // следующие регистры: EAX ESI EDI EBX
        #pragma aux MyFunc parm [EAX] [ESI] [EDI] [EBX];
    #endif
        return a+b+c+d;
}

main()
{
    printf("%x\n", MyFunc(0x1, 0x2, 0x3, 0x4));
    return 0;
}

```

Результат компиляции этого примера компилятором Microsoft Visual C++ 6.0 должен выглядеть так:

Листинг 71

```

main                proc near                ; CODE XREF: start+AFp
    push            ebp
    mov             ebp, esp

    push            4
    push            3
    ; Аргументы, которым не хватило регистров, передаются через стек, заносясь
    ; туда справа налево, и вычищает их оттуда вызываемая функция
    ; (т. е. все происходит как по stdcall-соглашению)

    mov             edx, 2
    ; Через EDX передается второй слева аргумент.
    ; Легко определить его тип - это int.
    ; То есть это явно не char, но и не указатель
    ; (два - странное значение для указателя)

    mov             cl, 1
    ; Через cl передается первый слева аргумент типа char
    ; (лишь у переменных типа char размер 8 битов)
    ;

    call            MyFunc
    ; Уже можно восстановить прототип функции MyFunc(char, int, int, int)
    ; Да, мы ошиблись и тип long int приняли за int, но, поскольку в компиляторе
    ; Microsoft Visual C++ эти типы идентичны, такой ошибкой можно пренебречь

    push            eax
    ; Передаем полученный результат функции printf

    push            offset asc_406030        ; "%x\n"
    call            _printf
    add             esp, 8

```

```

        xor     eax, eax
        pop     ebp
        retn

main     endp

MyFunc   proc near          ; CODE XREF: main+Ep

var_8    = dword ptr -8
var_4    = byte ptr -4

arg_0    = dword ptr 8
arg_4    = dword ptr 0Ch
; Через стек функции передавались лишь два аргумента, и их успешно распознала IDA

        push    ebp
        mov     ebp, esp
        sub     esp, 8
        ; Резервируем 8 байтов для локальных переменных

        mov     [ebp+var_8], edx
        ; Регистр EDX не был явно инициализирован до того загрузки в
        ; локальную переменную var_8. Значит, он используется для передачи аргументов!
        ; Поскольку эта программа была скомпилирована компилятором Microsoft Visual C,
        ; а он, как известно, передает аргументы в регистрах ECX:EDX, можно сделать
        ; вывод, что мы имеем дело со вторым, считая слева, аргументом функции
        ; и где-то ниже по тексту нам должно встретиться обращение к ECX - первому
        ; слева аргументу функции.
        ; (хотя необязательно - первый аргумент функцией может и не использоваться)

        mov     [ebp+var_4], cl
        ; Действительно, обращение к CL не заставило долго себя ждать.
        ; Поскольку через CL передается тип char, то, вероятно,
        ; первый аргумент функции - char.
        ; Некоторая неуверенность вызвана тем, что функция может просто обращаться
        ; к младшему байту аргумента типа int, скажем.
        ; Однако, посмотрев на код вызывающей функции, мы можем убедиться, что
        ; функции передается именно char, а не int.
        ; Попутно отметим глупость компилятора: стоило ли передавать аргументы через
        ; регистры, чтобы тут же заслать их в локальные переменные!
        ; Ведь обращение к памяти сжигает всю выгоду от быстрого вызова!
        ; Такой "быстрый" вызов быстрым даже язык не поворачивается назвать.

        movsx   eax, [ebp+var_4]
        ; В EAX загружается первый слева аргумент, переданный через CL, типа char
        ; со знаковым расширением до двойного слова. Значит, это signed char
        ; (т. е. char по умолчанию для Microsoft Visual C++)

        add     eax, [ebp+var_8]
        ; Складываем EAX со вторым слева аргументом

        add     eax, [ebp+arg_0]
        ; Складываем результат предыдущего сложения с третьим слева аргументом,
        ; переданным через стек...

        add     eax, [ebp+arg_4]
        ; и все это складываем с четвертым аргументом, также переданным через стек.

```



```

mov     esp, ebp
pop     ebp
; Закрываем кадр стека

retn    8
; Чистим за собой стек, как и положено по fastcall-соглашению

```

MyFunc endp

А теперь сравним это с результатом компиляции Borland C++:

Листинг 72

```

; int __cdecl main(int argc,const char **argv,const char *envp)
_main      proc near           ; DATA XREF: DATA:00407044o

argc       = dword ptr 8
argv       = dword ptr 0Ch
envp       = dword ptr 10h

push       ebp
mov        ebp, esp

push       4
; Передаем аргумент через стек. Скопив глаза вниз, мы обнаруживаем явную
; инициализацию регистров ECX, EDX, AL. Для четвертого аргумента регистров
; не хватило, и его пришлось передавать через стек. Значит, четвертый слева
; аргумент функции - 0x4

mov        ecx, 3
mov        edx, 2
mov        al, 1
; Этот код не может быть ничем иным, как передачей аргументов через регистры

call       MyFunc

push       eax
push       offset unk_407074    ; format
call       _printf
add        esp, 8

xor        eax, eax

pop        ebp
retn

_main      endp

MyFunc     proc near           ; CODE XREF: _main+11p

arg_0      = dword ptr 8
; Через стек функции передавался лишь один аргумент

push       ebp
mov        ebp, esp
; Открываем кадр стека

movsx     eax, al
; Borland сгенерировал более оптимальный код, чем Microsoft, не помещая
; регистр в локальную переменную и экономя тем самым память. Впрочем, если бы

```

```

; был задан соответствующий ключ оптимизации, Microsoft Visual C++ поступил
; точно так же.
; Обратите внимание еще и на то, что Borland обрабатывает аргументы
; в выражениях слева направо в порядке их перечисления в прототипе функции,
; в то время как Microsoft Visual C++ поступает наоборот.

```

```

add     edx, eax
add     ecx, edx
; Регистры EDX и CX не были инициализированы, значит, в них функции были
; переданы аргументы.

mov     edx, [ebp+arg_0]
; Загружаем в EDX последний аргумент функции, переданный через стек...

add     ecx, edx
; складываем еще раз

mov     eax, ecx
; Передаем в EAX (в EAX функция возвращает результат своего завершения)

pop     ebp
retn    4
; Вычищаем за собой стек

```

```
MyFunc     endp
```

Наконец, результат компиляции WATCOM C должен выглядеть так:

Листинг 73

```

main_      proc near                ; CODE XREF: __CMain+40p
    push    18h
    call    __CHK
    ; Проверка стека на переполнение

    push    ebx
    push    esi
    push    edi
    ; Сохраняем регистры в стеке

    mov     ebx, 4
    mov     edi, 3
    mov     esi, 2
    mov     eax, 1
    ; Смотрите, аргументы передаются через те регистры, которые мы указали!
    ; Более того, отметьте, что первый аргумент типа char передается через
    ; 32-разрядный регистр EAX! Такое поведение WATCOM'a чрезвычайно
    ; затрудняет восстановление прототипов функций! В данном случае присвоение
    ; регистрам значений происходит согласно порядку объявления аргументов
    ; в прототипе функции, считая справа. Но так, увы, бывает далеко не всегда.

    call    MyFunc

    push    eax
    push    offset unk_420004
    call    printf_

    add     esp, 8

```

```

        xor     eax, eax
        pop     edi
        pop     esi
        pop     ebx
        retn

main_      endp

MyFunc     proc near                ; CODE XREF: main_+21p
; Функция не принимает через стек ни одного аргумента

        push    4
        call    __CHK

        and     eax, 0FFh
; Обнуление старших двадцати четырех битов вкупе с обращением к регистру
; до его инициализации наводит на мысль, что через EAX передается тип char.
; Какой это аргумент, мы сказать не можем, увы...

        add     esi, eax
; Регистр ESI не был инициализирован нашей функцией, следовательно, через
; него передается аргумент типа int. Можно предположить, что это второй
; слева аргумент в прототипе функции, так как (если ничто не препятствует)
; регистры в вызывающей функции инициализируются согласно их порядку
; перечисления в прототипе, считая справа, а выражения вычисляются
; слева направо.
; Разумеется, подлинный порядок следования аргументов не критичен, но
; все-таки приятно, если удастся его восстановить

        lea     eax, [esi+edi]
; Опаньки, выдерем Тигре хвост с корнем! Вы думаете, что в EAX загружается
; указатель? А ESI и EDI переданные функции – также указатели? EAX с его
; типом char становится очень похожим на индекс...
; Увы! Компилятор WATCOM слишком хитер, и при анализе программ,
; скомпилированных с его помощью, очень легко впасть в грубые ошибки.
; Да, EAX – это указатель, в том смысле, что LEA используется для вычисления
; суммы ESI и EDI, но обращения к памяти по этому указателю не происходит
; ни в вызывающей, ни в вызываемой функции. Следовательно, аргументы функции
; не указатели, а константы!

        add     eax, ebx
; Аналогично EDX содержит в себе аргумент, переданный функции.
; Итак, прототип функции должен выглядеть так:
; MyFunc(char a, int b, int c, int d)
; Однако порядок следования аргументов может быть и иным...

        retn

MyFunc     endp

```

Как мы видим, в передаче аргументов через регистры ничего особенно сложного нет, можно даже восстановить подлинный прототип вызываемой функции. Однако ситуация, рассмотренная выше, достаточно идеализирована, и в реальных программах передача одних лишь непосредственных значений встречается редко. Давайте же теперь, освоившись с быстрыми вызовами, дизассемблируем более трудный пример:

Листинг 74. Трудный пример с fastcall

```

#if defined(__BORLANDC__) || defined (_MSC_VER)
__fastcall
#endif
MyFunc(char a, int *b, int c)
{
#if defined(__WATCOMC__)
#pragma aux MyFunc parm [EAX] [EBX] [ECX];
#endif

    return a+b[0]+c;
}

main()
{
    int a=2;
    printf("%x\n", MyFunc(strlen("1"), &a, strlen("333")));
}

```

Результат компиляции Microsoft Visual C++ должен выглядеть так:

Листинг 75

```

main                proc near                ; CODE XREF: start+AFp
var_4                = dword ptr -4

    push    ebp
    mov     ebp, esp
    ; Открываем кадр стека

    push    ecx
    push    esi
    ; Сохраняем регистры в стеке

    mov     [ebp+var_4], 2
    ; Присваиваем локальной переменной var_4 типа int значение 2.
    ; Тип определяется на основе того, что переменная занимает 4 байта
    ; (подробнее об этом см. "Идентификация локальных стековых переменных").

    push    offset a333                ; const char *
    ; Передаем функции strlen указатель на строку "333".
    ; Аргументы функции MyFunc, как и положено, передаются справа налево

    call    _strlen
    add     esp, 4

    push    eax
    ; Здесь либо мы сохраняем возвращенное функцией значение в стеке,
    ; либо передаем его следующей функции.

    lea     esi, [ebp+var_4]
    ; В ESI заносим указатель на локальную переменную var_4

    push    offset a1                ; const char *
    ; Передаем функции strlen указатель на строку "1"

    call    _strlen

```

```

add     esp, 4

mov     cl, al
; Возвращенное значение копируется в регистр CL, а ниже инициализируется EDX.
; Поскольку ECX:EDX используются для передачи аргументов fastcall-функциям,
; инициализация этих двух регистров перед вызовом функции явно неслучайна!
; Можно предположить, что через CL передается крайний левый аргумент типа char

mov     edx, esi
; В ESI содержится указатель на var_4, следовательно, второй аргумент функции
; типа int, заносимый в EDX, передается по ссылке.

call    MyFunc
; Предварительный прототип функции выглядит так:
; MyFunc(char *a, int *b, inc c)
; Откуда взялся аргумент c? А помните, выше в стек был затолкнут EAX и
; ни до вызова функции, ни после так и не вытолкнут? Впрочем, чтобы
; убедиться в этом окончательно, требуется посмотреть, сколько байтов со стека
; снимает вызываемая функция.
; Обратите также внимание и на то, что значения, возвращенные функцией strlen,
; не заносилось в локальные переменные, а передавались непосредственно MyFunc.
; Это наводит на мысль, что исходный код программы выглядел так:
; MyFunc(strlen("1"), &var_4, strlen("333"));
; Хотя, впрочем, не факт, компилятор мог оптимизировать код, выкинув
; локальную переменную, если она нигде более не используется. Однако,
; во-первых, судя по коду вызываемой функции, компилятор работает без
; оптимизации, а во-вторых, если значения, возвращенные функциями strlen,
; используются один-единственный раз в качестве аргумента MyFunc, то помещение
; их в локальную переменную – большая глупость, только затуманивающая суть
; программы. Тем более что исследователю важно не восстановить подлинный
; исходный код, а понять его алгоритм.

push    eax
push    offset asc_406038      ; "%x\n"
call    _printf
add     esp, 8

pop     esi

mov     esp, ebp
pop     ebp
; Закрываем кадр стека

retn

main     endp

MyFunc   proc near              ; CODE XREF: main+2Ep

var_8    = dword ptr -8
var_4    = byte ptr -4
arg_0    = dword ptr 8
; Функция принимает один аргумент, – значит, это и есть тот EAX, занесенный в стек

push    ebp
mov     ebp, esp
; Открываем кадр стека

```

```

sub     esp, 8
; Резервируем восемь байтов под локальные переменные

mov     [ebp+var_8], edx
; Поскольку EDX используется без явной инициализации, очевидно,
; через него передается второй слева аргумент функции
; (согласно соглашению fastcall-компилятора Microsoft Visual C++).
; Из анализа кода вызывающей функции мы уже знаем,
; что в EDX помещается указатель на var_4, следовательно,
; var_8 теперь содержит указатель на var_4.

mov     [ebp+var_4], cl
; Через CL передается самый левый аргумент функции типа char и тут же
; заносится в локальную переменную var_4.

movsx   eax, [ebp+var_4]
; Переменная var_4 расширяется до signed int.

mov     ecx, [ebp+var_8]
; В регистр ECX загружается содержимое указателя var_8, переданного через EDX.
; Действительно, как мы помним, через EDX функции передавался указатель.

add     eax, [ecx]
; Складываем EAX (хранит первый слева аргумент функции) с содержимым
; ячейки памяти, на которую указывает указатель ECX (второй слева аргумент).

add     eax, [ebp+arg_0]
; А вот и обращение к тому аргументу функции, что был передан через стек

mov     esp, ebp
pop     ebp
; Закрываем кадр стека

ret     4
; Функции был передан один аргумент через стек

```

MyFunc endp

Просто? Просто! Тогда рассмотрим результат творчества Borland C++, который должен выглядеть так:

Листинг 76

```

; int __cdecl main(int argc,const char **argv,const char *envp)
_main      proc near          ; DATA XREF: DATA:00407044o

var_4      = dword ptr -4
argc       = dword ptr 8
argv       = dword ptr 0Ch
envp       = dword ptr 10h

push       ebp
mov        ebp, esp
; Открываем кадр стека

push       ecx
; Сохраняем ECX... Пойдите! Это что-то новое! В прошлых примерах Borland
; никогда не сохранял ECX при входе в функцию. Очень похоже, что через ECX

```

```
; функции был передан какой-то аргумент и теперь она передает его другой
; функции через стек.
; Увы, каким бы убедительным такое решение ни выглядело, оно неверно!
; Компилятор просто резервирует под локальные переменные четыре байта. Почему?
; Из чего это следует? Смотрите: IDA распознала одну локальную переменную var_4,
; но память под нее явно не резервировалась, во всяком случае, команды SUB ESP,4
; не было. Постой-ка, постой, но ведь PUSH ECX как раз и приводит к уменьшению
; регистра ESP на четыре! Ох, уж эта оптимизация!

mov     [ebp+var_4], 2
; Заносим в локальную переменную значение 2

push    offset a333 ; s
; Передаем функции strlen указатель на строку 333

call    _strlen
pop     ecx
; Вытаскиваем аргумент из стека

push    eax
; Здесь либо мы передаем возвращенное функцией strlen значение следующей
; функции как стековый аргумент, либо временно сохраняем EAX в стеке
; (позже выяснится, что справедливо последнее предположение)

push    offset a1 ; s
; Передаем функции strlen указатель на строку 1

call    _strlen
pop     ecx
; Вытаскиваем аргумент из стека

lea     edx, [ebp+var_4]
; Загружаем в EDX смещение локальной переменной var_4

pop     ecx
; Что-то вытаскиваем из стека, но что именно? Прокручивая экран
; дизассемблера вверх, находим, что последним в стек заносился EAX,
; содержащий значение, возвращенное функцией strlen(333).
; Теперь оно помещается в регистр ECX
; (как мы помним, Borland передает через него второй слева аргумент).
; Попутно отметим для любителей fastcall'a: не всегда он приводит к ожидаемому
; ускорению вызова - у Intel 80x86 слишком мало регистров и их то и дело
; приходится сохранять в стеке.
; Передача аргумента через стек потребовала бы всего одного обращения: PUSH EAX,
; здесь же мы наблюдаем два - PUSH EAX и POP ECX!

call    MyFunc
; При восстановлении прототипа функции не забудьте о регистре EAX - он
; не инициализируется явно, но хранит значение, возвращенное последним вызовом
; strlen. Поскольку компилятор Borland C++ 5.x использует следующий список
; предпочтений: EAX, EDX, ECX, - можно сделать вывод, что в EAX передается первый
; слева аргумент функции, а два остальных - в EDX и ECX соответственно.
; Обратите внимание и на то, что Borland C++, в отличие от Microsoft Visual C++,
; обрабатывает аргументы не в порядке их перечисления, а сначала вычисляет
; значение всех функций, "выдергивая" их справа налево, и только
; потом переходит к переменным и константам.
```

```

; И в этом есть свой здравый смысл - функции
; изменяют значение многих регистров общего назначения и, до тех пор пока не
; будет вызвана последняя функция, нельзя приступить к передаче аргументов
; через регистры.

push    eax
push    offset asc_407074      ; format
call    _printf
add     esp, 8

xor     eax, eax
; Возвращаем нулевое значение

pop     ecx
pop     ebp
; Закрываем кадр стека

ret     0
_main   endp

MyFunc   proc near              ; CODE XREF: _main+26p
push     ebp
mov      ebp, esp
; Открываем кадр стека

movsx    eax, al
; Расширяем EAX до знакового двойного слова

mov      edx, [edx]
; Загружаем в EDX содержимое ячейки памяти, на которую указывает указатель EDX

add      eax, edx
; Складываем первый аргумент функции с переменной типа int, переданной
; вторым аргументом по ссылке

add      ecx, eax
; Складываем третий аргумент типа int с результатом предыдущего сложения

mov      eax, ecx
; Помещаем результат обратно в EAX
; Глупый компилятор, не проще ли было переставить местами аргументы предыдущей
; команды?

pop      ebp
; Закрываем кадр стека

ret     0
MyFunc   endp

```

А теперь рассмотрим результат компиляции того же примера компилятором WATCOM C, у которого всегда есть чему поучиться:

Листинг 77

```

main_    proc near              ; CODE XREF: __CMain+40p

var_C    = dword ptr -0Ch
; Локальная переменная

```



```
push    18h
call    __CHK
; Проверка стека на переполнение

push    ebx
push    ecx
; Сохранение модифицируемых регистров.
; Или, быть может, резервирование памяти под локальные переменные?

sub     esp, 4
; Вот это уж точно явное резервирование памяти под одну локальную переменную,
; следовательно, две команды PUSH, находящиеся выше, действительно сохраняют
; регистры.

mov     [esp+0Ch+var_C], 2
; Занесение в локальную переменную значения 2

mov     eax, offset a333      ; "333"
call    strlen_
; Обратите внимание, WATCOM передает функции strlen указатель на строку
; через регистр!

mov     ecx, eax
; Возвращенное функцией значение копируется в регистр ECX.
; WATCOM знает, что следующий вызов strlen не портит этот регистр!

mov     eax, offset a1        ; "1"
call    strlen_

and     eax, 0FFh
; Поскольку strlen возвращает тип int, здесь имеет место явное преобразование
; типов: int -> char

mov     ebx, esp
; В EBX заносится указатель на переменную var_C

call    MyFunc
; Какие же аргументы передавались функции? Во-первых, EAX - вероятно, крайний
; левый аргумент, во-вторых, EBX - явно инициализированный перед вызовом
; функции и, вполне возможно, ECX, хотя последнее и не обязательно.
; ECX может содержать и регистровую переменную, но в таком случае вызываемая
; функция не должна к нему обращаться.

push    eax
push    offset asc_42000A     ; "%x\n"

call    printf_

add     esp, 8
add     esp, 4
; А еще говорят, что WATCOM - оптимизирующий компилятор! А вот две команды
; объединить в одну он, увы, не смог!

pop     ecx
pop     ebx

retn

main_    endp
```

```

MyFunc      proc near          ; CODE XREF: main_+33p
    push     4
    call     __CHK
    ; Проверка стека

    and      eax, 0FFh
    ; Повторное обнуление 24 старших битов. WATCOM'у следовало бы определиться,
    ; где выполнять эту операцию - в вызываемой или вызывающей функции, но зато
    ; подобный "дублеж" упрощает восстановление прототипов функций

    add      eax, [ebx]
    ; Складываем EAX типа char и теперь расширенное до int с переменной типа int
    ; переданной по ссылке через регистр EBX

    add      eax, ecx
    ; Ага, вот оно, обращение к ECX, следовательно, этот регистр использовался
    ; для передачи аргументов

    retn
    ; Таким образом, прототип функции должен выглядеть так:
    ; MyFunc(char EAX, int *EBX, int ECX)
    ; Обратите внимание, что восстановить его удалось лишь совместным анализом
    ; вызываемой и вызывающей функций!

MyFunc      endp

```

Передача вещественных значений. Кодоломатели в своей массе не очень-то разбираются в вещественной арифметике, избегая ее как огня. Между тем в ней нет ничего сверхсложного и освоить управление сопроцессором можно буквально за полтора-два дня. Правда, с математическими библиотеками, поддерживающими вычисления с плавающей запятой, справиться намного труднее (особенно если IDA не распознает имен их функций), но какой компилятор сегодня пользуется библиотеками? Микропроцессор и сопроцессор монтируются на одном кристалле, и сопроцессор, начиная с 80486DX (если мне не изменяет память), доступен всегда, поэтому прибегать к его программной эмуляции нет никакой нужды.

До конца девяностых годов среди хакеров бытовало мнение, что можно всю жизнь прожить, но так и не столкнуться с вещественной арифметикой. Действительно, в старые добрые времена процессоры в своей медлительности ни в чем не уступали черепахам, сопроцессоры имелись не у всех, а задачи, стоящие перед компьютерами, допускали (не без ухищрений, правда) решения и в целочисленной арифметике.

Теперь все кардинально изменилось. Вычисления с плавающей точкой, выполняемые сопроцессором параллельно с работой основной программы, даже быстрее целочисленных вычислений, обчитываемых основным процессором. И программисты, окрыленные такой перспективой, стали лепить вещественные типы данных даже там, где раньше с лихвой хватало целочисленных. (Например, если $a = b / c * 100$, то, изменив порядок вычислений $a = b * 100 / c$, мы можем обойтись и типами int.) Современным исследователям программ без знания команд сопроцессора очень трудно обойтись.

Сопроцессоры 80x87 поддерживают три вещественных типа данных: **короткий** 32-битный, **длинный** 64-битный и **расширенный** 80-битный, соответствующие следующим типам языка C: *float*, *double* и *long double*. (Внимание! Стандарт ANSI C не оговаривает точного представления указанных выше типов, и это утверждение справедливо только для платформы PC, да и то не для всех реализаций.)

Таблица 8. Основная информация о вещественных типах сопроцессоров 80x87

Тип	Размер	Диапазон значений	Предпочтительные типы передачи
float	4 байта	$10^{-38} \dots 10^{+38}$	Регистры CPU, стек CPU, стек FPU
double	8 байт	$10^{-308} \dots 10^{+308}$	Регистры CPU, стек CPU, стек FPU
long double	10 байт	$10^{-4932} \dots 10^{+4932}$	Стек CPU, стек FPU
real*	6 байт	$2,9 \cdot 10^{-39} \dots 1,7 \cdot 10^{+38}$	Регистры CPU, стек CPU, стек FPU
* Тип Turbo-Pascal			

Аргументы типа float и double могут быть переданы функции тремя различными способами: через *регистры общего назначения основного процессора*, через стек основного процессора и через стек сопроцессора. Аргументы типа long double потребовали бы для своей передачи слишком много регистров общего назначения, поэтому в подавляющем большинстве случаев они заталкиваются в стек основного процессора или сопроцессора.

Первые два способа передачи нам уже знакомы, а вот третий — это что-то новенькое! Сопроцессор 80x87 имеет восемь 80-битных регистров, обозначаемых ST(0), ST(1), ST(2), ST(3), ST(4), ST(5), ST(6) и ST(7), организованных в форме кольцевого стека. Это обозначает, что большинство команд сопроцессора не оперируют номерами регистров, а в качестве приемника (источника) используют вершину стека. Например, чтобы сложить два вещественных числа, сначала необходимо затолкнуть их в стек сопроцессора, а затем вызывать команду сложения, суммирующую два числа, лежащих на вершине стека, и возвращающую результат своей работы опять-таки через стек. Существует возможность сложить число, лежащее в стеке сопроцессора, с числом, находящимся в оперативной памяти, но непосредственно сложить два числа из оперативной памяти невозможно!

Таким образом, первый этап операций с вещественными типами — записывание их в стек сопроцессора. Эта операция осуществляется командами из серии FLDxx, перечисленных с краткими пояснениями в табл. 9. В подавляющем большинстве случаев используется инструкция «FLD источник», заталкивающая в стек сопроцессора вещественное число из оперативной памяти или регистра сопроцессора. Строго говоря, это не одна команда, а четыре команды в одной упаковке с опкодами 0xD9 0x0?, 0xDD 0x0?, 0xDB 0x0? и 0xD9 0xCi, для загрузки *короткого*, *длинного*, *расширенного* типов и *регистра FPU* соответственно, где ? — адресное поле, уточняющее, где в регистре или памяти находится операнд; а i — индекс регистра FPU.

Отсутствие возможности загрузки вещественных чисел из регистров CPU обесмысливает их использование для передачи аргументов типа `float`, `double` или `long double`. Все равно, чтобы затолкать эти аргументы в стек сопроцессора, вызываемая функция будет вынуждена скопировать содержимое регистров в оперативную память. Как ни крути, от обращения к памяти не избавишься. Вот поэтому-то регистровая передача вещественных типов крайне редка, и в подавляющем большинстве случаев они, как и обычные аргументы, передаются через стек основного процессора или через стек сопроцессора. (Последнее умеют только продвинутые компиляторы, в частности WATCOM, но не Microsoft Visual C++ и не Borland C++.)

Впрочем, некоторые «избранные» значения могут загружаться и без обращений к памяти, в частности, существуют команды для заталкивания в стек сопроцессора чисел ноль, один, π и некоторых других — полный список приведен в табл. 9.

Таблица 9. Основные команды сопроцессора, применяющиеся для передачи/приема аргументов

Команда	Назначение
FLD источник	Заталкивает вещественное число из <i>источника</i> на вершину стека сопроцессора
FSTP приемник	Выталкивает вещественное число из вершины стека сопроцессора в приемник
FST приемник	Копирует вещественное число из вершины стека сопроцессора в <i>приемник</i>
FLDZ	Заталкивает ноль на вершину стека сопроцессора
FLD1	Заталкивает единицу на вершину стека сопроцессора
FLDPI	Заталкивает на вершину стека сопроцессора число π
FLDL2T	Заталкивает на вершину стека сопроцессора двоичный логарифм десяти
FLDL2E	Заталкивает на вершину стека сопроцессора двоичный логарифм числа e
FLDLG2	Заталкивает на вершину стека сопроцессора десятичный логарифм двух
FLDLN2	Заталкивает на вершину стека сопроцессора натуральный логарифм двух
FILD источник	Заталкивает целое число из <i>источника</i> на вершину стека сопроцессора
FIST приемник	Копирует целое число с вершины стека сопроцессора в <i>приемник</i>
FISTP приемник	Выталкивает целое число с вершины стека сопроцессора в <i>приемник</i>

Команда	Назначение
FBLD источник	Заталкивает десятичное число из <i>приемника</i> на вершину стека сопроцессора
FBSTP приемник	Копирует десятичное число с вершины стека сопроцессора в <i>приемник</i>
FXCH ST (индекс)	Обмен значениями между вершиной стека сопроцессора и регистром ST (<i>индекс</i>)

Любопытной особенностью сопроцессора является поддержка операций с целочисленными вычислениями. Мне не известно ни одного компилятора, использующего эту возможность, но такой прием иногда встречается в ассемблерных вставках, поэтому, пренебрегать изучением целочисленных команд сопроцессора все же не стоит.

Типы *double* и *long double* занимают более одного машинного слова и через стек основного процессора передаются за несколько итераций. Это приводит к тому, что анализ кода вызываемой функции не всегда позволяет установить количество и тип передаваемых вызываемой функции аргументов. Выход — в исследовании алгоритма работы вызываемой функции. Поскольку сопроцессор не может самостоятельно определить тип операнда, находящегося в памяти (т. е. не знает, сколько ячеек он занимает), за каждым типом закрепляется своя команда. Синтаксис ассемблера скрывает эти различия, позволяя программисту абстрагироваться от тонкостей реализации (а еще говорят, что ассемблер — язык низкого уровня), и мало кто знает, что FADD [float] и FADD [double] — это *разные* машинные инструкции с опкодами 0xD8 ??000??? и 0xDC ??000??? соответственно. Плохая новость, помет Тигры! Анализ дизассемблерного листинга не дает никакой информации о вещественных типах, для получения этой информации приходится спускаться на машинный уровень, вгрызаясь в шестнадцатеричные дампы инструкций.

В табл. 10 приведены опкоды основных команд сопроцессора, работающих с памятью. Обратите внимание, что с вещественными значениями типа *long double* непосредственные математические операции невозможны — прежде их необходимо загрузить в стек сопроцессора.

Таблица 10. Опкоды основных команд сопроцессора

Команда	Тип		
	Короткий (float)	Длинный (double)	Расширенный (long double)
FLD	0xD9 ??000???	0xDD ??000???	0xDB ??101???
FSTP	0xD9 ??011???	0xDD ??011???	0xDB ??111???
FST	0xD9 ??010???	0xDD ??010???	Нет
FADD	0xD8 ??000???	0xDC ??000???	Нет
FADDP	0xDE ??000???	0xDA ??000???	Нет

Команда	Тип		
	Короткий (float)	Длинный (double)	Расширенный (long double)
FSUB	0xD8 ??100???	0xDC ??100???	Нет
FDIV	0xD8 ??110???	0xDC ??110???	Нет
FMUL	0xD* ??001???	0xDC ??001???	Нет
FCOM	0xD8 ??010???	0xDC ??010???	Нет
FCOMP	0xD8 ??011???	0xDC ??011???	Нет
* Второй байт опкода представлен в двоичном виде. Знак вопроса обозначает любой бит.			

Замечание о вещественных типах языка Turbo-Pascal. Вещественные типы языка C вследствие его машиноориентированности совпадают с вещественными типами сопроцессора, что логично. Основным же вещественный тип языка Turbo-Pascal — Real, занимает 6 байтов и противоестествен для машины. Поэтому при вычислениях через сопроцессор он программно переводится в тип Extended (long double в терминах C). Это съедает львиную долю производительности, но других типов встроенная математическая библиотека, призванная заменить собой сопроцессор, увы, не поддерживает. При наличии же «живого» сопроцессора появляются чисто процессорные типы Single, Double, Extended и Comp, соответствующие float, double, long double и __int64.

Функциям математической библиотеки, обеспечивающим поддержку вычислений с плавающей запятой, вещественные аргументы передаются через регистры: в AX, BX, DX помещается первый слева аргумент, в CX, SI, DI — второй (если он есть). Системные функции сопряжения с интерфейсом процессора (в частности, функции преобразования Real в Extended) принимают аргументы через регистры, а результат возвращают через стек сопроцессора. Наконец, прикладные функции и процедуры получают вещественные аргументы через стек основного процессора.

В зависимости от настроек компилятора программа может компилироваться либо с использованием встроенной математической библиотеки (по умолчанию), либо с непосредственным вызовом команд сопроцессора (ключ N\$+). В первом случае программа вообще не использует возможности сопроцессора, даже если он и установлен на машине. Во втором же при наличии сопроцессора возлагает все вычислительные возможности на него, а если он отсутствует, попытка вызова сопроцессорных команд приводит к генерации основным процессором исключения int 0x7. Его отлавливает программный эмулятор сопроцессора — фактически та же самая встроенная библиотека поддержки вычислений с плавающей точкой.

Что ж, теперь мы в общих чертах представляем себе, как происходит передача вещественных аргументов, и горим нетерпением увидеть, как это происходит живьем. Для начала возьмем тривиальный пример:

Листинг 78. Демонстрация передачи функции вещественных аргументов

```
#include <stdio.h>

float MyFunc(float a, double b)
{
    #if defined(__WATCOMC__)
    #pragma aux MyFunc parm [8087];
    // Компилировать с ключом -7
    #endif
    return a+b;
}

main()
{
    printf("%f\n", MyFunc(6.66, 7.77));
}
```

Результат компиляции Microsoft Visual C++ должен выглядеть так:

Листинг 79

```
main          proc near          ; CODE XREF: start+AFp
var_8         = qword ptr -8
; Локальная переменная, занимающая, судя по всему, 8 байтов

    push      ebp
    mov       ebp, esp
    ; Открываем кадр стека

    push      401F147Ah
    ; К сожалению, IDA не может представить операнд в виде числа с плавающей запятой.
    ; К тому же у нас нет возможности определить, что это именно вещественное число.
    ; Его тип может быть каким угодно: и int, и указателем
    ; (кстати, оно очень похоже на указатель).

    push      0E147AE14h
    push      40D51EB8h
    ; "Черновой" вариант прототипа выглядит так: MyFunc(int a, int b, int c)

    call      MyFunc
    add       esp, 4
    ; Хвост Тигра! Со стека снимается только одно машинное слово, тогда как
    ; ложится туда три!

    fstp      [esp+8+var_8]
    ; Стягиваем со стека сопроцессора какое-то вещественное число. Чтобы узнать
    ; какое, придется нажать <ALT-0>, выбрать в открывшемся меню пункт
    ; Text representation и в нем в окно Number of opcode bytes ввести,
    ; сколько знакомест отводится под опкод команд, например, 4.
    ; Тут же слева от FSTP появляется ее машинное представление - DD 1C 24.
    ; По табл. 10 определяем тип данных, с которым манипулирует эта команда.
    ; Это - double. Следовательно, функция возвратила через стек сопроцессора
    ; вещественное значение. Раз функция возвращает вещественные значения, вполне
    ; возможно, что она их и принимает в качестве аргументов. Однако без анализа
    ; MyFunc подтвердить это предположение невозможно.
```

```

push    offset aF ; "%f\n"
; Передаем функции printf указатель на строку спецификаторов, предписывая ей
; вывести одно вещественное число. Но при этом мы его не заносим в стек!
; Как же так?! Прокручиваем окно дизассемблера вверх, параллельно с этим
; обдумывая все возможные пути разрешения ситуации.
; Внимательно рассматривая команду FSTP [ESP+8+var_8], попытаемся вычислить,
; куда же она помещает результат своей работы.
; IDA определила var_8 как qword ptr -8, следовательно, [ES+8-8] эквивалентно
; [ESP], т. е. вещественная переменная стягивается прямо на вершину стека.
; А что у нас на вершине? Два аргумента, переданных MyFunc и так и не
; вытолкнутых из стека. Какой хитрый компилятор! Он не стал создавать локальную
; переменную, а использовал аргументы функции для временного хранилища данных!

call    _printf
add     esp, 0Ch
; Вытаскиваем со стека три машинных слова

pop     ebp
retn

main    endp

MyFunc   proc near          ; CODE XREF: sub_401011+12p

var_4    = dword ptr -4
arg_0    = dword ptr 8
arg_4    = qword ptr 0Ch
; Смотрим, IDA обнаружила только два аргумента, в то время как функции передавалось
; три машинных слова! Очень похоже, что один из аргументов занимает 8 байтов...

push     ebp
mov      ebp, esp
; Открываем кадр стека

push     ecx
; Нет, это не сохранение ECX, это резервирование памяти под локальную
; переменную. Так как на том месте, где лежит сохраненный ECX, находится
; переменная var_4.

fld      [ebp+arg_0]
; Затягиваем на стек сопроцессора вещественную переменную, лежащую по адресу
; [ebp+8] (первый слева аргумент). Чтобы узнать тип этой переменной, посмотрим
; опкод инструкции FLD - D9 45 08. Ага, D9, - значит, float.
; Выходит, первый слева аргумент - float.

fadd     [ebp+arg_4]
; Складываем arg_0 типа float со вторым слева аргументом типа... Вы думаете,
; раз первый был float, то и второй также будет float'ом?
; А вот и не обязательно! Лезем в опкод - DC 45 0C, значит, второй аргумент
; double, а не float!

fst      [ebp+var_4]
; Копируем значение с верхушки стека сопроцессора
; (там лежит результат сложения) в локальную переменную var_4.
; Зачем? Ну мало ли, вдруг бы она потребовалась?
; Обратите внимание, значение не стягивается, а копируется! То есть, оно все еще
; остается в стеке. Таким образом, прототип функции MyFunc выглядел так:

```



```

        ; double MyFunc(float a, double b);

mov     esp, ebp
pop     ebp
; Закрываем кадр стека

ret     0

MyFunc   endp

```

Поскольку результат компиляции Borland C++ 5.x практически в точности идентичен уже рассмотренному выше примеру от Microsoft Visual C++ 6.x, не будем терять на него время и сразу перейдем к разбору WATCOM C (как всегда, у WATCOM'a есть чему поучиться):

Листинг 80

```

main_      proc near                ; CODE XREF: __CMain+40p
var_8       = qword ptr -8
; локальная переменная на 8 байт

        push    10h
        call    __CHK
        ; Проверка стека на переполнение

        fld     ds:dbl_420008
        ; Закидываем на вершину стека сопроцессора переменную типа double,
        ; взимая ее из сегмента данных.
        ; Тип переменной успешно определила сама IDA, предварив его префиксом dbl.
        ; А если бы не определила, тогда бы мы обратились к опкоду команды FLD.

        fld     ds:flt_420010
        ; Закидываем на вершину стека сопроцессора переменную типа float

        call    MyFunc
        ; Вызываем MyFunc с передачей двух аргументов через стек сопроцессора,
        ; значит, ее прототип выглядит так: MyFunc(float a, double b).

        sub     esp, 8
        ; Резервируем место для локальной переменной размером в 8 байтов

        fstp    [esp+8+var_8]
        ; Стыкаем с вершины стека вещественное типа double
        ; (тип определяется размером переменной).

        push    offset unk_420004
        call    printf_
        ; Ага, уже знакомый нам трюк передачи var_8 функции printf!

        add     esp, 0Ch
        ret     0

main_      endp

MyFunc     proc near                ; CODE XREF: main_+16p
var_C      = qword ptr -0Ch
var_4      = dword ptr -4
; IDA нашла две локальные переменные

```

```

push    10h
call    __CHK

sub     esp, 0Ch
; Резервируем место под локальные переменные

fstp    [esp+0Ch+var_4]
; Стягиваем с вершины стека сопроцессора вещественное значение типа float
; (оно, как мы помним, было занесено туда последним).
; На всякий случай, впрочем, можно удостовериться в этом, посмотрев опкод
; команды FSTP - D9 5C 24 08. Ну, раз 0xD9, значит, точно float.

fstp    [esp+0Ch+var_C]
; Стягиваем с вершины стека сопра вещественное значение типа double
; (оно, как мы помним, было занесено туда перед float).
; На всякий случай удостоверяемся в этом, взглянув на опкод команды FSTP.
; Он и есть DD 1C 24. 0xDD - раз 0xDD, значит, действительно double.

fld     [esp+0Ch+var_4]
; Затаскиваем на вершину стека наш float обратно и...

fadd    [esp+0Ch+var_C]
; складываем его с нашим double. Вот, а еще говорят, что WATCOM C
; оптимизирующий компилятор! Трудно же с этим согласится, раз компилятор
; не знает, что от перестановки слагаемых сумма не изменяется!

add     esp, 0Ch
; Освобождаем память, ранее выделенную для локальных переменных
retn

MyFunc      endp

dbl_420008   dq 7.77                ; DATA XREF: main_+A↑r
flt_420010   dd 6.6599998            ; DATA XREF: main_+10↑r

```

Настала очередь компилятора Turbo-Pascal for Windows 1.0. Наберем в текстовом редакторе следующий пример:

Листинг 81. Демонстрация передачи вещественных значений компилятором Turbo-Pascal for Windows 1.0

```

USES WINCRT;

Procedure MyProc(a:Real);
begin
    WriteLn(a);
end;

VAR
    a: Real;
    b: Real;

BEGIN
    a:=6.66;
    b:=7.77;
    MyProc(a+b);
END.

```

А теперь, тяпнув с Тигрой пивка для храбрости, откомпилируем его без поддержки сопроцессора (так и происходит с настройками по умолчанию).

Листинг 82

```
PROGRAM      proc near

    call      INITTASK
    call      @__SystemInit$qv ; __SystemInit(void)
    ; Инициализация модуля SYSTEM

    call      @__WINCRTInit$qv ; __WINCRTInit(void)
    ; Инициализация модуля WINCRT

    push      bp
    mov       bp, sp
    ; Открываем кадр стека

    xor       ax, ax
    call      @__StackCheck$q4Word ; Stack overflow check (AX)
    ; Проверяем, есть ли в стеке хотя бы ноль свободных байтов

    mov       word_2030, 0EC83h
    mov       word_2032, 0B851h
    mov       word_2034, 551Eh
    ; Инициализируем переменную типа Real.
    ; Что это именно Real, мы пока, конечно,
    ; знаем только лишь из исходного текста программы.
    ; Визуально отличить эту серию команд
    ; от трех переменных типа Word невозможно.

    mov       word_2036, 3D83h
    mov       word_2038, 0D70Ah
    mov       word_203A, 78A3h
    ; Инициализируем другую переменную типа Real

    mov       ax, word_2030
    mov       bx, word_2032
    mov       dx, word_2034
    mov       cx, word_2036
    mov       si, word_2038
    mov       di, word_203A
    ; Передаем через регистры две переменные типа Real

    call      @$brplu$q4Realt1 ; Real(AX:BX:DX)+=Real(CX:SI:DI)
    ; К счастью, IDA "узнала" в этой функции оператор сложения и даже
    ; подсказала нам ее прототип. Без ее помощи нам вряд ли удалось понять,
    ; что делает эта очень длинная и запутанная функция.

    push      dx
    push      bx
    push      ax
    ; Передаем возвращенное значение процедуре MyProc через стек,
    ; следовательно, ее прототип выглядит так: MyProc(a:Real).

    call      MyProc
```

```

        pop        bp
        ; Закрываем кадр стека

        xor        ax, ax
        call       @Halt$q4Word    ; Halt(Word)
        ; Прерываем выполнение программы

PROGRAM      endp

MyProc       proc near                ; CODE XREF: PROGRAM+5Cp

arg_0        = word ptr 4
arg_2        = word ptr 6
arg_4        = word ptr 8
; Три аргумента, переданные процедуре, как мы уже выяснили,
; на самом деле представляют собой
; три "дольки" одного аргумента типа Real.

        push      bp
        mov       bp, sp
        ; Открываем кадр стека

        xor       ax, ax
        call      @_StackCheck$q4Word ; Stack overflow check (AX)
        ; Есть ли в стеке ноль байтов?

        mov       di, offset unk_2206
        push      ds
        push      di
        ; Заталкиваем в стек указатель на буфер для вывода строки

        push      [bp+arg_4]
        push      [bp+arg_2]
        push      [bp+arg_0]
        ; Заталкиваем все три полученных аргумента в стек

        mov       ax, 11h
        push      ax
        ; Ширина вывода - 17 символов

        mov       ax, 0FFFFh
        push      ax
        ; Число точек после запятой - max

        call      @Write$qm4Text4Real4Wordt3 ; Write(var f; v: Real; width, decimals: Word)
        ; Выводим вещественное число в буфер unk_2206

        call      @WriteLn$qm4Text ; WriteLn(var f: Text)
        ; Выводим строку из буфера на экран

        call      @__IOCheck$qv ; Exit if error
        pop       bp
        retn      6
MyProc      endp

```

А теперь, используя ключ `/ $N+`, задействуем команды сопроцессора и посмотрим, как это скажется на коде:

Листинг 83

```
PROGRAM      proc near

    call      INITTASK
    call      @__SystemInit$qv ; __SystemInit(void)
    ; Инициализируем модуль System

    call      @__InitEM86$qv ; Initialize software emulator
    ; Врубаем эмулятор сопроцессора

    call      @__WINCRTInit$qv ; __WINCRTInit(void)
    ; Инициализируем модуль WINCRT

    push      bp
    mov       bp, sp
    ; Открываем кадр стека

    xor       ax, ax
    call      @__StackCheck$q4Word ; Stack overflow check (AX)
    ; Проверка стека на переполнение

    mov       word_21C0, 0EC83h
    mov       word_21C2, 0B851h
    mov       word_21C4, 551Eh
    mov       word_21C6, 3D83h
    mov       word_21C8, 0D70Ah
    mov       word_21CA, 78A3h
    ; Пока мы не можем определить тип инициализируемых переменных.
    ; Это с равным успехом может быть и WORD, и Real

    mov       ax, word_21C0
    mov       bx, word_21C2
    mov       dx, word_21C4
    call      @Extended$q4Real ; Convert Real to Extended
    ; А вот теперь мы передаем word_21C0, word_21C2 и word_21C4 функции,
    ; преобразующий Real в Extend с загрузкой последнего в стек сопроцессора,
    ; значит, word_21C0 - word_21C4 - это переменная типа Real.

    mov       ax, word_21C6
    mov       bx, word_21C8
    mov       dx, word_21CA
    call      @Extended$q4Real ; Convert Real to Extended
    ; Аналогично word_21C6 - word_21CA - переменная типа Real

    wait
    ; Ждем-с, пока сопроцессор не закончит свою работу

    faddp     st(1), st
    ; Складываем два числа типа extended, лежащих на вершине стека сопроцессора
    ; с сохранением результата в том же самом стеке.

    call      @Real$q8Extended ; Convert Extended to Real
    ; Преобразуем Extended в Real.
    ; Аргумент передается через стек сопроцессора, а возвращается в
    ; регистрах AX BX DX.

    push      dx
```

```

        push    bx
        push    ax
        ; Регистры AX, BX и DX содержат значение типа Real,
        ; следовательно, прототип процедуры выглядит так:
        ; MyProc(a:Real);

        call    MyProc

        pop     bp
        xor     ax, ax
        call    @Halt$q4Word ; Halt(Word)
PROGRAM endp

MyProc      proc near                ; CODE XREF: PROGRAM+6Dp
arg_0       = word ptr 4
arg_2       = word ptr 6
arg_4       = word ptr 8
; Как мы уже помним, эти три аргумента на самом деле один аргумент типа Real

        push    bp
        mov     bp, sp
        ; Открываем кадр стека

        xor     ax, ax
        call    @__StackCheck$q4Word ; Stack overflow check (AX)
        ; Проверка стека на переполнение

        mov     di, offset unk_2396
        push    ds
        push    di
        ; Заносим в стек указатель на буфер для вывода строки

        mov     ax, [bp+arg_0]
        mov     bx, [bp+arg_2]
        mov     dx, [bp+arg_4]
        call    @Extended$q4Real ; Convert Real to Extended
        ; Преобразуем Real в Extended

        mov     ax, 17h
        push    ax
        ; Ширина вывода 0x17 знаков

        mov     ax, 0FFFFh
        push    ax
        ; Количество знаков после запятой - все, что есть, все и выводить

        call    @Write$qm4Text8Extended4Wordt3 ; Write(var f; v: Extended{st(0); width decimals: Word)
        ; Вывод вещественного числа со стека сопроцессора в буфер

        call    @WriteLn$qm4Text ; WriteLn(var f: Text)
        ; Печать строки из буфера

        call    @__IOCheck$qv ; Exit if error
        pop     bp
        retn    6
MyProc      endp

```

Соглашения о вызовах *thiscall* и соглашения о вызове по умолчанию.

В C++ программах каждая функция объекта неявно принимает аргумент *this* — указатель на экземпляр объекта, из которого вызывается функция. Подробнее об этом уже рассказывалось в главе «Идентификация аргумента *this*», поэтому не будем здесь повторяться.

По умолчанию все известные мне C++ компиляторы используют комбинированное соглашение о вызовах, передавая явные аргументы через стек (если только функция не объявлена как *fastcall*), а указать *this* через регистр с наибольшим предпочтением (см. табл. 2—7).

Соглашения же *cdecl* и *stdcall* предписывают передать все аргументы через стек, включая неявный аргумент *this*, заносимый в стек в последнюю очередь — после всех явных аргументов (другими словами, *this* — самый левый аргумент).

Рассмотрим следующий пример:

Листинг 84. Демонстрация передачи неявного аргумента *this*

```
#include <stdio.h>

class MyClass{
public:
    void          demo(int a);
    // прототип demo в действительности выглядит так: demo(this, int a)

    void __stdcall demo_2(int a, int b);
    // прототип demo_2 в действительности выглядит так: demo_2(this, int a, int b)

    void __cdecl demo_3(int a, int b, int c);
    // прототип demo_2 в действительности выглядит так: demo_2(this, int a, int b, int c)
};

// Реализация функции demo, demo_2, demo_3 для экономии места опущена

main()
{
    MyClass *zzz = new MyClass;
    zzz->demo();
    zzz->demo_2();
    zzz->demo_3();
}
```

Результат компиляции этого примера компилятором Microsoft Visual C++ 6.0 должен выглядеть так (показана лишь функция *main*, все остальное не представляет на данный момент никакого интереса):

Листинг 85

```
main          proc near          ; CODE XREF: start+AFp
    push      esi
    ; Сохраняем ESI в стеке

    push      1
    call      ??2@YAPAXI@Z ; operator new(uint)
    ; Выделяем один байт для экземпляра объекта
```

```

mov     esi, eax
; ESI содержит указатель на экземпляр объекта

add     esp, 4
; Выталкиваем аргумент из стека

mov     ecx, esi
; Через ECX функции Demo передается указатель this.
; Как мы помним, компилятор Microsoft Visual C++ использует регистр ECX
; для передачи самого первого аргумента функции.
; В данном случае этим аргументом и является указатель this.
; А компилятор Borland C++ 5.x передал бы this через регистр EAX, так как
; он отдает ему наибольшее предпочтение (см. табл. 4)

push    1
; Заносим в стек явный аргумент функции. Значит, это не fastcall-функция,
; иначе бы данный аргумент был помещен в регистр EDI. Выходит,
; мы имеем дело с типом вызова по умолчанию.

call    Demo

push    2
; Затапливаем в стек первый справа аргумент

push    1
; Затапливаем в стек второй справа аргумент

push    esi
; Затапливаем в стек неявный аргумент this.
; Такая схема передачи говорит о том, что имело место явное преобразование
; типа функции в stdcall или cdecl. Прокручивая экран дизассемблера немного
; вниз, мы видим, что стек вычищает вызываемая функция, значит, она следует
; соглашению stdcall.

call    demo_2

push    3
push    2
push    1
push    esi
call    sub_401020
add     esp, 10h
; Раз функция вычищает за собой стек сама, то она имеет либо тип по умолчанию,
; либо – cdecl. Передача указателя this через стек подсказывает, что истинно
; второе предположение.

xor     eax, eax
pop     esi
retn

main    endp

```

Аргументы по умолчанию. Для упрощения вызова функций с «хороводом» аргументов в язык C++ была введена возможность задания аргументов по умолчанию. Отсюда возникает вопрос: отличается ли чем-нибудь вызов функций с аргументами по умолчанию от обычных функций? И кто инициализирует опущенные аргументы — вызываемая или вызывающая функция?

Так вот, при вызове функций с аргументами по умолчанию компилятор самостоятельно добавляет недостающие аргументы, и вызов такой функции ничем не отличается от вызова обычных функций.

Докажем это на следующем примере:

Листинг 86. Демонстрация передачи аргументов по умолчанию

```
#include <stdio.h>

MyFunc(int a=1, int b=2, int c=3)
{
    printf("%x %x %x\n",a,b,c);
}

main()
{
    MyFunc();
}
```

Результат его компиляции будет выглядеть приблизительно так (для экономии места показана только вызывающая функция):

Листинг 87

```
main          proc near          ; CODE XREF: start+AFp
               push             ebp
               mov              ebp, esp
               push             3
               push             2
               push             1
               ; Как видно, все опущенные аргументы были переданы функции
               ; самим компилятором
               call             MyFunc
               add              esp, 0Ch
               pop              ebp
               retn
main          endp
```

Техника исследования механизма передачи аргументов неизвестным компилятором. Огромное многообразие существующих компиляторов и постоянное появление новых не позволяет привести здесь всеохватывающую таблицу, расписывающую характер каждого из компиляторов. Как же быть, если вам попадается программа, откомпилированная компилятором, не освещенным в данной книге?

Если компилятор удастся опознать (например, с помощью IDA или по текстовым строкам, содержащимся в файле), остается только раздобыть его экземпляр и прогнать через него серию тестовых примеров с передачей «подопытной» функции аргументов различного типа. Нелишне изучить прилагаемую к компилятору документацию, возможно, там будут хотя бы кратко описаны все поддерживаемые им механизмы передачи аргументов.

Хуже, когда компилятор не опознается или же достать его копию нет никакой возможности. Тогда придется кропотливо и тщательно исследовать взаимодействие вызываемой и вызывающей функций.

Идентификация значения, возвращаемого функцией

...каждый язык — это своя философия, свой взгляд на деятельность программиста, отражение определенной технологии программирования.

Кауфман

Традиционно под значением, возвращаемым функцией, понимается значение, возвращенное оператором `return`, однако это лишь надводная часть айсберга, не раскрывающая всей картины взаимодействия функций друг с другом. В качестве наглядной демонстрации рассмотрим довольно типичный пример, кстати позаимствованный из реального кода программы:

Листинг 88. Демонстрация возвращения значения в аргументе, переданном по ссылке

```
int xdiv(int a, int b, int *c=0)
{
    if (!b) return -1;
    if (c) c[0]=a % b;
    return a / b;
}
```

Функция `xdiv` возвращает результат целочисленного деления аргумента **a** на аргумент **b**, но помимо этого записывает в переменную **c**, переданную по ссылке, остаток. Так сколько же значений вернула функция? И чем возвращение результата по ссылке хуже или «незаконнее» классического `return`?

Популярные издания склонны упрощать проблему идентификации значения, возвращенного функцией, рассматривая один лишь частный случай с оператором `return`. В частности, так поступает Мэтт Питтерек в своей книге *«Секреты системного программирования в Windows 95»*, оставляя все остальные способы «за кадром».

Мы же рассмотрим следующие механизмы:

- возврат значения оператором `return` (через регистры или стек сопроцессора);
- возврат значений через аргументы, переданные по ссылке;
- возврат значений через динамическую память (кучу);
- возврат значений через глобальные переменные;
- возврат значений через флаги процессора.

Вообще-то к этому списку не помешало бы добавить возврат значений через дисковые и проецируемые в память файлы, но это выходит за рамки обсуждаемой

темы (хотя, рассматривая функцию как «черный ящик» с входом и выходом, нельзя не признать, что вывод функцией результатов своей работы в файл, фактически и есть возвращаемое ею значение).

Возврат значения оператором *return*. По общепринятому соглашению значение, возвращаемое оператором *return*, помещается в регистр EAX (в AX в 16-разрядном режиме), а если его разрядности оказывается недостаточно, старшие 32 бита операнда помещаются в EDX (в 16-разрядном режиме старшее слово помещается в DX).

Вещественные типы в большинстве случаев возвращаются через стек сопроцессора, реже — через регистры EDX:EAX (DX:AX — в 16-разрядном режиме).

А как возвращаются типы, занимающие более 8 байтов? Скажем, некая функция возвращает структуру, состоящую из сотен байтов или объект не меньшего размера. Ни то ни другое в регистры не запишешь, даже стека сопроцессора не хватит!

Таблица 11. Механизм возвращения значения оператором *return* в 16-разрядных компиляторах

Тип	Способ возврата	
Однобайтовый	AL	AX
Двухбайтовый	AX	
Четырехбайтовый	DX:AX	
real	DX:BX:AX	
float	DX:AX	Стек сопроцессора
double	Стек сопроцессора	
near pointer	AX	
far pointer	DX:AX	
Свыше четырех байт	Через неявный аргумент по ссылке	

Таблица 12. Механизм возвращения значения оператором *return* в 32-разрядных компиляторах

Тип	Способ возврата		
Однобайтовый	AL	AX	EAX
Двухбайтовый	AX		EAX
Четырехбайтовый	EAX		
Восьмибайтовый	EDX:EAX		
float	Стек сопроцессора		EAX
double	Стек сопроцессора		EDX:EAX
near pointer	EAX		
Свыше восьми байт	Через неявный аргумент по ссылке		

Оказывается, если возвращаемое значение не может быть втиснуто в регистры, компилятор скрыто от программиста передает функции неявный аргумент — ссылку на локальную переменную, в которую и записывается возвращенный результат. Таким образом, функции *struct mystuct MyFunc(int a, int b)* и *void MyFunc(struct mystrect *my, int a, int b)* компилируются в **идентичный** (или близкий к тому) код и «вытянуть» из машинного кода подлинный прототип **невозможно!**

Единственную зацепку дает компилятор Microsoft Visual C++, возвращающий в этом случае указатель на возвращаемую переменную, т. е. восстановленный прототип выглядит приблизительно так: *struct mystrect* MyFunc(struct mystrect* my, int a, int b)*. Согласитесь, несколько странно, чтобы программист в здравом уме да при живой теще возвращал указатель на аргумент, который своими руками только что и передал функции? Компилятор же Borland C++ в данной ситуации возвращает тип *void*, стирая различие между аргументом, возвращаемым по значению и аргументом, возвращаемым по ссылке. Впрочем, невозможность восстановления подлинного прототипа не должна огорчать. Скорее наоборот! Подлинный прототип утверждает, что результат работы функции возвращается по значению, а в действительности он возвращается по ссылке! Так ради чего называть кошку мышкой?

Пара слов об определении типа возвращаемого значения. Если функция при выходе явно присваивает регистру EAX или EDX (AX и DX в 16-разрядном режиме) некоторое значение, то его тип можно начерно определить по табл. 11 и 12. Если же оставляет эти регистры неопределенными, то, скорее всего, возвращается тип *void*, т. е. ничто. Уточнить информацию помогает анализ вызывающей функции, а точнее, то, как она обращается с регистрами EAX [EDX] (AX [DX] в 16-разрядном режиме). Например, для типов *char* характерно либо обращение к младшей половине регистра EAX [AX] — регистру AL, либо обнуление старших байтов регистра EAX операцией логического AND. Логично предположить: если вызывающая функция не использует значения, оставленного вызываемой функцией в регистрах EAX [EDX], — ее тип *void*. Но это предположение неверно. Частенько программисты игнорируют возвращаемое значение, вводя исследователей в заблуждение.

Рассмотрим следующий пример, демонстрирующий механизм возвращения основных типов значений:

Листинг 89. Пример, демонстрирующий механизм возвращения основных типов значений

```
#include <stdio.h>
#include <malloc.h>

// Демонстрация возвращения переменной типа char оператором return
char char_func(char a, char b)
{
    return a+b;
}
```

```
// Демонстрация возвращения переменной типа int оператором return
int int_func(int a, int b)
{
    return a+b;
}

// Демонстрация возвращения переменной типа int64 оператором return
__int64 int64_func(__int64 a, __int64 b)
{
    return a+b;
}

// Демонстрация возвращения указателя на int оператором return
// Демонстрация возвращения значения через аргументы, переданные по ссылке
int* near_func(int* a, int* b)
{
    int *c;
    c=(int *)malloc(sizeof(int));
    c[0]=a[0]+b[0];
    return c;
}

main()
{
    int a;
    int b;

    a=0x666;
    b=0x777;

    printf("%x\n",
        char_func(0x1,0x2)+
        int_func(0x3,0x4)+
        int64_func(0x5,0x6)+
        near_func(&a,&b)[0]);
}
```

Результат его компиляции Microsoft Visual C++ 6.0 с настройками по умолчанию будет выглядеть так:

Листинг 90

```
char_func      proc near          ; CODE XREF: main+1Ap
arg_0          = byte ptr 8
arg_4          = byte ptr 0Ch

    push      ebp
    mov       ebp, esp
    ; Открываем кадр стека

    movsx     eax, [ebp+arg_0]
    ; Загружаем в EAX arg_0 типа signed char, попутно расширяя его до int,

    movsx     ecx, [ebp+arg_4]
    ; Загружаем в EAX arg_0 типа signed char, попутно расширяя его до int,

    add       eax, ecx
    ; Складываем arg_0 и arg_4, расширенные до int, сохраняя их в регистре EAX, -
    ; это есть значение, возвращаемое функцией.
    ; К сожалению, достоверно определить его тип невозможно.
```

```

; Он с равным успехом может представлять собой и int, и char,
; причем int даже более вероятен,
; так как сумма двух char по соображениям безопасности должна помещаться в int,
; иначе возможно переполнение.

    pop     ebp
    retn

char_func    endp

int_func     proc near                ; CODE XREF: main+29p

arg_0        = dword ptr 8
arg_4        = dword ptr 0Ch

    push    ebp
    mov     ebp, esp
    ; Открываем кадр стека

    mov     eax, [ebp+arg_0]
    ; Загружаем в EAX значение аргумента arg_0 типа int

    add     eax, [ebp+arg_4]
    ; Складываем arg_0 с arg_4 и оставляем результат в регистре EAX.
    ; Это и есть значение, возвращаемое функцией, вероятнее всего, типа int.
    pop     ebp
    retn

int_func     endp

int64_func   proc near                ; CODE XREF: main+40p

arg_0        = dword ptr 8
arg_4        = dword ptr 0Ch
arg_8        = dword ptr 10h
arg_C        = dword ptr 14h

    push    ebp
    mov     ebp, esp
    ; открываем кадр стека

    mov     eax, [ebp+arg_0]
    ; Загружаем в EAX значение аргумента arg_0

    add     eax, [ebp+arg_8]
    ; Складываем arg_0 с arg_8

    mov     edx, [ebp+arg_4]
    ; Загружаем в EDX значение аргумента arg_4

    adc     edx, [ebp+arg_C]
    ; Складываем arg_4 и arg_C с учетом флага переноса, оставшегося от сложения
    ; arg_0 с arg_8.
    ; Выходит, arg_0 и arg_4, как и arg_8 и arg_C, это половинки двух
    ; аргументов типа __int64, складываемые друг с другом.
    ; Стало быть, результат вычислений возвращается в регистрах EDX:EAX

    pop     ebp
    retn

int64_func   endp

```

```

near_func      proc near                ; CODE XREF: main+54p

var_4          = dword ptr -4
arg_0          = dword ptr 8
arg_4          = dword ptr 0Ch

    push      ebp
    mov      ebp, esp
    ; Отрываем кадр стека

    push      ecx
    ; Сохраняем ECX

    push      4          ; size_t
    call     _malloc
    add      esp, 4
    ; Выделяем 4 байта из кучи

    mov      [ebp+var_4], eax
    ; Заносим указатель на выделенную память в переменную var_4

    mov      eax, [ebp+arg_0]
    ; Загружаем в EAX значение аргумента arg_0

    mov      ecx, [eax]
    ; Загружаем в ECX значение ячейки памяти типа int, на которую указывает
    ; регистр EAX. Таким образом, тип аргумента arg_0 - int *

    mov      edx, [ebp+arg_4]
    ; Загружаем в EDX значение аргумента arg_4

    add      ecx, [edx]
    ; Складываем с *arg_0 значение ячейки памяти типа int, на которое указывает EDX.
    ; Следовательно, тип аргумента arg_4 - int *

    mov      eax, [ebp+var_4]
    ; Загружаем в EAX указатель на выделенный из кучи блок памяти

    mov      [eax], ecx
    ; Копируем в кучу значение суммы *arg_0 и *arg_4

    mov      eax, [ebp+var_4]
    ; Загружаем в EAX указатель на выделенный из кучи блок памяти.
    ; Это и будет значением, возвращаемым функцией, т.е. ее прототип выглядел так:
    ; int* MyFunc(int *a, int *b);

    mov      esp, ebp
    pop      ebp
    retn

near_func      endp

main           proc near                ; CODE XREF: start+AFp

var_8          = dword ptr -8
var_4          = dword ptr -4

    push      ebp
    mov      ebp, esp
    ; Открываем кадр стека

```

```

sub     esp, 8
; Резервируем место для локальных переменных

push    esi
push    edi
; Сохраняем регистры в стеке

mov     [ebp+var_4], 666h
; Заносим в локальную переменную var_4 типа int значение 0x666

mov     [ebp+var_8], 777h
; Заносим в локальную переменную var_8 типа int значение 0x777

push    2
push    1
call    char_func
add     esp, 8
; Вызываем функцию char_func(1,2). Как мы помним, у нас были сомнения в типе
; возвращаемого ею значения - либо int, либо char.

movsx   esi, al
; Расширяем возвращенное функцией значение до signed int, следовательно, она
; возвратила signed char

push    4
push    3
call    int_func
add     esp, 8
; Вызываем функцию int_func(3,4), возвращающую значение типа int

add     eax, esi
; Прибавляем к значению, возвращенному функцией, содержимое ESI

cdq
; Преобразуем двойное слово, содержащееся в регистре EAX в четверное,
; помещаемое в регистр EDX:EAX. Это говорит о том, что тип возвращенного функцией
; значения преобразуется из int в int64. Пока непонятно, для чего и зачем.

mov     esi, eax
mov     edi, edx
; Копируем расширенное четверное слово в регистры EDI:ESI

push    0
push    6
push    0
push    5
call    int64_func
add     esp, 10h
; Вызываем функцию int64_func(5,6), возвращающую тип __int64.
; Теперь становится понятно, чем вызвано расширение предыдущего результата

add     esi, eax
adc     edi, edx
; К четверному слову, содержащемуся в регистрах EDI:ESI, добавляем результат,
; возвращенный функцией int64_func

lea     eax, [ebp+var_8]
; Загружаем в EAX указатель на переменную var_8

```



```
    push    eax
    ; Передаем функции near_func указатель на var_8 как аргумент

    lea     ecx, [ebp+var_4]
    ; Загружаем в ECX указатель на переменную var_4

    push    ecx
    ; Передаем функции near_func указатель на var_4 как аргумент

    call    near_func
    add     esp, 8
    ; Вызываем near_func

    mov     eax, [eax]
    ; Как мы помним, в регистре EAX функция возвратила указатель на переменную
    ; типа int, загружаем значение этой переменной в регистр EAX

    cdq
    ; Расширяем EAX до четверного слова

    add     esi, eax
    adc     edi, edx
    ; Складываем два четверных слова

    push    edi
    push    esi
    ; Результат сложения передаем функции printf

    push    offset unk_406030
    ; Передаем указатель на строку спецификаторов

    call    _printf
    add     esp, 0Ch

    pop     edi
    pop     esi
    mov     esp, ebp
    pop     ebp
    retn

main      endp
```

Как мы видим, в идентификации типа значения, возвращенного оператором `return`, ничего хитрого нет, все прозаично. Но не будем спешить. Рассмотрим следующий пример. Как вы думаете, что именно и в каких регистрах будет возвращаться?

Листинг 91. Пример демонстрирующий возвращения структуры по значению

```
#include <stdio.h>
#include <string.h>

struct XT
{
    char s0[4];
    int x;
};
```

```

struct XT MyFunc(char *a, int b)
// функция возвращает значение типа структура XT по значению
{
    struct XT xt;
    strcpy(&xt.s0[0],a);
    xt.x=b;
    return xt;
}

main()
{
    struct XT xt;
    xt=MyFunc("Hello, Sailor!",0x666);
    printf("%s %x\n",&xt.s0[0],xt.x);
}

```

Заглянем в откомпилированный результат:

Листинг 92

```

MyFunc      proc near          ; CODE XREF: sub_401026+10p
var_8       = dword ptr -8
var_4       = dword ptr -4
; Эти локальные переменные на самом деле элементы "расщепленной" структуры XT
; Как уже говорилось в главе "Идентификация объектов, структур и массивов",
; компилятор всегда стремится обращаться к элементам структуры по их фактическим
; адресам, а не через базовый указатель.
; Поэтому не так-то просто отличить структуру от несвязанных между собой переменных,
; а подчас это и вовсе невозможно!

arg_0       = dword ptr 8
arg_4       = dword ptr 0Ch
; Функция принимает два аргумента

    push     ebp
    mov      ebp, esp
    ; Открываем кадр стека

    sub      esp, 8
    ; Резервируем место для локальных переменных

    mov      eax, [ebp+arg_0]
    ; Загружаем в регистр EAX содержимое аргумента arg_0

    push     eax
    ; Передаем arg_0 функции strcpy, следовательно,
    ; arg_0 представляет собой указатель на строку.

    lea      ecx, [ebp+var_8]
    ; Загружаем в ECX указатель на локальную переменную var_8 и...

    push     ecx
    ; передаем его функции strcpy
    ; Следовательно, var_8 - строковый буфер размером 4 байта

    call     strcpy

```

```

    add     esp, 8
    ; Копируем переданную через arg_0 строку в var_8

    mov     edx, [ebp+arg_4]
    ; Загружаем в регистр EDX значение аргумента arg_4

    mov     [ebp+var_4], edx
    ; Копируем arg_4 в локальную переменную var_4

    mov     eax, [ebp+var_8]
    ; Загружаем в EAX содержимое (не указатель!) строкового буфера

    mov     edx, [ebp+var_4]
    ; Загружаем в EDX значение переменной var_4
    ; Столь явная загрузка регистров EDX:EAX перед выходом из функции указывает
    ; на то, что это и есть значение, возвращаемое функцией.
    ; Надо же какой неожиданный сюрприз! Функция возвращает в EDX и EAX
    ; две переменные различного типа! А вовсе не __int64, как могло бы показаться
    ; при беглом анализе программы.
    ; Второй сюрприз - возврат типа char[4] не через указатель или ссылку, а через
    ; регистр! Нам еще повезло, если бы структура была объявлена как
    ; struct XT{short int a, char b, char c}, в регистре EAX возвратились бы
    ; целых три переменные двух типов!

    mov     esp, ebp
    pop     ebp
    retn

MyFunc      endp

main        proc near                ; CODE XREF: start+AFp

var_8       = dword ptr -8
var_4       = dword ptr -4
; Две локальные переменные типа int.
; Тип установлен путем вычисления размера каждой из них

    push    ebp
    mov     ebp, esp
    ; Открываем кадр стека

    sub     esp, 8
    ; Резервируем восемь байтов под локальные переменные

    push    666h
    ; Передаем функции MyFunc аргумент типа int
    ; Следовательно, arg_4 имеет тип int (по коду вызываемой функции это не было
    ; очевидно, - arg_4 с не меньшим успехом мог оказаться и указателем).
    ; Значит, в регистре EDX функция возвращает тип int

    push    offset aHelloSailor      ; "Hello, Sailor!"
    ; Передаем функции MyFunc указатель на строку
    ; Внимание! Строка занимает более 4 байтов, поэтому не рекомендуется
    ; запускать этот пример "вживую".

    call    MyFunc
    add     esp, 8
    ; Вызываем MyFunc. Она неким образом изменяет регистры EDX и EAX.
    ; Мы уже знаем типы возвращаемых в них значений, и остается только

```

```

; удостовериться, "правильно" ли они используются вызывающей функцией.

mov     [ebp+var_8], eax
; Заносим в локальную переменную var_8 содержимое регистра EAX

mov     [ebp+var_4], edx
; Заносим в локальную переменную var_4 содержимое регистра EDX
; Согласитесь, очень похоже на то, что функция возвращает __int64

mov     eax, [ebp+var_4]
; Загружаем в EAX содержимое var_4
; (т. е. регистра EDX, возвращенного функцией MyFunc) и...

push    eax
; передаем его функции printf.
; Согласно строки спецификаторов, это тип int.
; Следовательно, в EDX функция возвратила int или по крайней мере его
; старшую часть

lea     ecx, [ebp+var_8]
; Загружаем в ECX указатель на переменную var_8, хранящую значение,
; возвращенное функцией через регистр EAX.
; Согласно строки спецификаторов, это указатель на строку
; Итак, мы подтвердили, что типы значений, возвращенных через регистры EDX:EAX,
; различны!
; Немного поразмыслив, мы даже сможем восстановить подлинный прототип:
; struct X{char a[4]; int b} MyFunc(char* c, int d);

push    ecx
push    offset aSX      ; "%s %x\n"
call    _printf
add     esp, 0Ch

mov     esp, ebp
pop     ebp
; Закрываем кадр стека

retn
main    endp

```

А теперь слегка изменим структуру XT, заменив `char s0[4]` на `char9 s0[10]`, что гарантированно не влезает в регистры EDX:EAX, и посмотрим, как изменится от этого код:

Листинг 93

```

main    proc near                ; CODE XREF: start+AFp

var_20   = byte ptr -20h
var_10   = dword ptr -10h
var_C    = dword ptr -0Ch
var_8    = dword ptr -8
var_4    = dword ptr -4

        push    ebp
        mov     ebp, esp
; Отырываем кадр стека

```

```
sub     esp, 20h
; Резервируем 0x20 байтов под локальные переменные

push    666h
; Передаем функции MyFunc крайний правый аргумент - значение 0x666 типа int

push    offset aHelloSailor ; "Hello, Sailor!"
; Передаем функции MyFunc второй справа аргумент - указатель на строку

lea     eax, [ebp+var_20]
; Загружаем в EAX адрес локальной переменной var_20

push    eax
; Передаем функции MyFunc указатель на переменную var_20
; Стоп! Этого аргумента не было в прототипе функции! Откуда же он взялся?!
; Верно, не было. Его вставил компилятор для возвращения структуры по значению.
; Последнюю фразу вообще-то стоило заключить в кавычки для придания ей
; ироничного оттенка - структура, возвращаемая по значению, в действительности
; возвращается по ссылке.

call    MyFunc
add     esp, 0Ch
; Вызываем MyFunc

mov     ecx, [eax]
; Функция в ECX возвратила указатель на возвращенную ей по ссылке структуру.
; Этот прием характерен лишь для Microsoft Visual C++, большинство компиляторов
; оставляют значение EAX на выходе неопределенным или равным нулю.
; Но, так или иначе, в ECX загружается первое двойное слово,
; на которое указывает указатель EAX. На первый взгляд это элемент типа int.
; Однако не будем бежать поперек косы и торопиться с выводами

mov     [ebp+var_10], ecx
; Сохранение ECX в локальной переменной var_10

mov     edx, [eax+4]
; В EDX загружаем второе двойное слово по указателю EDX

mov     [ebp+var_C], edx
; Копируем его в переменную var_C.
; Выходит, что и второй элемент структуры имеет тип int?
; Мы, знающие, как выглядел исходный текст программы, уже начинаем замечать
; подвох. Что-то здесь определенно не так...

mov     ecx, [eax+8]
; Загружаем третье двойное слово, от указателя EAX, и...

mov     [ebp+var_8], ecx
; копируем его в var_8. Еще один тип int? Да откуда же они берутся в таком
; количестве, когда у нас он был только один! И где, собственно, строка?

mov     edx, [eax+0Ch]
mov     [ebp+var_4], edx
; И еще один тип int переносим из структуры в локальную переменную. Нет,
; определенно, это выше наших сил!

mov     eax, [ebp+var_4]
; Загружаем в EAX содержимое переменной var_4
```

```

    push    eax
    ; Передаем значение var_4 функции printf.
    ; Судя по строке спецификаторов, var_4 действительно имеет тип int

    lea     ecx, [ebp+var_10]
    ; Получаем указатель на переменную var_10 и...

    push    ecx
    ; передаем его функции printf.
    ; Судя по строке спецификаторов, тип ECX - char *, следовательно, var_10
    ; и есть искомая строка. Интуиция нам подсказывает, что var_C и var_8,
    ; расположенные ниже (т. е. в более старших адресах) ее, также содержат
    ; строку. Просто компилятор, вместо того чтобы вызывать strcpy, решил, что
    ; будет быстрее скопировать ее самостоятельно, чем и ввел нас в заблуждение.
    ; Поэтому никогда не следует торопиться с идентификацией типов элементов
    ; структур! Тщательно проверяйте каждый байт, как он инициализируется и как
    ; используется. Операции пересылки в локальные переменные еще ни о чем
    ; не говорят!

    push    offset aSX      ; "%s %x\n"
    call    _printf
    add     esp, 0Ch

    mov     esp, ebp
    pop     ebp
    ; Закрываем кадр стека

    retn

main      endp

MyFunc    proc near          ; CODE XREF: main+14p

var_10    = dword ptr -10h
var_C     = dword ptr -0Ch
var_8     = dword ptr -8
var_4     = dword ptr -4

arg_0     = dword ptr 8
arg_4     = dword ptr 0Ch
arg_8     = dword ptr 10h
; Обратите внимание: функции передаются три аргумента, а не два, как было
; объявлено в прототипе

    push    ebp
    mov     ebp, esp
    ; Открываем кадр стека

    sub     esp, 10h
    ; Резервируем память для локальных переменных

    mov     eax, [ebp+arg_4]
    ; Загружаем в EAX указатель на второй справа аргумент

    push    eax
    ; Передаем указатель на arg_4 функции strcpy

    lea     ecx, [ebp+var_10]
    ; Загружаем в ECX указатель на локальную переменную var_10

```

```
push    ecx
; Передаем функции strcpy указатель на локальную переменную var_10

call    strcpy
add     esp, 8
; Копируем строку, переданную функции MyFunc, через аргумент arg_4

mov     edx, [ebp+arg_8]
; Загружаем в EDX значение самого правого аргумента, переданного MyFunc

mov     [ebp+var_4], edx
; Копируем arg_8 в локальную переменную var_4

mov     eax, [ebp+arg_0]
; Загружаем в EAX значение аргумента arg_0. Как мы уже знаем, в этом аргументе
; компилятор скрыто от программиста передает указатель на локальную переменную.
; Именно в эту переменную функция помещает структуру, возвращаемую "по значению".

mov     ecx, [ebp+var_10]
; Загружаем в ECX содержимое локальной переменной var_10.
; Как мы помним, в локальную переменную var_10 ранее была скопирована строка,
; следовательно, сейчас мы вновь увидим ее "двухсловное" копирование!

mov     [eax], ecx
mov     edx, [ebp+var_C]
mov     [eax+4], edx
mov     ecx, [ebp+var_8]
mov     [eax+8], ecx
; И точно! Из локальной переменной var_10 в локальную переменную *arg_0
; копирование происходит "вручную", а не с помощью strcpy!
; В общей сложности сейчас было скопировано 12 байтов, значит, первый элемент
; структуры выглядит так: char s0[12].
; Да, конечно, в исходном тесте было char s0[10], но компилятор,
; выравнивая элементы структуры по адресам, кратным четырем, перенес второй
; элемент - int x - по адресу base+0x12, тем самым создав "дыру" между концом
; строки и началом второго элемента.
; Анализ дизассемблерного листинга не позволяет восстановить истинный вид
; структуры. Единственное, что можно сказать, - длина строки s0
; лежит в интервале [9 - 12]
;
mov     edx, [ebp+var_4]
mov     [eax+0Ch], edx
; Копируем переменную var_4 (содержащую аргумент arg_8) в [eax+0C]
; Действительно, второй элемент структуры - int x - расположен по смещению
; 12 байтов от ее начала.

mov     eax, [ebp+arg_0]
; Возвращаем в EAX указатель на аргумент arg_0, содержащий указатель на
; возвращенную структуру

mov     esp, ebp
pop     ebp
; Закрываем кадр стека

retn
; Итак, прототип функции выглядит так:
```

```

; struct X {char s0[12], int a} MyFunc(struct X *x, char *y, int z)
;

MyFunc      endp

```

Возникает вопрос: а как возвращаются структуры, состоящие из сотен и тысяч байтов? Ответ: они копируются в локальную переменную, неявно переданную компилятором по ссылке, инструкцией **MOVS**, в чем мы сейчас и убедимся, изменив в исходном тексте предыдущего примера `char s0[10]`, на `char s0[0x666]`.

Результат перекомпиляции должен выглядеть так:

Листинг 94

```

MyFunc      proc near          ; CODE XREF: main+10p
var_66C      = byte ptr -66Ch
var_4        = dword ptr -4
arg_0        = dword ptr 8
arg_4        = dword ptr 0Ch
arg_8        = dword ptr 10h

    push     ebp
    mov      ebp, esp
    ; Открываем кадр стека

    sub      esp, 66Ch
    ; Резервируем память для локальных переменных

    push     esi
    push     edi
    ; Сохраняем регистры в стеке

    mov      eax, [ebp+arg_4]
    push     eax
    lea      ecx, [ebp+var_66C]
    push     ecx
    call     strcpy
    add      esp, 8
    ; Копируем переданную функции строку в локальную переменную var_66C

    mov      edx, [ebp+arg_8]
    mov      [ebp+var_4], edx
    ; Копируем аргумент arg_8 в локальную переменную var_4

    mov      ecx, 19Bh
    ; Заносим в ECX значение 0x19B, пока еще не понимая, что оно выражает,

    lea      esi, [ebp+var_66C]
    ; Устанавливаем регистр ESI на локальную переменную var_66C

    mov      edi, [ebp+arg_0]
    ; Устанавливаем регистр EDI на переменную, на которую указывает
    ; указатель, переданный в аргументе arg_0

    repe     movsd
    ; Копируем ECX двойных слов с ESI в EDI.
    ; Переводя это в байты, получаем: 0x19B*4 = 0x66C.

```



```

; Таким образом, копируется и строка var_66C, и переменная var_4
mov     eax, [ebp+arg_0]
; Возвращаем в EAX указатель на возвращенную структуру

pop     edi
pop     esi

mov     esp, ebp
pop     ebp
; Закрываем кадр стека
retn

MyFunc      endp

```

Следует учитывать, что многие компиляторы (например, WATCOM) передают функции указатель на буфер для возвращаемого значения не через стек, а через регистр, причем регистр, по обыкновению, берется не из очереди кандидатов в порядке предпочтения (см. табл. 6), а используется особый регистр, специально предназначенный для этой цели. Например, у WATCOM'a это регистр ESI.

Возвращение вещественных значений. Соглашения *cdecl* и *stdcall* предписывают возвращать вещественные значения (float, double, long double) через стек сопроцессора, значение же регистров EAX и EDX на выходе из такой функции может быть любым (другими словами, функции, возвращающие вещественные значения, оставляют регистры EAX и EDX в неопределенном состоянии).

Теоретически fastcall-функции могут возвращать вещественные переменные и в регистрах, но на практике до этого дело обычно не доходит, поскольку сопроцессор не может напрямую читать регистры основного процессора и их приходится проталкивать через оперативную память, что сводит на нет всю выгоду быстрого вызова.

Для подтверждения сказанного исследуем следующий пример:

Листинг 95. Пример, демонстрирующий возвращение вещественных значений

```

#include <stdio.h>

float MyFunc(float a, float b)
{
    return a+b;
}

main()
{
    printf("%f\n", MyFunc(6.66, 7.77));
}

```

Результат его компиляции Microsoft Visual C++ должен выглядеть приблизительно так:

Листинг 96

```

main                proc near                ; CODE XREF: start+AFp
var_8                = qword ptr -8

```

```

    push    ebp
    mov     ebp, esp
    ; Открываем кадр стека

    push    40F8A3D7h
    push    40D51EB8h
    ; Передаем функции MyFunc аргументы. Пока мы не можем установить их тип

    call    MyFunc

    fstp     [esp+8+var_8]
    ; Стягиваем со стека сопроцессора вещественное значение, занесенное туда
    ; функцией MyFunc.
    ; Чтобы определить его тип, смотрим опкод инструкции DD 1C 24.
    ; По табл. 10 определяем - он принадлежит double.
    ; Постой, постой, как double, ведь функция должна возвращать float?!
    ; Так-то оно так, но здесь имеет место неявное преобразование типов
    ; при передаче аргумента функции printf, ожидающей double.
    ; Обратите внимание на то, куда стягивается возвращенное функцией значение:
    ; [esp+8-8] == [esp], т. е. оно помещается на вершину стека, что равносильно
    ; его заталкиванию командами PUSH.

    push     offset aF      ; "%f\n"
    ; Передаем функции printf указатель на строку спецификаторов "%f\n"

    call     _printf
    add      esp, 0Ch

    pop      ebp
    retn

main        endp

MyFunc      proc near      ; CODE XREF: main+Dp

arg_0       = dword ptr 8
arg_4       = dword ptr 0Ch

    push    ebp
    mov     ebp, esp
    ; Открываем кадр стека

    fld     [ebp+arg_0]
    ; Затягиваем на вершину стека сопроцессора аргумент arg_0
    ; Чтобы определить его тип, смотрим на опкод инструкции FLD - D9 45 08.
    ; Раз так, это float

    fadd     [ebp+arg_4]
    ; Складываем arg_0, только что зятанутый на вершину стека сопроцессора, с arg_4,
    ; помещая результат в тот же стек, и...

    pop     ebp
    retn
    ; возвращаемся из функции, оставляя результат сложения двух float'ов
    ; на вершине стека сопроцессора.
    ; Забавно, если объявить функцию как double, это даст идентичный код!

MyFunc      endp

```

Замечание о механизме возвращения значений в компиляторе WATCOM C. Компилятор WATCOM C предоставляет программисту возможность ручного выбора регистра (регистров), в котором функция будет возвращать результат своей работы. Это серьезно осложняет анализ, ведь, как уже было сказано выше, по общепринятым соглашениям функция не должна портить регистры EBX, ESI и EDI (BX, SI и DI — в 16-разрядном коде). Увидев операцию чтения регистра ESI, идущую после вызова функции, в первую очередь мы решим, что он был инициализирован еще до ее вызова, ведь так происходит в подавляющем большинстве случаев. Но только не с WATCOM! Этот товарищ может заставить функцию возвращать значение в любом регистре общего назначения, за исключением EBP (BP), заставляя тем самым, исследовать и вызывающую и вызываемую функцию.

Таблица 13. Допустимые регистры для возвращения значения функции в компиляторе WATCOM C

Тип	Допустимые регистры					
Однобайтовый	AL		BL	CL	DL	
	AH		BH	CH	DH	
Двухбайтный	AX	CX	BX	DX	SI	DI
Четырехбайтный	EAX	EBX	ECX	EDX	ESI	EDI
Восьмибайтовый	EDX:EAX	ECX:EBX	ECX:EAX	ECX:ESI	EDX:EBX	EBX:EAX
	EDI:EAX	ECX:EDI	EDX:ESI	EDI:EBX	ESI:EAX	ECX:EDX
	EDX:EDI	EDI:ESI	ESI:EBX			
Ближний указатель	EAX	EBX	ECX	EDX	ESI	EDI
Дальний указатель	DX:EAX	CX:EBX	CX:EAX	CX:ESI	DX:EBX	DI:EAX
	CX:EDI	DX:ESI	DI:EBX	SI:EAX	CX:EDX	DX:EDI
	DI:ESI	SI:EBX	BX:EAX	FS:ECX	FS:EDX	FS:EDI
	FS:ESI	FS:EBX	FS:EAX	GS:ECX	GS:EDX	GS:EDI
	GS:ESI	GS:EBX	GS:EAX	DS:ECX	DS:EDX	DS:EDI
	DS:ESI	DS:EBX	DS:EAX	ES:ECX	ES:EDX	ES:EDI
	ES:ESI	ES:EBX	ES:EAX			
float	8087	???	???	???	???	???
double	8087	EDX:EAX	ECX:EBX	ECX:EAX	ECX:ESI	EDX:EBX
	EDI:EAX	ECX:EDI	EDX:ESI	EDI:EBX	ESI:EAX	ECX:EDX
	EDX:EDI	EDI:ESI	ESI:EBX	EBX:EAX		

Жирным шрифтом выделен регистр (регистры), используемые по умолчанию. Обратите внимание, что по используемому регистру невозможно непосредственно узнать тип возвращаемого значения, а только его размер. В частности, через

регистр EAX может возвращаться и переменная типа int, и структура из четырех переменных типа char (или двух char или одного short int).

Покажем, как это выглядит на практике. Рассмотрим следующий пример:

Листинг 97. Пример, демонстрирующий возвращение значения в произвольном регистре

```
#include <stdio.h>

int MyFunc(int a, int b)
{
#pragma aux MyFunc value [ESI]
// Прагма AUX вкупе с ключевым словом value позволяет вручную задавать регистр,
// через который будет возвращен результат вычислений.
// В данном случае его предписывается возвращать через ESI

    return a+b;
}

main()
{
    printf("%x\n", MyFunc(0x666, 0x777));
}
```

Результат компиляции этого примера должен выглядеть приблизительно так:

Листинг 98

```
main_      proc near          ; CODE XREF: __CMain+40p
            push    14h
            call    __CHK
            ; Проверка стека на переполнение

            push    edx
            push    esi
            ; Сохраняем ESI и EDX.
            ; Это говорит о том, что данный компилятор придерживается соглашения
            ; о сохранении ESI. Команды сохранения EDI не видно, однако этот регистр
            ; не модифицируется данной функцией и, стало быть, сохранять его незачем

            mov     edx, 777h
            mov     eax, 666h
            ; Передаем функции MyFunc два аргумента типа int

            call    MyFunc
            ; Вызываем MyFunc. По общепринятым соглашениям EAX, EDX и подчас ECX
            ; на выходе из функции содержат либо неопределенное, либо возвращенное функцией
            ; значение. Остальные регистры в общем случае должны быть сохранены

            push    esi
            ; Передаем регистр ESI функции printf. Мы не можем с уверенностью сказать,
            ; содержит ли он значение, возвращенное функцией, или был инициализирован еще
            ; до ее вызова

            push    offset asc_420004    ; "%x\n"
            call    printf_
            add     esp, 8
```

```

        pop     esi
        pop     edx

    retn

main_    endp

MyFunc   proc near          ; CODE XREF: main_+16p
    push     4
    call     __CHK
    ; Проверка стека на переполнение

    lea     esi, [eax+edx]
    ; А вот уже знакомый нам хитрый трюк со сложением. На первый взгляд в ESI
    ; загружается указатель на EAX+EBX, фактически так оно и происходит, но ведь
    ; указатель на EAX+EBX в то же время является и их суммой, т. е. эта команда
    ; эквивалентна ADD EAX, EDX/MOV ESI, EAX.
    ; Это и есть возвращаемое функцией значение, ведь ESI был модифицирован и
    ; не сохранен!
    ; Таким образом, вызывающая функция командой PUSH ESI передает printf
    ; результат сложения 0x666 и 0x777, что и требовалось выяснить

    retn

MyFunc   endp

```

Возвращение значений *in-line assembler* функциями. Создатель ассемблерной функции волен возвращать значения в любых регистрах, каких ему будет угодно, однако, поскольку вызывающие функции языка высокого уровня ожидают увидеть результат вычислений в строго определенных регистрах, писанные соглашения приходится соблюдать. Другое дело, внутренние ассемблерные функции, они могут вообще не придерживаться никаких правил, что и демонстрирует следующий пример:

Листинг 99. Пример, демонстрирующий возвращение значения встроенными ассемблерными функциями

```

#include <stdio.h>

// naked-функция, не имеющая прототипа, обо всем должен заботиться сам программист!
__declspec( naked ) int MyFunc()
{
    __asm{
        lea ebp, [eax+ecx] ; Возвращаем в EBP сумму EAX и ECX.
                           ; Такой трюк допустим лишь при условии, что эта
                           ; функция будет вызываться из ассемблерной функции,
                           ; знающей, через какие регистры передаются аргументы
                           ; и через какие возвращается результат вычислений

        ret
    }
}

main()
{
    int a=0x666;
    int b=0x777;
    int c;

```

```

__asm{
    push ebp
    push edi

    mov eax,[a];
    mov ecx,[b];
    lea edi,c

    // Вызываем функцию MyFunc из ассемблерной функции, передавая ей аргументы
    // через те регистры, которые она "хочет"
    call MyFunc;

    // Принимаем возвращенное в EBP значение и сохраняем его в локальной переменной
    mov [edi],ebp

    pop edi
    pop ebp
}

printf("%x\n",c);
}

```

Результат компиляции Microsoft Visual C++ (а другими компиляторами этот пример откомпилировать и вовсе не удастся, ибо они не поддерживают ключевое слово `naked`) должен выглядеть так:

Листинг 100

```

MyFunc          proc near ; CODE XREF: main+25p

    lea     ebp, [eax+ecx]
    ; Принимаем аргументы через регистры EAX и ECX, возвращая через регистр EBP
    ; их сумму. Конечно, пример несколько надуман, зато нагляден!

    retn
MyFunc endp

main            proc near          ; CODE XREF: start+AFp

var_C           = dword ptr -0Ch
var_8           = dword ptr -8
var_4           = dword ptr -4

    push    ebp
    mov     ebp, esp
    ; Открываем кадр стека

    sub     esp, 0Ch
    ; Резервируем место для локальных переменных

    push    ebx
    push    esi
    push    edi
    ; Сохраняем изменяемые регистры

    mov     [ebp+var_4], 666h
    mov     [ebp+var_8], 777h
    ; Инициализируем переменные var_4 и var_8

```

```
push    ebp
push    edi
; Сохраняем регистры или передаем их функции?
; Пока нельзя ответить однозначно

mov     eax, [ebp+var_4]
mov     ecx, [ebp+var_8]
; Загружаем в EAX значение переменной var_4, а в ECX - var_8

lea     edi, [ebp+var_C]
; Загружаем в EDI указатель на переменную var_C

call    MyFunc
; Вызываем MyFunc. Из анализа вызывающей функции не очень понятно, как
; ей передаются аргументы. Может, через стек, а может, и через регистры.
; Только исследование кода MyFunc позволяет установить, что верным оказывается
; последнее предположение. Да, аргументы передаются через регистры!

mov     [edi], ebp
; Что бы это значило? Анализ одной лишь вызывающей функции не может дать
; исчерпывающего ответа, и только анализ вызываемой подсказывает, что
; через EBP она возвращает результат вычислений.

pop     edi
pop     ebp
; Восстанавливаем измененные регистры.
; Это говорит о том, что выше эти регистры действительно сохранялись в стеке,
; а не передавались функции в качестве аргументов

mov     eax, [ebp+var_C]
; Загружаем в EAX содержимое переменной var_C

push    eax
push    offset unk_406030
call    _printf
add     esp, 8
; Вызываем printf

pop     edi
pop     esi
pop     ebx
; Восстанавливаем регистры

mov     esp, ebp
pop     ebp
; Закрываем кадр стека

retn

main    endp
```

Возврат значений через аргументы, переданные по ссылке. Идентификация значений, возвращенных через аргументы, переданные по ссылке, тесно переплетается с идентификацией самих аргументов (см. главу «Идентификация аргументов функций»). Выделив среди аргументов, переданных функции, указатели, заносим их в список кандидатов на возвращаемые значения.

Теперь поищем, нет ли среди них указателей на неинициализированные переменные, очевидно, их инициализирует сама вызываемая функция. Однако не стоит вычеркивать указатели на инициализированные переменные (особенно равные нулю), они так же могут возвращать значения. Уточнить ситуацию позволит анализ вызываемой функции — нас будут интересовать все операции модификации переменных, переданных по ссылке. Только не путайте это с модификацией переменных, переданных по значению. Последние автоматически умирают в момент завершения функции (точнее, вычистки аргументов из стека). Фактически это локальные переменные функции, и она безболезненно может изменять их, как ей вздумается.

Листинг 101. Пример, демонстрирующий возврат значений через переменные, переданные по ссылке

```
#include <stdio.h>
#include <string.h>

// Функция инвертирования строки src с ее записью в строку dst
void Reverse(char *dst, const char *src)
{
    strcpy(dst, src);
    _strrev( dst);
}

// Функция инвертирования строки s (результат записывается в саму же строку s)
void Reverse(char *s)
{
    _strrev( s );
}

// Функция возвращает сумму двух аргументов
int sum(int a, int b)
{
    // Мы можем безболезненно модифицировать аргументы, переданные по значению,
    // обращаясь с ними как с обычными локальными переменными,
    a+=b;    return a;
}

main()
{
    char s0[]="Hello, Sailor!";
    char s1[100];

    // Инвертируем строку s0, записывая ее в s1
    Reverse(&s1[0], &s0[0]);
    printf("%s\n", &s1[0]);

    // Инвертируем строку s1, перезаписывая ее
    Reverse(&s1[0]);
    printf("%s\n", &s1[0]);

    // Выводим сумму двух чисел
    printf("%x\n", sum(0x666, 0x777));
}
```


Результат компиляции этого примера должен выглядеть приблизительно так:

Листинг 102

```
main                proc near                ; CODE XREF: start+AFp
var_74              = byte ptr -74h
var_10              = dword ptr -10h
var_C               = dword ptr -0Ch
var_8               = dword ptr -8
var_4               = word ptr -4

    push    ebp
    mov     ebp, esp
    ; Открываем кадр стека

    sub     esp, 74h
    ; Резервируем память для локальных переменных

    mov     ax, dword ptr aHelloSailor ; "Hello,Sailor!"
    ; Заносим в регистр EAX четыре первых байта строки "Hello, Sailor!"
    ; Вероятно, компилятор копирует строку в локальную переменную таким
    ; хитро-тигриным способом

    mov     [ebp+var_10], eax
    mov     ecx, dword ptr aHelloSailor+4
    mov     [ebp+var_C], ecx
    mov     edx, dword ptr aHelloSailor+8
    mov     [ebp+var_8], edx
    mov     ax, word ptr aHelloSailor+0Ch
    mov     [ebp+var_4], ax
    ; Точно, строка "Hello,Sailor!" копируется в локальную переменную var_10
    ; типа char s[0x10].
    ; Число 0x10 было получено подсчетом количества копируемых байтов -
    ; четыре итерации по четыре байта в каждой, итого шестнадцать!

    lea     ecx, [ebp+var_10]
    ; Загрузка в ECX указателя на локальную переменную var_10,
    ; содержащую строку "Hello, World!"

    push    ecx      ; int
    ; Передача функции Reverse_1 указателя на строку "Hello, World!".
    ; Смотрите, IDA неверно определила тип, ну какой же это int,
    ; когда это char *.
    ; Однако, вспомнив, как копировалась строка, мы поймем, почему ошиблась IDA.

    lea     edx, [ebp+var_74]
    ; Загрузка в ECX указателя на неинициализированную
    ; локальную переменную var_74

    push    edx      ; char *
    ; Передача функции Reverse_1 указателя на неинициализированную переменную
    ; типа char s1[100].
    ; Число 100 было получено вычитанием смещения переменной var_74 от смещения
    ; следующей за ней переменной, var_10, содержащей строку "Hello, World!"
    ; 0x74 - 0x10 = 0x64 или в десятичном представлении - 100.
    ; Факт передачи указателя на неинициализированную переменную говорит о том,
```

```

; что, скорее всего, функция возвратит через нее некоторое значение, -
; возьмите это себе на заметку.

call    Reverse_1
add     esp, 8
; Вызов функции Reverse_1

lea     eax, [ebp+var_74]
; Загрузка в EAX указателя на переменную var_74

push    eax
; Передача функции printf указателя на переменную var_74. Поскольку
; вызывающая функция не инициализировала эту переменную, можно предположить,
; что вызываемая возвратила через нее свое значение.
; Возможно, функция Reverse_1 модифицировала и переменную var_10, однако
; об этом нельзя сказать с определенностью до тех пор, пока не будет
; изучен ее код

push    offset unk_406040
call    _printf
add     esp, 8
; Вызов функции printf для вывода строки

lea     ecx, [ebp+var_74]
; Загрузка в ECX указателя на переменную var_74, по-видимому
; содержащую возвращенное функцией Reverse_1 значение

push    ecx    ; char *.
; Передача функции Reverse_2 указателя на переменную var_74.
; Функция Reverse_2 также может вернуть в переменной var_74
; свое значение или некоторым образом модифицировать ее.
; Однако может ведь и не вернуть!
; Уточнить ситуацию позволяет анализ кода вызываемой функции.

call    Reverse_2
add     esp, 4
; Вызов функции Reverse_2

lea     edx, [ebp+var_74]
; Загрузка в EDX указателя на переменную var_74

push    edx
; Передача функции printf указателя на переменную var_74.
; Поскольку значение, возвращенное функцией через регистры EDX:EAX,
; не используется, можно предположить, что она возвращает его не через
; регистры, а в переменной var_74. Но это не более чем предположение

push    offset unk_406044
call    _printf
add     esp, 8
; Вызов функции printf

push    777h
; Передача функции Sum значения 0x777 типа int

push    666h
; Передача функции Sum значения 0x666 типа int

```

```

    call    Sum
    add     esp, 8
    ; Вызов функции Sum

    push    eax
    ; В регистре EAX содержится возвращенное функцией Sum значение.
    ; Передаем его функции printf в качестве аргумента

    push    offset unk_406048
    call    _printf
    add     esp, 8
    ; Вызов функции printf

    mov     esp, ebp
    pop     ebp
    ; Закрытие кадра стека

    retn

main      endp

; int __cdecl Reverse_1(char *,int)
; Обратите внимание, что прототип функции определен неправильно!
; На самом деле, как мы уже установили из анализа вызывающей функции, он выглядит так:
; Reverse(char *dst, char *src)
; Название аргументов дано на основании того, что левый аргумент -
; указатель на неинициализированный буфер и, скорее всего,
; он выступает в роли приемника,
; соответственно правый аргумент в таком случае источник.

Reverse_1    proc near                ; CODE XREF: main+32p

arg_0        = dword ptr 8
arg_4        = dword ptr 0Ch

    push     ebp
    mov      ebp, esp
    ; Открываем кадр стека

    mov      eax, [ebp+arg_4]
    ; Загружаем в EAX значение аргумента arg_4

    push     eax
    ; Передаем arg_4 функции strcpy

    mov      ecx, [ebp+arg_0]
    ; Загружаем в ECX значение аргумента arg_0

    push     ecx
    ; Передаем arg_0 функции strcpy

    call     strcpy
    add      esp, 8
    ; Копируем содержимое строки, на которую указывает arg_4, в буфер,
    ; на который указывает arg_0

    mov      edx, [ebp+arg_0]
    ; Загружаем в EDX содержимое аргумента arg_0, указывающего на буфер,
    ; содержащий только что скопированную строку,

```

```

    push    edx    ; char *
    ; Передаем функции __strrev arg_0

    call    __strrev
    add     esp, 4
    ; Функция strrev инвертирует строку, на которую указывает arg_0,
    ; следовательно, функция Reverse_1 действительно возвращает свое значение
    ; через аргумент arg_0, переданный по ссылке.
    ; Напротив, строка, на которую указывает arg_4, остается неизменной, поэтому
    ; прототип функции Reverse_1 выглядит так:
    ; void Reverse_1(char *dst, const char *src);
    ; Никогда не пренебрегайте квалификатором const, так как он ясно указывает на
    ; то, что переменная, на которую указывает данный указатель, используется
    ; лишь на чтение. Эта информация значительно облегчит работу с
    ; дизассемблерным листингом, особенно когда вы вернетесь к нему спустя
    ; некоторое время, основательно подзабыв алгоритм исследуемой программы,

    pop     ebp
    ; Закрываем кадр стека

    retn
Reverse_1    endp

; int __cdecl Reverse_2(char *)
; А вот на этот раз прототип функции определен верно!
; (Ну, за исключением того, что возвращаемый тип void, а не int.)

Reverse_2    proc near                ; CODE XREF: main+4Fp
arg_0        = dword ptr 8

    push    ebp
    mov     ebp, esp
    ; Открываем кадр стека

    mov     eax, [ebp+arg_0]
    ; Загружаем в EAX содержимое аргумента arg_0

    push    eax ; char *
    ; Передаем arg_0 функции strrev

    call    __strrev
    add     esp, 4
    ; Инвертируем строку, записывая результат на то же самое место.
    ; Следовательно, функция Reverse_2 действительно возвращает значение
    ; через arg_0 и наше предварительное предположение оказалось правильным!

    pop     ebp    ; Закрываем кадр стека

    retn
; Прототип функции Reverse_2, по данным последних исследований, выглядит так:
; void Reverse_2(char *s)

Reverse_2    endp

Sum          proc near                ; CODE XREF: main+72p

arg_0        = dword ptr 8
arg_4        = dword ptr 0Ch

```

```

push    ebp
mov     ebp, esp
; Открываем кадр стека

mov     eax, [ebp+arg_0]
; Загружаем в EAX значение аргумента arg_0

add     eax, [ebp+arg_4]
; Складываем arg_0 с arg_4, записывая результат в EAX

mov     [ebp+arg_0], eax
; Копируем результат сложения arg_0 и arg_4 обратно в arg_0.
; Неопытные хакеры могут принять это за возвращение значения через аргумент,
; однако это предположение неверно.
; Дело в том, что аргументы, переданные функции, после ее завершения
; выталкиваются из стека и тут же "погибают". Не забывайте:
; аргументы, переданные по значению, ведут себя так же, как и локальные
; переменные.

mov     eax, [ebp+arg_0]
; А вот сейчас в регистр EAX действительно копируется возвращаемое значение.
; Следовательно, прототип функции выглядит так:
; int Sum(int a, int b);

pop     ebp
; Закрываем кадр стека

ret     0
Sum     endp

```

Возврат значений через динамическую память (кучу). Возвращение значения через аргумент, переданный по ссылке, не очень-то украшает прототип функции. Он вмиг перестает быть интуитивно понятным и требует развернутых пояснений, что с этим аргументом ничего передавать не надо, напротив, будьте готовы отсюда принять. Но хвост с ней, с наглядностью и эстетикой (кто говорил, что быть программистом легко?), существует и более серьезная проблема — далеко не во всех случаях размер возвращаемых данных известен наперед, частенько он выясняется лишь в процессе работы вызываемой функции. Выделить буфер с запасом? Некрасиво и неэкономично, даже в системах с виртуальной памятью ее объем не безграничен. Вот если бы вызываемая функция самостоятельно выделяла для себя память, как раз по потребности, а потом возвращала на нее указатель. Сказано — сделано! Ошибка многих начинающих программистов как раз и заключается в попытке вернуть указатель на локальные переменные, увы, они «умирают» вместе с завершением функции, и указатель указывает в «космос». Правильное решение заключается в выделении памяти из кучи (динамической памяти), скажем, вызовом *malloc* или *new*, эта память «живет» вплоть до ее принудительного освобождения функцией *free* или *delete* соответственно.

Для анализа программы механизм выделения памяти не существен, основную роль играет тип возвращаемого значения. Отличить указатель от остальных типов достаточно легко — только указатель может использоваться в качестве поадресного выражения.

Разберем следующий пример:

Листинг 103. Пример, демонстрирующий возвращения значения через кучу

```
#include <stdio.h>
#include <malloc.h>
#include <stdlib.h>

char* MyFunc(int a)
{
    char *x;
    x = (char *) malloc(100);

    _ltoa(a,x,16);
    return x;
}

main()
{
    char *x;
    x=MyFunc(0x666);
    printf("0x%s\n",x);
    free(x);
}
```

Листинг 104

```
main                proc near                ; CODE XREF: start+AFp
var_4                = dword ptr -4

    push    ebp
    mov     ebp, esp
    ; Открываем кадр стека

    push    ecx
    ; Выделяем память под локальную переменную размером 4 байта (см. var_4)

    push    666h
    ; Передаем функции MyFunc значение 666 типа int

    call    MyFunc
    add     esp, 4
    ; Вызываем MyFunc. Обратите внимание, что функции ни один аргумент
    ; не был передан по ссылке!

    mov     [ebp+var_4], eax
    ; Копирование содержимого возвращенного функцией значения в переменную var_4

    mov     eax, [ebp+var_4]
    ; Супер! Загружаем в EAX возвращенное функцией значение обратно!

    push    eax
    ; Передаем возвращенное функцией значение функции printf.
    ; Судя по спецификатору, тип возвращенного значения char *.
    ; Поскольку функции MyFunc ни один из аргументов не передавался по ссылке,
    ; она явно выделила память самостоятельно и записала туда полученную строку.
    ; А если бы функции MyFunc передавались один или более аргументов по ссылке?
```

```

; Тогда не было бы никакой уверенности, что она не возвратила один из таких
; аргументов обратно, предварительно его модифицировав.
; Впрочем, модификация необязательно, например, передаем функции указатели на
; две строки, и она возвращает указатель на ту из них, которая, скажем, короче
; или содержит больше гласных букв.
; Поэтому не всякое возвращение указателя свидетельствует о модификации

push    offset a0xS      ; "0x%s\n"
call    _printf
add     esp, 8
; Вызов printf - вывод на экран строки, возвращенной функцией MyFunc,

mov     ecx, [ebp+var_4]
; В ECX загружаем значение указателя, возвращенного функцией MyFunc,

push    ecx ; void *
; Передаем указатель, возвращенный функцией MyFunc, функции free.
; Значит, MyFunc действительно самостоятельно выделяла память вызовом malloc

call    _free
add     esp, 4
; Освобождаем память, выделенную MyFunc для возвращения значения

mov     esp, ebp
pop     ebp
; Закрываем кадр стека

retn
; Таким образом, протип MyFunc выглядит так:
; char* MyFunc(int a)

main          endp

MyFunc        proc near                ; CODE XREF: main+9p

var_4         = dword ptr -4
arg_0         = dword ptr 8

push    ebp
mov     ebp, esp
; Открываем кадр стека

push    ecx
; Резервируем память под локальные переменные

push    64h ; size_t
call    _malloc
add     esp, 4
; Выделяем 0x64 байта памяти из кучи либо для собственных нужд функции, либо
; для возвращения результата. Поскольку из анализа кода вызывающей функции нам
; уже известно, что MyFunc возвращает указатель, очень вероятно, что вызов
; malloc выделяет память как раз для этой цели.
; Впрочем, вызовов malloc может быть и несколько, а указатель возвращается
; только на один из них

mov     [ebp+var_4], eax
; Запоминаем указатель в локальной переменной var_4

```

```

push    10h ; int
; Передаем функции __ltoa аргумент 0x10 (крайний справа) - требуемая система
; исчисления для перевода числа

mov     eax, [ebp+var_4]
; Загружаем в EAX содержимое указателя на выделенную из кучи память

push    eax ; char *
; Передаем функции ltoa указатель на буфер для возвращения результата

mov     ecx, [ebp+arg_0]
; Загружаем в EAX значение аргумента arg_0

push    ecx ; __int32
; Передаем функции ltoa аргумент arg_0 - значение типа int

call    __ltoa
add     esp, 0Ch
; Функция ltoa переводит число в строку и записывает ее в буфер по переданному
; указателю

mov     eax, [ebp+var_4]
; Возвращаем указатель на регион памяти, выделенный самой MyFunc из кучи, и
; содержащий результат работы ltoa

mov     esp, ebp
pop     ebp
; Закрываем кадр стека

retn
MyFunc      endp

```

Возврат значений через глобальные переменные. «Мыльную оперу» перепевов с возвращением указателей продолжает серия *«Возвращение значений через глобальные переменные (и/или указателя на глобальные переменные)»*. Вообще-то глобальные переменные — дурной тон, и такой стиль программирования характерен в основном для программистов с мышлением, необратимо искаленным идеологией Бацика с его недоразвитым механизмом вызова подпрограмм.

Подробнее об идентификации глобальных переменных рассказывается в одноименном разделе данной главы, здесь же мы сосредоточим наши усилия именно на изучении механизмов возвращения значений через глобальные переменные.

Фактически все глобальные переменные можно рассматривать как неявные аргументы каждой вызываемой функции и в то же время как возвращаемые значения. Любая функция может произвольным образом читать и модифицировать их, причем ни «передача», ни «возвращение» глобальных переменных не заметны при анализе кода вызывающей функции, для этого необходимо тщательно исследовать вызываемую, манипулирует ли она с глобальными переменными, и если да, то с какими. Можно зайти и с обратной стороны, просмотром сегмента данных найти все глобальные переменные, определить их смещение и, пройдясь контекстным поиском по всему файлу, выявить функции, которые на них ссылаются (подробнее см. раздел *«Идентификация глобальных переменных»*).

Помимо глобальных, существуют еще и *статические* переменные. Они также располагаются в сегменте данных, но непосредственно доступны только объявившей их функции. Точнее, ограничение наложено не на сами переменные, а на их имена. Чтобы предоставить другим функциям доступ к собственным статическим переменным, достаточно передать указатель. К счастью, этот трюк не создает хакерам никаких проблем (хоть некоторые злопыхатели и объявляют его «про-рехой в защите»), отсутствие непосредственного доступа к «чужим» статическим переменным и необходимость взаимодействовать с функцией-владелицей через предсказуемый интерфейс (возвращенный указатель) позволяет разбить программу на отдельные независимые модули, каждый из которых может быть проанализирован отдельно. Чтобы не быть голословным, продемонстрируем это на следующем примере:

Листинг 105. Пример, демонстрирующий возврат значения через глобальные статические переменные

```
#include <stdio.h>

char* MyFunc(int a)
{
    static char x[7][16]={"Понедельник", "Вторник", "Среда",
                          "Четверг", "Пятница",
                          "Суббота", "Воскресенье"};

    return &x[a-1][0];
}

main()
{
    printf("%s\n", MyFunc(6));
}
```

Результат компиляции компилятором Microsoft Visual C++ 6.0 с настройками по умолчанию выглядит так:

Листинг 106

```
MyFunc      proc near          ; CODE XREF: main+5p
arg_0       = dword ptr 8

    push    ebp
    mov     ebp, esp
    ; Открываем кадр стека

    mov     eax, [ebp+arg_0]
    ; Загружаем в EAX значение аргумента arg_0

    sub     eax, 1
    ; Уменьшаем EAX на единицу. Это косвенно свидетельствует о том, что arg_0 -
    ; не указатель, хотя математические операции над указателями в Си разрешены
    ; и активно используются

    shl     eax, 4
    ; Умножаем (arg_0 -1) на 16. Битовый сдвиг вправо на четыре равносителен 24 == 16
```

```

    add     eax, offset aPonedelNik      ; "Понедельник"
    ; Складываем полученное значение с базовым указателем на таблицу строк,
    ; расположенных в сегменте данных. А в сегменте данных находятся либо
    ; статические, либо глобальные переменные.
    ; Поскольку значение аргумента arg_0 умножается на некоторую величину
    ; (в данном случае на 16), можно предположить, что мы имеем дело с
    ; двумерным массивом. В данном случае - массивом строк фиксированной длины.
    ; Таким образом, в EAX содержится указатель на строку с индексом arg_0 - 1.
    ; Или, другими словами, с индексом arg_0, считая с одного.

    pop     ebp
    ; Закрываем кадр стека, возвращая в регистре EAX указатель на соответствующий
    ; элемент массива. Как мы видим, нет никакой принципиальной разницы между
    ; возвращением указателя на регион памяти, выделенный из кучи, с возвращением
    ; указателя на статические переменные, расположенные в сегменте данных.

    retn

MyFunc     endp

main       proc near                    ; CODE XREF: start+AFp
    push    ebp
    mov     ebp, esp
    ; Открываем кадр стека

    push    6
    ; Передаем функции MyFunc значение типа int
    ; (шестой день - суббота)

    call    MyFunc
    add     esp, 4
    ; Вызываем MyFunc

    push    eax
    ; Передаем возвращенное MyFunc значение функции printf.
    ; Судя по строке спецификаторов, это указатель на строку

    push    offset aS                   ; "%s\n"
    call    _printf
    add     esp, 8

    pop     ebp
    ; Закрываем кадр стека

    retn

main       endp

aPonedelNik db "Понедельник",0,0,0,0,0 ; DATA XREF: MyFunc+Co
; Наличие перекрестной ссылки только на одну функцию подсказывает, что тип
; этой переменной - static

aVtornik   db 'Вторник',0,0,0,0,0,0,0,0
aSreda     db 'Среда',0,0,0,0,0,0,0,0,0,0
aCetverg   db 'Четверг',0,0,0,0,0,0,0,0,0
aPqtnica   db 'Пятница',0,0,0,0,0,0,0,0,0,0
aSubbota   db 'Суббота',0,0,0,0,0,0,0,0,0,0
aVoskresenE db 'Воскресенье',0,0,0,0,0,0,0,0,0,0
aS         db '%s',0Ah,0 ;          DATA XREF: main+Eo

```

А теперь сравним предыдущий пример с настоящими глобальными переменными:

Листинг 107. Пример, демонстрирующий возврат значения через глобальные переменные

```
#include <stdio.h>

int a;
int b;
int c;

MyFunc()
{
    c=a+b;
}

main()
{
    a=0x666;
    b=0x777;
    MyFunc();
    printf("%x\n",c);
}
```

Листинг 108

```
main                proc near                ; CODE XREF: start+AFp
    push    ebp
    mov     ebp, esp
    ; Открываем кадр стека

    call    MyFunc
    ; Вызываем MyFunc. Обратите внимание, функции явно ничего не передается
    ; и ничего не возвращается. Потому ее прототип выглядит
    ; (по предварительному заключению) так:
    ; void MyFunc()

    call    Sum
    ; Вызываем функцию Sum, явно не принимающую и не возвращающую никаких значений.
    ; Ее предварительный прототип выглядит так: void Sum()

    mov     eax, c
    ; Загружаем в EAX значение глобальной переменной c.
    ; Смотрим в сегмент данных, так-так, вот она, переменная c, равная нулю.
    ; Однако этому значению нельзя доверять, - быть может, ее уже успели изменить
    ; ранее вызванные функции.
    ; Предположение о модификации подкрепляется парой перекрестных ссылок,
    ; одна из которых указывает на функцию Sum. Суффикс w, завершающий
    ; перекрестную ссылку, говорит о том, что Sum записывает в переменную c
    ; какое-то значение. Какое? Это можно узнать из анализа кода самой Sum.

    push    eax
    ; Передаем значение, возвращенное функцией Sum, через глобальную переменную c
    ; функции printf.
    ; Судя по строке спецификаторов, аргумент имеет тип int
```

```

    push    offset asc_406030    ; "%x\n"
    call    _printf
    add     esp, 8
    ; Выводим возвращенный Sum результат на терминал

    pop     ebp
    ; Закрываем кадр стека

    retn

main      endp

Sum       proc near              ; CODE XREF: main+8p
; Функция Sum не принимает через стек никаких аргументов!

    push    ebp
    mov     ebp, esp
    ; Открываем кадр стека

    mov     eax, a
    ; Загружаем в EAX значение глобальной переменной a.
    ; Находим a в сегменте данных, ага, есть перекрестная ссылка на MyFunc,
    ; которая что-то записывает в переменную a.
    ; Поскольку вызов MyFunc предшествовал вызову Sum, можно сказать, что MyFunc
    ; возвратила в a некоторое значение

    add     eax, b
    ; Складываем EAX (хранящий значение глобальной переменной a) с содержимым
    ; глобальной переменной b
    ; (все сказанное выше относительно a справедливо и для b).

    mov     c, eax
    ; Помещаем результат сложения a+b в переменную c.
    ; Как мы уже знаем (из анализа функции main), функция Sum в переменной c
    ; возвращает результат своих вычислений. Теперь мы узнали, каких именно.

    pop     ebp
    ; Закрываем кадр стека

    retn

Sum       endp

MyFunc    proc near              ; CODE XREF: main+3p
    push    ebp
    mov     ebp, esp
    ; Открываем кадр стека

    mov     a, 666h
    ; Присваиваем глобальной переменной a значение 0x666

    mov     b, 777h
    ; Присваиваем глобальной переменной b значение 0x777.
    ; Как мы выяснили из анализа двух предыдущих функций, функция MyFunc
    ; возвращает в переменных a и b результат своих вычислений.
    ; Теперь мы определили, какой именно, а вместе с тем смогли разобраться,
    ; как три функции взаимодействуют друг с другом.
    ; main() вызывает MyFunc(), та инициализирует глобальные переменные a и b,
    ; затем main() вызывает Sum(), помещая сумму a и b в глобальную c,
    ; наконец, main() берет эту c и передает ее через стек printf

```

```

; для вывода на экран.
; Уф! Как все запутано, а ведь это простейший пример из трех функций!
; Что же говорить о реальной программе, в которой этих функций тысячи, причем
; порядок вызова и поведение каждой из них далеко не так очевидны!

pop     ebp

ret     4
MyFunc  endp

a        dd 0          ; DATA XREF: MyFunc+3w Sum+3r
b        dd 0          ; DATA XREF: MyFunc+Dw Sum+8r
c        dd 0          ; DATA XREF: Sum+Ew main+Dr
; Судя по перекрестным ссылкам, все три переменные - глобальные, так как к
; каждой из них имеет непосредственный доступ более одной функции.

```

Возврат значений через флаги процессора. Для большинства ассемблерных функций характерно использование регистра флагов процессора для возвращения результата успешности выполнения функции. По общепринятому соглашению установленный флаг переноса (CF) свидетельствует об ошибке, второе место по популярности занимает флаг нуля (ZF), а остальные флаги практически вообще не используются.

Установка флага переноса осуществляется командой STC или любой математической операцией, приводящей к образованию переноса (например, CMP a, b где a < b), а сброс — командой CLC или соответствующей математической операцией.

Проверка флага переноса обычно осуществляется условными переходами JC xxx и JNC xxx, соответственно исполняющихся при наличии и отсутствии переноса. Условные переходы JB xxx и JNB xxx — их синтаксические синонимы, дающие при ассемблировании идентичный код.

Листинг 109

```

#include <stdio.h>

// Функция сообщения об ошибке деления
Err(){ printf("--ERR: DIV by Zero\n");}

// Вывод результата деления на экран
Ok(int a){printf("%x\n",a);}

// Ассемблерная функция деления.
// Делит EAX на EBX, возвращая частное в EAX, а остаток - в EDX
// При попытке деления на ноль устанавливает флаг переноса
__declspec(naked) MyFunc()
{
__asm{
    xor edx,edx    ; Обнуляем EDX, т. е. команда div ожидает делимого в EDX:EAX.
    test ebx,ebx   ; Проверка делителя на равенство нулю.
    jz _err        ; Если делитель равен нулю, перейти к ветке _err.

    div ebx        ; Делим EDX:EAX на EBX (EBX заведомо не равен нулю).

    ret           ; Выход в c возвратом частного в EAX и остатка в EDX.
}
}

```

```

_err:                ; // Эта ветка получает управление при попытке деления на ноль.
        stc          ; устанавливаем флаг переноса, сигнализируя об ошибке и...
        ret          ; выходим.
    }
}

// Обертка для MyFunc.
// Принимаем два аргумента через стек - делимое и делитель -
// и выводим результат деления (или сообщение об ошибке) на экран
__declspec(naked) MyFunc_2(int a, int b)
{
    __asm{
        mov eax,[esp+4]        ; Загружаем в EAX содержимое аргумента a.
        mov ebx,[esp+8]        ; Загружаем в EDX содержимое аргумента b.

        call MyFunc           ; Пытаемся делить a/b.
        jnc _ok               ; Если флаг переноса сброшен, выводим результат, иначе...

        call Err              ; сообщение об ошибке.

        ret                   ; Возвращаемся.
    _ok:
        push eax              ; Передаем результат деления и...
        call 0k                ; выводим его на экран.
        add esp,4              ; Вычищаем за собой стек.

        ret                   ; Возвращаемся.
    }
}

main(){MyFunc_2(4,0);}

```

Идентификация локальных стековых переменных

...общая масса бактерий гораздо больше, чем наша с вами суммарная масса. Бактерии — основа жизни на земле...

А. П. Капица

Локальные переменные размещаются в *стеке* (также называемым *автоматической памятью*) и удаляются оттуда вызываемой функцией по ее завершении. Рассмотрим подробнее, как это происходит. Сначала в стек затягиваются аргументы, передаваемые функции (если они есть), а сверху на них кладется адрес возврата, помещаемый туда инструкцией CALL, вызывающей эту функцию. Получив управление, функция *открывает кадр стека* — сохраняет прежнее значение регистра EBP и устанавливает его равным регистру ESP (регистр — указатель вершины стека). «Выше» (т. е. в более младших адресах) EBP находится свободная область стека, ниже — служебные данные (сохраненный EBP, адрес возврата) и аргументы.

Сохранность области стека, расположенная выше указателя вершины стека (регистра ESP), не гарантирована от затирания и искажения. Ее беспрепятст-

венно могут использовать, например, обработчики аппаратных прерываний, вызываемые в непредсказуемом месте в непредсказуемое время. Да и использование стека самой функцией (для сохранения ли регистров, или передачи аргументов) приведет к его искажению. Какой из этой ситуации выход? Принудительно переместить указатель вершины стека вверх, тем самым занимая данную область стека. Сохранность памяти, находящейся «ниже» ESP, **гарантируется** (имеется в виду — гарантируется от непреднамеренных искажений), очередной вызов инструкции PUSH занесет данные на вершину стека, не затирая локальные переменные.

По окончании же своей работы функция обязана вернуть ESP на прежнее место, иначе функция RET снимет со стека отнюдь не адрес возврата, а вообще неведь что (значение самой «верхней» локальной переменной) и передаст управление «в космос»...

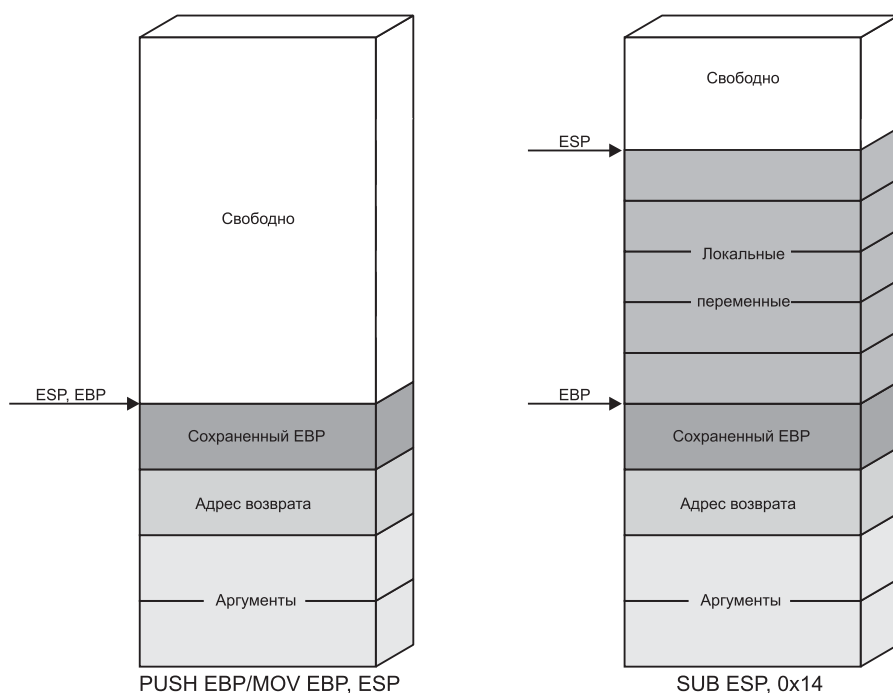


Рис. 15. Механизм размещения локальных переменных в стеке

На левой картинке показано состояние стека на момент вызова функции. Она открывает кадр стека, сохраняя прежнее значение регистра EBP и устанавливает его равным ESP. На правой картинке изображено резервирование 0x14 байтов стековой памяти под локальные переменные. Резервирование осуществляется перемещением регистра ESP «вверх» — в область младших адресов. Фактически локальные переменные размещаются в стеке так, как будто бы они были туда записаны командой PUSH. При завершении своей работы функция увеличивает значение регистра ESP, возвращая его на прежнюю позицию, освобождая тем са-

мым паять, занятую локальными переменными, стягивает со стека и восстанавливает значение EBP, закрывая тем самым кадр стека.

Адресация локальных переменных. Адресация локальных переменных очень похожа на адресацию стековых аргументов (см. раздел «Идентификация аргументов функций :: Адресация аргументов в стеке»), только аргументы располагаются «ниже» EBP, а локальные переменные — «выше». Другими словами, аргументы имеют положительные смещения относительно EBP, а локальные переменные — отрицательные, поэтому их очень легко отличить друг от друга. Так, например, [EBP+xxx] — аргумент, а [EBP-xxx] — локальная переменная.

Регистр-указатель кадра стека служит как бы барьером: по одну сторону от него аргументы функции, по другую — локальные переменные (рис. 16). Теперь понятно, почему при открытии кадра стека значение ESP копируется в EBP, иначе бы адресация локальных переменных и аргументов значительно усложнилась, а разработчики компиляторов (как это ни странно) тоже люди и не хотят без нужды усложнять себе жизнь. Впрочем, оптимизирующие компиляторы умеют адресовать локальные переменные и аргументы непосредственно через ESP, освобождая регистр EBP для более полезных целей. Подробнее об этом см. «FPO Frame Pointer Omission».

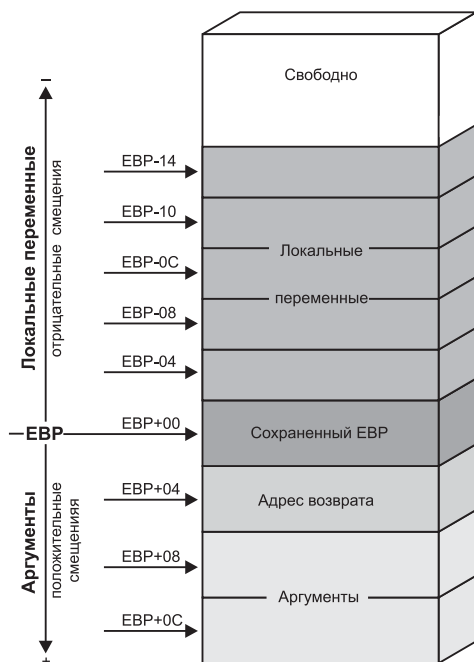


Рис. 16. Адресация локальных переменных

Механизм адресации локальных переменных очень похож на адресацию стековых аргументов, только аргументы расположены ниже указателя кадра стека — регистра EBP, а локальные переменные «проживают» выше его.

Детали технической реализации. Существует множество вариаций реализации выделения и освобождения памяти под локальные переменные. Казалось бы, чем плохо очевидное `SUB ESP, xxx` на входе и `ADD ESP, xxx` на выходе? А вот Borland C++ и некоторые другие компиляторы в стремлении отличаться от всех остальных резервируют память не уменьшением, а увеличением `ESP`... да, на отрицательное число, которое по умолчанию большинством дизасемблеров отображается как очень большое положительное. Оптимизирующие компиляторы при отводе небольшого количества памяти заменяют `SUB` на `PUSH reg`, что на несколько байтов короче. Последнее создает очевидные проблемы идентификации: попробуй разберись, то ли перед нами сохранение регистров в стеке, то ли передача аргументов, то ли резервирование памяти для локальных переменных (подробнее об этом см. раздел «Идентификация механизма выделения памяти»).

Алгоритм освобождения памяти также неоднозначен. Помимо увеличения регистра указателя вершины стека инструкцией `ADD ESP, xxx` (или в особо извращенных компиляторах увеличения его на отрицательное число), часто встречается конструкция `MOV ESP, EBP`. (Мы ведь помним, что при открытии кадра стека `ESP` копировался в `EBP`, а сам `EBP` в процессе исполнения функции не изменялся.) Наконец, память может быть освобождена инструкцией `POP`, выталкивающей локальные переменные одну за другой в какой-нибудь ненужный регистр (понятное дело, такой способ оправдывает себя лишь на небольшом количестве локальных переменных).

Таблица 14. Наиболее распространенные варианты реализации резервирования памяти под локальные переменные и ее освобождение

Действие	Варианты реализации		
Резервирование памяти	<code>SUB ESP, xxx</code>	<code>ADD ESP, -xxx</code>	<code>PUSH reg</code>
Освобождение памяти	<code>ADD ESP, xxx</code>	<code>SUB ESP, -xxx</code>	<code>POP reg</code>
	<code>MOV ESP, EBP</code>		

Идентификация механизма выделения памяти. Выделение памяти инструкциями `SUB` и `ADD` непротиворечиво и всегда интерпретируется однозначно. Если же выделение памяти осуществляется командой `PUSH`, а освобождение — `POP`, эта конструкция становится неотличима от простого освобождения/сохранения регистров в стеке. Ситуация серьезно усложняется тем, что в функции присутствуют и настоящие команды сохранения регистров, сливаясь с командами выделения памяти. Как узнать, сколько байтов резервируется для локальных переменных и резервируются ли они вообще (может, в функции локальных переменных и нет вовсе)?

Ответить на этот вопрос позволяет поиск обращений к ячейкам памяти, лежащих «выше» регистра `EBP`, т. е. с отрицательными относительными смещениями.

Рассмотрим два примера, приведенные в листинге 110.

Листинг 110

PUSH EBP	PUSH EBP
PUSH ECX	PUSH ECX
xxx	xxx
xxx	MOV [EBP-4], 0x666
xxx	xxx
POP ECX	POP ECX
POP EBP	POP EBP
RET RET	

В левом из них никакого обращения к локальным переменным не происходит вообще, а в правом наличествует конструкция `MOV [EBP-4], 0x666`, копирующая значение `0x666` в локальную переменную `var_4`. А раз есть локальная переменная, для нее кем-то должна быть выделена память. Поскольку инструкций `SUB ESP, xxx` и `ADD ESP, - xxx` в теле функций не наблюдается, подозрение падает на `PUSH ECX`, так как сохраненное содержимое регистра `ECX` располагается в стеке на четыре байта «выше» `EBP`. В данном случае подозревается лишь одна команда — `PUSH ECX`, поскольку `PUSH EBP` на роль «резерватора» не тянет, но как быть, если подозреваемых несколько?

Определить количество выделенной памяти можно по смещению самой «высокой» локальной переменной, которую удастся обнаружить в теле функции. То есть, отыскав все выражения типа `[EBP-xxx]`, выберем наибольшее смещение «xxx» — в общем случае оно равно количеству байтов выделенной под локальные переменные памяти. В частностях же встречаются объявленные, но не используемые локальные переменные. Им выделяется память (хотя оптимизирующие компиляторы просто выкидывают такие переменные за ненадобностью), но ни одного обращения к ним не происходит, и описанный выше алгоритм подсчета объема резервируемой памяти дает заниженный результат. Впрочем, эта ошибка никак не сказывается на результатах анализа программы.

Инициализация локальных переменных. Существует два способа инициализации локальных переменных: *присвоение необходимого значения инструкцией MOV* (например, `MOV [EBP-04], 0x666`) и *непосредственное заталкивание значения в стек инструкцией PUSH* (например, `PUSH 0x777`). Последнее позволяет выгодно комбинировать выделение памяти под локальные переменные с их инициализацией (разумеется, только в том случае, если этих переменных много).

Популярные компиляторы в подавляющем большинстве случаев выполняют операцию инициализации с помощью `MOV`, а `PUSH` более характерен для ассемблерных извращений, встречающихся, например, в защитах, имеющих задачу сбить хакеров с толку. Ну, если такой прием и совет хакера, то только начинающего.

Размещение массивов и структур. Массивы и структуры размещаются в стеке последовательно в смежных ячейках памяти, при этом *меньший индекс массива (элемент структуры) лежит по меньшему адресу*, но — **внимание** — адресуется большим модулем смещения относительно регистра указателя кадра стека. Это не покажется удивительным, если вспомнить, что локальные пе-

ременные адресуются отрицательными смещениями, следовательно, `[EBP-0x4] > [EBP-0x10]`.

Путаницу усиливает то обстоятельство, что, давая локальным переменным имена, IDA опускает знак минус. Поэтому из двух имен, скажем `var_4` и `var_10`, по меньшему адресу лежит то, чей индекс *больше*! Если `var_4` и `var_10` — это два конца массива, то с непривычки возникает произвольное желание поместить `var_4` в голову, а `var_10` в хвост массива, хотя на самом деле все наоборот!

Выравнивание в стеке. В некоторых случаях элементы структуры, массива и даже просто отдельные переменные требуется располагать по кратным адресам. Но ведь значение указателя вершины заранее не определено и неизвестно компилятору. Как же он, не зная фактического значения указателя, сможет выполнить это требование? Да очень просто — возьмет и откинет младшие биты ESP!

Легко доказать, что если младший бит равен нулю, число — четное. Чтобы быть уверенным, что значение указателя вершины стека делится на два без остатка, достаточно лишь сбросить его младший бит. Сбросив два бита, мы получим значение заведомо кратное четырем, три — восьми и т. д.

Сброс битов в подавляющем большинстве случаев осуществляется инструкцией AND. Например, `AND ESP, 0FFFFFF0` дает ESP кратным шестнадцати. Как было получено это значение? Переводим `0FFFFFF0` в двоичный вид, получаем — `11111111 11111111 11111111 11110000`. Видите четыре нуля на конце? Значит, четыре младших бита любого числа будут маскированы и оно разделится без остатка на $2^4 = 16$.

Хотя с локальными переменными мы уже неоднократно встречались при изучении прошлых примеров, не помешает это сделать еще один раз:

Листинг 111. Демонстрация идентификации локальных переменных

```
#include <stdio.h>
#include <stdlib.h>

int MyFunc(int a, int b)
{
    int c;           // Локальная переменная типа int.
    char x[50]       // Массив (демонстрирует схему размещения массивов в памяти_

    c=a+b;                               // Заносим в c сумму аргументов а и b

    ltoa(c,&x[0],0x10) ;                  // Переводим сумму а и b в строку

    printf("%x == %s == ",c,&x[0]);       // Выводим строку на экран

    return c;
}

main()
{
    int a=0x666; // Объявляем локальные переменные а и b для того, чтобы
    int b=0x777; // продемонстрировать механизм их инициализации компилятором.

    int c[1];    // Такие извращения понадобились для того, чтобы запретить
```

```

        // оптимизирующему компилятору помещать локальную переменную
        // в регистр (см. "Идентификация регистровых переменных").
        // Так как функции printf передается указатель на c, а
        // указатель на регистр быть передан не может, компилятор
        // вынужен оставить переменную в памяти

    c[0]=MyFunc(a,b);
    printf("%x\n",&c[0]);

    return 0;
}

```

Результат компиляции компилятора Microsoft Visual C++6.0 с настройками по умолчанию должен выглядеть так:

Листинг 112

```

MyFunc      proc near                ; CODE XREF: main+1Cp

var_38      = byte ptr -38h
var_4       = dword ptr -4
; Локальные переменные располагаются по отрицательному смещению относительно EBP,
; а аргументы функции – по положительному.
; Заметьте также, чем "выше" расположена переменная, тем больше модуль ее смещения

arg_0       = dword ptr 8
arg_4       = dword ptr 0Ch

    push     ebp
    mov      ebp, esp
    ; Открываем кадр стека

    sub      esp, 38h
    ; Уменьшаем значение ESP на 0x38, резервируя 0x38 байтов под локальные переменные

    mov      eax, [ebp+arg_0]
    ; загружаем в EAX значение аргумента arg_0.
    ; О том, что это аргумент, а не нечто иное, говорит его положительное
    ; смещение относительно регистра EBP

    add      eax, [ebp+arg_4]
    ; Складываем EAX со значением аргумента arg_0

    mov      [ebp+var_4], eax
    ; А вот и первая локальная переменная!
    ; На то, что это именно локальная переменная, указывает ее отрицательное
    ; смещение относительно регистра EBP. Почему отрицательное? А посмотрите,
    ; как IDA определила var_4.
    ; По моему личному мнению, было бы намного нагляднее, если бы отрицательные
    ; смещения локальных переменных подчеркивались более явно.

    push     10h ; int
    ; Передаем функции ltoa значение 0x10 (тип системы исчисления)

    lea      ecx, [ebp+var_38]
    ; Загружаем в ECX указатель на локальную переменную var_38.
    ; Что это за переменная? Прокрутим экран дизассемблера немного вверх,
    ; там, где содержится описание локальных переменных, распознанных IDA,

```

```

; var_38 = byte ptr -38h
; var_4 = dword ptr -4
;
; Ближайшая нижняя переменная имеет смещение -4, а var_38 соответственно -38.
; Вычитая из первого последнее, получаем размер var_38.
; Он, как нетрудно подсчитать, будет равен 0x34.
; С другой стороны, известно, что функция ltoa ожидает указатель на char*.
; Таким образом, в комментарии к var_38 можно записать char s[0x34].
; Это делается так: в меню Edit открываем подменю Functions, а в нем
; пункт Stack variables или нажимаем "горячую" комбинацию <Ctrl-K>.
; Открывается окно с перечнем всех распознанных локальных переменных.
; Подводим курсор к var_34 и нажимаем <;> для ввода повторяемого комментария
; и пишем нечто вроде char s[0x34]. Теперь нажимаем <Ctrl-Enter>
; для завершения ввода и <Esc> для закрытия окна локальных переменных.
; Все! Теперь возле всех обращений к var_34 появляется введенный нами
; комментарий
;
push    ecx    ; char *
; Передаем функции ltoa указатель на локальный буфер var_38

mov     edx, [ebp+var_4]
; Загружаем в EDX значение локальной переменной var_4

push    edx    ; __int32
; Передаем значение локальной переменной var_38 функции ltoa.
; На основании прототипа этой функции IDA уже определила тип переменной - int.
; Вновь нажмем <Ctrl-K> и прокомментируем var_4

call    __ltoa
add     esp, 0Ch
; Переводим содержимое var_4 в шестнадцатеричную систему исчисления,
; записанную в строковой форме, возвращая ответ в локальном буфере var_38,

lea     eax, [ebp+var_38]    ; char s[0x34]
; Загружаем в EAX указатель на локальный буфер var_34

push    eax
; Передаем указатель на var_34 функции printf для вывода содержимого на экран

mov     ecx, [ebp+var_4]
; Копируем в ECX значение локальной переменной var_4

push    ecx
; Передаем функции printf значение локальной переменной var_4

push    offset aXS    ; "%x == %s == "
call    _printf
add     esp, 0Ch

mov     eax, [ebp+var_4]
; Возвращаем в EAX значение локальной переменной var_4

mov     esp, ebp
; Освобождаем память, занятую локальными переменными

pop     ebp
; Восстанавливаем прежнее значение EBP

```

```

        retn
MyFunc      endp

main        proc near                ; CODE XREF: start+AFp
var_C       = dword ptr -0Ch
var_8       = dword ptr -8
var_4       = dword ptr -4

        push    ebp
        mov     ebp, esp
        ; Открываем кадр стека

        sub     esp, 0Ch
        ; Резервируем 0xС байт памяти для локальных переменных

        mov     [ebp+var_4], 666h
        ; Инициализируем локальную переменную var_4, присваивая ей значение 0x666,

        mov     [ebp+var_8], 777h
        ; Инициализируем локальную переменную var_8, присваивая ей значение 0x777.
        ; Смотрите, локальные переменные расположены в памяти в обратном порядке.
        ; Их обращения к ним! Не объявления, а именно обращения!
        ; Вообще-то порядок расположения не всегда бывает именно таким, это
        ; зависит от компилятора, поэтому полагаться на него никогда не стоит!

        mov     eax, [ebp+var_8]
        ; Копируем в регистр EAX значение локальной переменной var_8

        push    eax
        ; Передаем функции MyFunc значение локальной переменной var_8

        mov     ecx, [ebp+var_4]
        ; Копируем в ECX значение локальной переменной var_4

        push    ecx
        ; Передаем MyFunc значение локальной переменной var_4

        call    MyFunc
        add     esp, 8
        ; Вызываем MyFunc

        mov     [ebp+var_C], eax
        ; Копируем возвращенное функцией значение в локальную переменную var_C

        lea     edx, [ebp+var_C]
        ; Загружаем в EDX указатель на локальную переменную var_C

        push    edx
        ; Передаем функции printf указатель на локальную переменную var_C

        push    offset asc_406040     ; "%x\n"
        call    _printf
        add     esp, 8

        xor     eax, eax
        ; Возвращаем ноль

        mov     esp, ebp
        ; Освобождаем память, занятую локальными переменными

```

```

        pop     ebp
        ; Закрываем кадр стека

        retn

main    endp

```

Не очень сложно, правда? Что ж, тогда рассмотрим результат компиляции этого примера компилятором Borland C++ 5.0 — это будет немного труднее!

Листинг 113

```

MyFunc      proc near                ; CODE XREF: _main+14p

var_34      = byte ptr -34h
; Смотрите, только одна локальная переменная! А ведь мы объявляли целых три...
; Куда же они подевались?! Это хитрый компилятор поместил их в регистры, а не в стек
; для более быстрого к ним обращения
; (подробнее об этом см. "Идентификация регистровых и временных переменных").

        push    ebp
        mov     ebp, esp
        ; Открываем кадр стека

        add     esp, 0FFFFFFCh
        ; Резервируем... нажимаем <-> в IDA, превращая число в знаковое, получаем -34.
        ; Резервируем 0x34 байта под локальные переменные.
        ; Обратите внимание, на этот раз выделение памяти осуществляется не SUB, а ADD!

        push    ebx
        ; Сохраняем EBX в стеке или выделяем память локальным переменным?
        ; Поскольку память уже выделена инструкцией ADD, то в данном случае
        ; команда PUSH действительно сохраняет регистр в стеке

        lea     ebx, [edx+eax]
        ; А этим хитрым сложением мы получаем сумму EDX и EAX
        ; Поскольку EAX и EDX не инициализировались явно, очевидно, через них
        ; были переданы аргументы (см. раздел "Идентификация аргументов функций")

        push    10h
        ; Передаем функции ltoa выбранную систему исчисления

        lea     eax, [ebp+var_34]
        ; Загружаем в EAX указатель на локальный буфер var_34

        push    eax
        ; Передаем функции ltoa указатель на буфер для записи результата

        push    ebx
        ; Передаем сумму (не указатель!) двух аргументов функции MyFunc

        call    _ltoa
        add     esp, 0Ch

        lea     edx, [ebp+var_34]
        ; Загружаем в EDX указатель на локальный буфер var_34

        push    edx
        ; Передаем функции printf указатель на локальный буфер var_34, содержащий

```

```

; результат преобразования суммы аргументов MyFunc в строку
push    ebx
; Передаем сумму аргументов функции MyFunc

push    offset aXS    ; format
call    _printf
add     esp, 0Ch

mov     eax, ebx
; Возвращаем сумму аргументов в EAX

pop     ebx
; Вытаскиваем EBX из стека, восстанавливая его прежнее значение

mov     esp, ebp
; Освобождаем память, занятую локальными переменными

pop     ebp
; Закрываем кадр стека

ret     0
MyFunc   endp

; int __cdecl main(int argc,const char **argv,const char *envp)
_main    proc near          ; DATA XREF: DATA:00407044o

var_4    = dword ptr -4
; IDA распознала по крайней мере одну локальную переменную.
; Возьмем это себе на заметку.

argc     = dword ptr 8
argv     = dword ptr 0Ch
envp     = dword ptr 10h

push     ebp
mov      ebp, esp
; Открываем кадр стека

push     ecx
push     ebx
push     esi
; Сохраняем регистры в стеке

mov      esi, 777h
; Помещаем в регистр ESI значение 0x777

mov      ebx, 666h
; Помещаем в регистр EBX значение 0x666

mov      edx, esi
mov      eax, ebx
; Передаем функции MyFunc аргументы через регистры

call     MyFunc
; Вызываем MyFunc

mov      [ebp+var_4], eax
; Копируем результат, возвращенный функцией MyFunc в локальную переменную var_4.
; Стоп! Какую такую локальную переменную?! А кто под нее выделял память?!

```



```

; Не иначе как одна из команд PUSH. Только вот какая?
; Смотрим на смещение переменной - она лежит на четыре байта выше EBP, а эта
; область памяти занята содержимым регистра, сохраненного первым PUSH,
; следующим за открытием кадра стека.
; (Соответственно второй PUSH кладет значение регистра по смещению -8 и т. д.)
; А первой была команда PUSH ECX, следовательно, это никакое не сохранение
; регистра в стеке, а резервирование памяти под локальную переменную.
; Поскольку обращений к локальным переменным var_8 и var_C не наблюдается,
; команды PUSH EBX и PUSH ESI, по-видимому, действительно сохраняют регистры

```

```

lea    ecx, [ebp+var_4]
; Загружаем в ECX указатель на локальную переменную var_4

push    ecx
; Передаем указатель на var_4 функции printf

push    offset asc_407081    ; format
call    _printf
add     esp, 8

xor     eax, eax
; Возвращаем в EAX ноль

pop     esi
pop     ebx
; Восстанавливаем значения регистров ESI и EBX

pop     ecx
; Освобождаем память, выделенную локальной переменной var_4

pop     ebp
; Закрываем кадр стека

retn

_main    endp

```

FPO — Frame Pointer Omission. Традиционно для адресации локальных переменных используется регистр EBP. Учитывая, что регистров общего назначения всего *семь*, «насовсем» отдавать один из них локальным переменным уж очень не хочется. Нельзя найти какое-нибудь другое, более элегантное решение?

Хорошенько подумав, мы придем к выводу, что отдельный регистр для адресации локальных переменных вообще не нужен, — можно (не без ухищрений, правда) обойтись одним лишь ESP — регистром-указателем стека.

Единственная проблема — **плавающий кадр стека**. Пусть после выделения памяти под локальные переменные ESP указывает на вершину выделенного региона. Тогда переменная buff (рис. 17) окажется расположена по адресу ESP+0xC. Но стоит занести что-нибудь в стек (аргумент вызываемой функции или регистр на временное хранение), как кадр «уползет» и buff окажется расположен уже не по ESP+0xC, а **ESP+0x10!**

Современные компиляторы умеют адресовать локальные переменные через ESP, динамически отслеживая его значение (правда, при условии, что в теле функции нет хитрых ассемблерных вставок, изменяющих значение ESP непредсказуемым образом).

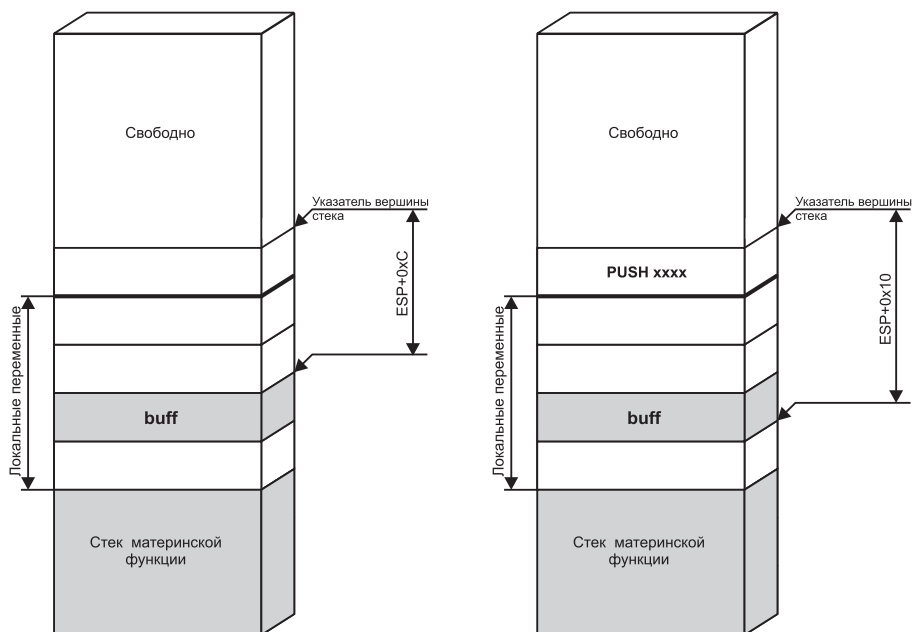


Рис. 17. Адресация локальных переменных через регистр ESP приводит к образованию плавающего кадра стека

Это чрезвычайно затрудняет изучение кода, поскольку теперь невозможно, ткнув пальцем в произвольное место кода, определить, к какой именно локальной переменной происходит обращение, приходится прочесывать всю функцию целиком, внимательно следя за значением ESP (и нередко впадая при этом в грубые ошибки, пускающие всю работу насмарку). К счастью, дизассемблер IDA умеет обращаться с такими переменными, но хакер тем и отличается от простого смертного, что никогда всецело не полагается на автоматику, а *сам* стремится понять, как это работает!

Рассмотрим наш старый добрый `simple.c`, откомпилировав его с ключом `/O2` — оптимизация по скорости. Тогда компилятор будет стремиться использовать все регистры и адресовать локальные переменные через ESP, что нам и надо.

```
>c1 sample.c /O2
```

```
00401000: 83 EC 64          sub     esp, 64h
```

Выделяем память для локальных переменных. Обратите внимание, теперь уже нет команд `PUSH EBP/MOV EBP,ESP`!

```
00401003: A0 00 69 40 00    mov     al,[00406900] ; mov al,0
```

```
00401008: 53                push    ebx
```

```
00401009: 55                push    ebp
```

```
0040100A: 56                push    esi
```

```
0040100B: 57                push    edi
```

Сохраняем регистры.

```
0040100C: 88 44 24 10       mov     byte ptr [esp+10h],al
```

Заносим в локальную переменную `[ESP+0x10]` (назовем ее `buff`) значение ноль.

```
00401010: B9 18 00 00 00    mov     ecx, 18h
00401015: 33 C0             xor     eax, eax
00401017: 8D 7C 24 11      lea     edi, [esp+11h]
```

Устанавливаем EDI на локальную переменную [ESP+0x11] (неинициализированный хвост buff).

```
0040101B: 68 60 60 40 00    push    406060h ; Enter password
```

Заносим в стек смещение строки Enter password. **Внимание!** Регистр ESP теперь уползает на 4 байта «вверх».

```
00401020: F3 AB             rep stos dword ptr [edi]
00401022: 66 AB             stos     word ptr [edi]
00401024: 33 ED             xor     ebp, ebp
00401026: AA               stos     byte ptr [edi]
```

Обнуляем буфер.

```
00401027: E8 F4 01 00 00    call    00401220
```

Вывод строки Enter password на экран. **Внимание!** Аргументы все еще не вытолкнуты из стека!

```
0040102C: 68 70 60 40 00    push    406070h
```

Заносим в стек смещение указателя на указатель stdin. **Внимание!** ESP еще уползает на четыре байта вверх.

```
00401031: 8D 4C 24 18      lea     ecx, [esp+18h]
```

Загружаем в ECX указатель на переменную [ESP+0x18]. Еще один буфер? Да как бы не так! Это уже знакомая нам переменная [ESP+0x10], но «сменившая облик» за счет изменения ESP. Если из 0x18 вычесть 8 байтов, на которые уполз ESP, получим 0x10, т. е. нашу старую знакомую — [ESP+0x10]!

Крохотную процедуру из десятка строк вручную проштудировать несложно, но вот на программе в миллион строк можно и лапти скинуть! Или... воспользоваться IDA. Посмотрите на результат ее работы:

```
.text:00401000 main          proc near          ; CODE XREF: start+AF↓p
.text:00401000
.text:00401000 var_64       = byte ptr -64h
.text:00401000 var_63       = byte ptr -63h
```

IDA обнаружила две локальные переменные, расположенные относительно кадра стека по смещениям 63 и 64, оттого и названных соответственно var_64 и var_63.

```
.text:00401000             sub     esp, 64h
.text:00401003             mov     al, byte_0_406900
.text:00401008             push    ebx
.text:00401009             push    ebp
.text:0040100A             push    esi
.text:0040100B             push    edi
.text:0040100C             mov     [esp+74h+var_64], al
```

IDA автоматически подставляет имя локальной переменной к ее смещению в кадре стека.

```
.text:00401010             mov     ecx, 18h
.text:00401015             xor     eax, eax
.text:00401017             lea     edi, [esp+74h+var_63]
```

Конечно, IDA не смогла распознать инициализацию первого байта буфера и ошибочно приняла его за отдельную переменную, но это не ее вина, а компилятора! Разобраться, сколько переменных тут, в действительности может только человек!

```
.text:0040101B      push      offset aEnterPassword ; "Enter password:"
.text:00401020      repe stosd
.text:00401022      stosw
.text:00401024      xor       ebp, ebp
.text:00401026      stosb
.text:00401027      call     sub_0_401220
.text:0040102C      push     offset off_0_406070
.text:00401031      lea      ecx, [esp+7Ch+var_64]
```

Обратите внимание, IDA правильно распознала обращение к нашей переменной, хотя ее смещение — 0x7C — отличается от 0x74!

Идентификация регистровых и временных переменных

Ничто не постоянно так, как временное.

Народная мудрость

Стремясь минимализировать количество обращений к памяти, оптимизирующие компиляторы размещают наиболее интенсивно используемые локальные переменные в регистрах общего назначения, только по необходимости сохраняя их в стеке (а в идеальном случае, не сохраняя их вовсе).

Какие трудности для анализа это создает? Во-первых, вводит *контекстную зависимость* в код. Так, увидев в любой точке функции команду типа MOV EAX,[EBP+var_10], мы с уверенностью можем утверждать, что здесь в регистр EAX копируется содержимое переменной var_10. А что это за переменная? Это можно легко узнать, пройдясь по телу функции на предмет поиска всех вхождений var_10, они-то и подскажут назначение переменной!

С регистровыми переменными этот номер не пройдет! Положим, нам встретилась инструкция MOV EAX,ESI и мы хотим отследить все обращения к регистровой переменной ESI. Как быть, ведь поиск подстроки ESI в теле функции ничего не даст, вернее, наоборот, выдаст множество ложных срабатываний. Ведь один и тот же регистр (в нашем случае ESI) может использоваться (и используется) для временного хранения множества различных переменных! Поскольку регистров общего назначения всего семь, да к тому же EBP закреплен за указателем кадра стека, а EAX и EDX — за возвращаемым значением функции, остается всего четыре регистра, пригодных для хранения локальных переменных. А в программах на языке C++ и того меньше — один из этих четырех регистров идет под указатель на виртуальную таблицу, а другой — под указатель на экземпляр this. Плохи дела! С двумя регистрами особо не разгонишься, в типичной функции локальных переменных — десятки! Вот компилятор и использует регистры как кэш, только в исключительных случаях каждая локальная переменная сидит в своем регистре, чаще всего переменные хаотично скачут по регистрам, временами сохраняются в стеке, зачастую выталкиваясь совсем в другой регистр (не в тот, чье содержимое сохранялась).

Практически все распространенные дизассемблеры (в том числе и IDA) не в состоянии отслеживать «миграции» регистровых переменных, и эту операцию

приходится выполнять «вручную». Определить содержимое интересующего регистра в произвольной точке программы достаточно просто, хотя и утомительно, достаточно прогнать программу с начала функции до этой точки на «эмуляторе Pentium'a», работающего в «голове», отслеживая все операции пересылки. Гораздо сложнее выяснить, **сколько** локальных переменных хранится в таком-то регистре. Когда большое количество переменных отображается на небольшое число регистров, однозначно восстановить отображение становится невозможно. Вот, например, программист объявляет переменную *a*, — компилятор помещает ее в регистр *X*. Затем некоторое время спустя программист объявляет переменную *b*, и, если переменная *a* более не используется (что бывает довольно часто), компилятор может поместить в тот же самый регистр *X* переменную *b*, не заботясь о сохранении значения *a* (*a* зачем его сохранять, если оно не нужно). В результате мы теряем одну переменную. На первый взгляд здесь нет никаких проблем. Теряем, ну и ладно! Теоретически это мог сделать и сам программист. Спрашивается: зачем он вводил *b*, когда для работы вполне достаточно одной *a*? Если переменные *a* и *b* имеют один тип, то никаких проблем, действительно, не возникает, но в противном случае анализ программы будет чрезвычайно затруднен.

Перейдем к технике идентификации регистровых переменных. Во многих хакерских руководствах утверждается, что регистровая переменная отличается от остальных тем, что никогда не обращается к памяти вообще. Это неверно. Регистровые переменные могут временно сохраняться в стеке командой *PUSH* и восстанавливаться обратно командой *POP*. Конечно, в некотором, «высшем смысле» такая переменная перестает быть регистровой, но и не становится стековой. Чтобы не дробить типы переменных на множество классов, условимся считать, что (как утверждают другие хакерские руководства) регистровая переменная — это переменная, содержащаяся в регистре общего назначения, возможно, сохраняемая в стеке, но всегда на *вершине*, а не в *кадре* стека. Другими словами, регистровые переменные никогда не адресуются через *EBP*. Если переменная адресуется через *EBP*, следовательно, она «прописана» в кадре стека и является стековой переменной. Правильно? Нет! Посмотрите, что произойдет, если регистровой переменной *a* присвоить значение стековой переменной *b*. Компилятор сгенерирует приблизительно следующий код *MOV REG, [EBP-xxx]*, соответственно присвоение стековой переменной значения регистровой будет выглядеть так: *MOV [EBP-xxx], REG*. Но, несмотря на явное обращение к кадру стека, переменная *REG* все же остается регистровой переменной. Рассмотрим следующий код:

Листинг 114

```
...  
MOV [EBP-0x4], 0x666  
MOV ESI, [EBP-0x4]  
MOV [EBP-0x8], ESI  
MOV ESI, 0x777  
SUB ESI, [EBP-0x8]  
MOV [EBP-0xC], ESI  
...
```

Его можно интерпретировать двояко — то ли действительно существует некая регистровая переменная ESI (тогда исходный тест примера должен выглядеть, как показано в листинге 115, а), то ли регистр ESI используется как временная переменная для пересылки данных (тогда исходный текст примера должен выглядеть, как показано в листинге 115, б):

Листинг 115

<pre>int var_4=0x666; int var_8=var_4; int var_C=0x777 - var_8</pre>	<pre>int var_4=0x666; register ⁴ int ESI = var_4; int var_8=ESI; ESI=0x777-var_8; int var_C = ESI</pre>
а)	б)

При том, что алгоритм обоих листингов абсолютно идентичен, левый из них заметно выигрывает в наглядности у правого. А главная цель дизассемблирования отнюдь не воспроизведение подлинного исходного текста программы, а реконструирование ее алгоритма. Совершенно безразлично, что представляет собой ESI — регистровую или временную переменную. Главное, чтобы костюмчик сидел. Грубо говоря, из нескольких вариантов интерпретации выбирайте самый наглядный!

Вот мы и подошли к понятию временных переменных, но, прежде чем заняться его изучением вплотную, завершим изучение регистровых переменных, исследованием следующего примера:

Листинг 116. Пример, демонстрирующий идентификацию регистровых переменных

```
main()
{
    int a=0x666;
    int b=0x777;
    int c;
    c=a+b;
    printf("%x + %x = %x\n", a, b, c);
    c=b-a;
    printf("%x - %x = %x\n", a, b, c);
}
```

⁴ В языках Си/Си++ существует ключевое слово `register` предназначенное для принудительного размещения переменных в регистрах. И все было бы хорошо, да подавляющее большинство компиляторов втихомолку игнорируют предписания программистов, размещая переменные там, где, по мнению компилятора, им будет «удобно». Разработчики компиляторов объясняют это тем, что компилятор лучше «знает», как построить наиболее эффективный код. Не надо, говорят они, пытаться помочь ему. Напрашивается следующая аналогия: пассажир говорит: мне надо в аэропорт, а таксист без возражений едет, «куда удобнее». Ну не должна работа на компиляторе превращаться в войну с ним, ну никак не должна! Отказ разместить переменную в регистре вполне законен, но в таком случае компиляция должна быть прекращена с выдачей сообщения об ошибке, типа «убери `register`, а то компилировать не буду!», или на худой конец — вывода предупреждения.

Результат компиляции Borland C++ 5.x должен выглядеть приблизительно так:

Листинг 117

```
; int __cdecl main(int argc,const char **argv,const char *envp)
_main      proc near          ; DATA XREF: DATA:00407044o

argc       = dword ptr  8
argv       = dword ptr  0Ch
envp       = dword ptr  10h
; Обратите внимание, IDA не распознала ни одной стековой переменной,
; хотя они объявлялись в программе.
; Выходит, компилятор разместил их в регистрах

    push    ebp
    mov     ebp, esp
    ; Открываем кадр стека

    push    ebx
    push    esi
    ; Сохраняем регистры в стеке или выделяем память для стековых переменных?
    ; Поскольку IDA не обнаружила ни одной стековой переменной, вероятнее всего,
    ; этот код сохраняет регистры

    mov     ebx, 666h
    ; Смотрите, инициализируем регистр! Сравните это с примером 112, приведенным в
    ; главе "Идентификация локальных стековых переменных". Помните, там было:
    ; mov [ebp+var_4], 666h
    ; Следовательно, можно заподозрить, что EBX – это регистровая переменная.
    ; Существование переменной доказывает тот факт, что, если бы значение 0x666
    ; непосредственно передавалось функции, т. е. так – printf(“%x %x %x\n”, 0x666),
    ; компилятор бы и поместил в код инструкцию PUSH 0x666.
    ; А раз не так, следовательно, значение 0x666 передавалось через переменную.
    ; Реконструируя исходный тест, пишем:
    ; 1. int a=0x666

    mov     esi, 777h
    ; Аналогично ESI, скорее всего, представляет собой регистровую переменную
    ; 2. int b=0x777

    lea     eax, [esi+ebx]
    ; Загружаем в EAX сумму ESI и EBX.
    ; Нет, EAX – не указатель, это просто сложение такое хитрое

    push    eax
    ; Передаем функции printf сумму регистровых переменных ESI и EBX.
    ; А вот что такое EAX – уже интересно. Ее можно представить и самостоятельной
    ; переменной, и непосредственной передачей суммы переменных a и b функции
    ; printf. Исходя из соображений удобочитаемости, выбираем последний вариант
    ; 3. printf (,,,a+b)

    push    esi
    ; Передаем функции printf регистровую переменную ESI, выше обозначенную нами
    ; как b
    ; 3. printf(,,,b,a+b)
```

```

push    ebx
; Передаем функции printf регистровую переменную EBX, выше обозначенную как a
; 3. printf(,,a,b,a+b)

push    offset aXXX    ; "%x + %x = %x"
; Передаем функции printf указатель на строку спецификаторов, судя по которой
; все три переменные имеют тип int
; 3. printf("%x + %x = %x", a, b, a + b)

call    _printf
add     esp, 10h

mov     eax, esi
; Копируем в EAX значение регистровой переменной ESI, обозначенную нами b,
; 4. int c=b

sub     eax, ebx
; Вычитаем от регистровой переменной EAX (c) значение переменной EBX (a)
; 5. c=c-a

push    eax
; Передаем функции printf разницу значений переменных EAX и EBX
; Ага! Мы видим, что от переменной c можно отказаться, непосредственно
; передав функции printf разницу значений b и a. Вычеркиваем строку 5
; (совершаем откат), а вместо 4. пишем следующее:
; 4. printf(,,,b-a)

push    esi
; Передаем функции printf значение регистровой переменной ESI (b)
; 4. printf(,,b, b-a)

push    ebx
; Передаем функции printf значение регистровой переменной EBX (a)
; 4. printf(,,a, b, b-a)

push    offset aXXX_0    ; "%x + %x = %x"
; Передаем функции printf указатель на строку спецификаторов, судя по которой
; все трое имеют тип int
; 4. printf("%x + %x = %x",a, b, b-a)

call    _printf
add     esp, 10h

xor     eax, eax
; Возвращаем в EAX нулевое значение
; return 0

pop     esi
pop     ebx
; Восстанавливаем регистры

pop     ebp
; Закрываем кадр стека

retn
; В итоге реконструированный текст выглядит так:
; 1. int a=0x666
; 2. int b=0x777

```



```

; 3. printf("%x + %x = %x", a, b, a + b)
; 4. printf("%x + %x = %x", a, b, b - a)
;
; Сравнивая свой результат с оригинальным исходным текстом, с некоторой досадой
; обнаруживаем, что все-таки слегка ошиблись, выкинув переменную c.
; Однако эта ошибка отнюдь не загубила нашу работу, напротив, придала
; листингу более "причесанный" вид, облегчая его восприятие.
; Впрочем, о вкусах не спорят, и если вы желаете точнее следовать ассемблерному
; коду, что ж, воля ваша, вводите еще и переменную s. Это решение, кстати,
; имеет тот плюс, что не придется делать "отката" - переписывать уже
; реконструированные строки для удаления из них лишней переменной

_main      endp

```

...когда же лебедь ушел от нас, мы его имя оставили себе, поскольку мы считали, что оно лебедю больше не понадобится.

А. А. Милн. Дом в медвежьем углу

Временные переменные. Временными переменными мы будем называть локальные переменные, внедряемые в код программы самим компилятором. Для чего они нужны? Рассмотрим следующий пример: `int b=a`. Если `a` и `b` — стековые переменные, то непосредственное присвоение невозможно, поскольку в микропроцессорах серии 80x86 отсутствует адресация «память — память». Вот и приходится выполнять эту операцию в два этапа: «память → регистр» + «регистр → память». Фактически компилятор генерирует следующий код:

```

register int tmp=a;          mov eax, [ebp+var_4]
int b=tmp;                  mov [ebp+var_8], eax

```

где `tmp` — и есть временная переменная, создаваемая лишь на время выполнения операции `b=a`, а затем уничтожаемая за ненадобностью.

Компиляторы (особенно оптимизирующие) всегда стремятся размещать временные переменные в регистрах и только в крайних случаях заталкивают их в стек. Механизмы выделения памяти и способы чтения/записи временных переменных довольно разнообразны.

Сохранение переменных в стеке — обычная реакция компилятора на острый недостаток регистров. Целочисленные переменные чаще всего закидываются на вершину стека командой `PUSH`, а стягиваются оттуда командой `POP`. Встретив в тексте программы «тянитолкая» (инструкцию `PUSH` в паре с соответствующей ей `POP`), сохраняющего содержимое инициализированного регистра, но не стековый аргумент функции (см. «Идентификация аргументов функции»), можно достаточно уверенно утверждать, что мы имеем дело с целочисленной временной переменной.

Выделение памяти под вещественные переменные и их инициализация в большинстве случаев происходят раздельно. Причина в том, что команды, позволяющей перебрасывать числа с вершины стека сопроцессора на вершину стека основного процессора, не существует и эту операцию приходится осуществлять

вручную. Первым делом «приподнимается» регистр — указатель вершины стека (обычно SUB ESP, xxx), затем в выделенные ячейки памяти записывается вещественное значение (обычно FSTP [ESP]), наконец, когда временная переменная становится не нужна, она удаляется из стека командой ADD ESP, xxx или подобной ей (SUB, ESP, - xxx).

Продвинутые компиляторы (например, Microsoft Visual C++) умеют располагать временные переменные в аргументах, оставшихся на вершине стека после завершения последней вызванной функции. Разумеется, этот трюк применим исключительно к cdecl-, но не stdcall-функциям, ибо последние самостоятельно вычищают свои аргументы из стека (подробнее об этом см. раздел «Идентификация аргументов функций»). Мы уже сталкивались с таким приемом при исследовании механизма возврата значений функцией в главе «Идентификация значения, возвращаемого функцией».

Временные переменные размером свыше восьми байтов (строки, массивы, структуры, объекты) практически всегда размещаются в стеке, заметно выделяясь среди прочих типов своим механизмом инициализации, — вместо традиционного MOV здесь используется одна из команд циклической пересылки MOVSw, при необходимости предваренная префиксом повторения REP (Microsoft Visual C++, Borland C++), или несколько команд MOVSw к ряду (WATCOM C).

Механизм выделения памяти под временные переменные практически идентичен механизму выделения памяти стековым локальным переменным, однако никаких проблем идентификации не возникает. Во-первых, выделение памяти стековым переменным происходит сразу же после открытия кадра стека, а временными переменными — в любой точке функции. Во-вторых, временные переменные адресуются не через регистр — указатель кадра стека, а через указатель вершины стека.

Таблица 15. Основные механизмы манипуляции с временными переменными

Действие	Методы		
	1-й	2-й	3-й
Резервирование памяти	PUSH	SUB ESP	Использовать стековые аргументы*
Освобождение памяти	POP	ADD ESP, xxx	
Запись переменной	PUSH	MOV [ESP+xxx]	MOVS
Чтение переменной	POP	MOV, [ESP+xxx]	Передача вызываемой функции
* Только в cdecl!			

В каких же случаях компилятором создаются временные переменные? Вообще-то это зависит от «нрава» самого компилятора (чужая душа всегда потемки, а уж тем более душа компилятора). Однако можно выделить по крайней мере два случая, когда без создания временных переменных ну никак не обойтись: 1) *при операциях присвоения, сложения, умножения*; 2) *в тех случаях, когда*

аргумент функции или член выражения — другая функция. Рассмотрим оба случая подробнее.

Создание временных переменных при пересылках данных и вычислении выражений. Как уже отмечалось выше, микропроцессоры серии 80x86 не поддерживают непосредственную пересылку данных из памяти в память, поэтому присвоение одной переменной значения другой требует ввода временной регистровой переменной (при условии, что остальные переменные не регистровые).

Вычисление выражений (особенно сложных) также требует временных переменных для хранения промежуточных результатов. Вот, например, сколько, по-вашему, требуется временных переменных для вычисления следующего выражения?

```
int a=0x1;int b=0x2;
int c= 1/((1-a) / (1-b));
```

Начнем со скобок, переписав их как: `int tmp_d = 1; tmp_d=tmp_d-a;` и `int tmp_e=1; tmp_e=tmp_e-b;` затем: `int tmp_f = tmp_d / tmp_e;` и наконец: `tmp_j=1; c=tmp_j / tmp_f.` Итого насчитываем... раз, два, три, четыре, ага, четыре временных переменных. Не слишком ли много? Давайте попробуем записать это короче:

```
int tmp_d = 1;tmp_d=tmp_d-a; // (1-a);
int tmp_e=1; tmp_e=tmp_e-b; // (1-b);
tmp_d=tmp_d/tmp_e; // (1-a) / (1-b);
tmp_e=1; tmp_e=tmp_e/tmp_d;
```

Как мы видим, вполне можно обойтись всего двумя временными переменными. Совсем другое дело! А что, если бы выражение было чуточку посложнее? Скажем, присутствовало бы десять пар скобок вместо трех. Сколько бы тогда потребовалось временных переменных? Нет, не соблазняйтесь искушением сразу же заглянуть в ответ, попробуйте сосчитать это сами! Уже сосчитали? Да что там считать, каким бы сложным выражение ни было, для его вычисления вполне достаточно всего двух временных переменных. А если раскрыть скобки, то можно ограничиться и одной, однако это потребует излишних вычислений. Этот вопрос во всех подробностях мы рассмотрим в главе «Идентификация выражений», а сейчас посмотрим, что за код сгенерировал компилятор:

Листинг 118

```
mov    [ebp+var_4], 1
mov    [ebp+var_8], 2
mov    [ebp+var_C], 3
; Инициализация локальных переменных

mov    eax, 1
; Вот вводится первая временная переменная.
; В нее записывается непосредственное значение, так как команда вычитания SUB,
; в силу архитектурных особенностей микропроцессоров серии 80x86, всегда
; записывает результат вычисления на место уменьшаемого, и потому
; уменьшаемое не может быть непосредственным значением, вот и приходится
; вводить временную переменную
```

```

sub    eax, [ebp+var_4]
; tEAX := 1 - var_4
; В регистре EAX теперь хранится вычисленное значение (1-a)

mov    ecx, 1
; Вводится еще одна временная переменная, поскольку EAX трогать нельзя -
; он занят

sub    ecx, [ebp+var_8]
; tECX := 1- var_8
; В регистре ECX теперь хранится вычисленное значение (1-b)

cdq
; Преобразуем двойное слово, лежащее в EAX, в четверное слово,
; помещаемое в EDX:EAX
; (машинная команда idiv всегда ожидает увидеть делимое именно в этих регистрах)

idiv   ecx
; Делим (1-a) на (1-b), помещая частное в tEAX.
; Прежнее значение временной переменной при этом неизбежно затирается, однако
; для дальнейших вычислений оно и не нужно.
; Вот и пускай себе затирается - не беда!

mov    ecx, eax
; Копируем значение (1-a) / (1-b) в регистр ECX.
; Фактически это новая временная переменная t2ECX, но в том же самом регистре
; (старое содержимое ECX нам также уже не нужно).
; Индекс 2 после префикса t дан для того, чтобы показать, что t2ECX
; вовсе не то же самое, что tECX, хотя обе эти временные переменные хранятся
; в одном регистре

mov    eax, 1
; Заносим в EAX непосредственное значение 1.
; Это еще одна временная переменная - t2EAX

cdq
; Обнуляем EDX

idiv   ecx
; Делим 1 на ((1-a) / (1-b))
; Частое помещается в EAX

mov    [ebp+var_10], eax
; c := 1 / ((1-a) / (1-b))
; Итак, для вычисления данного выражения потребовалось четыре временных
; переменных и всего два регистра общего назначения.

```

Создание временных переменных для сохранения значения, возвращенного функцией, и результатов вычисления выражений. Большинство языков высокого уровня (в том числе и Си/Си++) допускают подстановку функций и выражений в качестве непосредственных аргументов. Например, `myfunc(a+b, myfunc_2(c))` Прежде чем вызвать `myfunc`, компилятор должен вычислить значение выражения `a+b`. Это легко, но возникает вопрос: во что записать результат сложения? Посмотрим, как с этим справится компилятор:

Листинг 119

```
mov     eax, [ebp+var_C]
; Создается временная переменная tEAX, и в нее копируется значение
; локальной переменной var_C

push    eax
; Временная переменная tEAX сохраняется в стеке, передавая функции myfunc
; в качестве аргумента значение локальной переменной var_C.
; Хотя локальная переменная var_C в принципе могла бы быть непосредственно
; передана функции PUSH [ebp+var_4], и никаких временных переменных!

call    myfunc
add     esp, 4
; Функция myfunc возвращает свое значение в регистре EAX.
; Его можно рассматривать как своего рода еще одну временную переменную

push    eax
; Передаем функции myfunc_2 результат, возвращенный функцией myfunc

mov     ecx, [ebp+var_4]
; Копируем в ECX значение локальной переменной var_4.
; ECX еще одна временная переменная.
; Правда, не совсем понятно, почему компилятор не использовал регистр EAX,
; ведь предыдущая временная переменная ушла из области видимости и,
; стало быть, занимаемый ею регистр EAX освободился...

add     ecx, [ebp+var_8]
; ECX := var_4 + var_8

push    ecx
; Передаем функции myfunc_2 сумму двух локальных переменных

call    _myfunc_2
```

Область видимости временных переменных. Временные переменные — это в некотором роде *очень локальные переменные*. Область их видимости в большинстве случаев ограничена несколькими строками кода, вне контекста которых временная переменная не имеет никакого смысла. По большому счету временная переменная не имеет смысла вообще и только загромождает код. В самом деле, `myfunc(a+b)` намного короче и понятнее, чем `int tmp=a+b; myfunc(tmp)`. Поэтому, чтобы не засорять дизассемблерный листинг, стремитесь не употреблять в комментариях временные переменные, а подставляйте вместо них их фактические значения. Сами же временные переменные разумно предварять каким-нибудь характерным префиксом, например `tmp_` (или `t`, если вы патологический любитель краткости). Например:

Листинг 120

```
MOV EAX, [EBP+var_4] ; // var_8 := var_4
; ^ tEAX := var_4
ADD EAX, [EBP+var_8], ; ^ tEAX += var_8

PUSH EAX
; // MyFunc(var_4+var_8)
CALL MyFunc
```

Идентификация глобальных переменных

«Да, — подумала Алиса, — вот это дерябнулась так дерябнулась!»

Льюис Кэрролл. Алиса в стране чудес

Программа, наштапованная глобальными переменными, едва ли на самое страшное проклятие хакеров: вместо древа строгой иерархии компоненты программы тесно переплетаются друг с другом, и чтобы понять алгоритм одного из них, приходится прочесывать весь листинг в поисках перекрестных ссылок. А в совершенстве восстанавливать перекрестные ссылки не умеет ни один дизассемблер, даже IDA!

Идентифицировать глобальные переменные очень просто, гораздо проще, чем все остальные конструкции языков высокого уровня. Глобальные переменные сразу же выдают себя непосредственной адресацией памяти, т. е. обращение к ним выглядит приблизительно так: `MOV EAX, [401066]`, где `0x401066` и есть адрес глобальной переменной.

Сложнее понять, для чего эта переменная, собственно, нужна и каково ее содержимое на данный момент. В отличие от локальных переменных, глобальные переменные **контекстно-зависимы**. В самом деле, каждая локальная переменная инициализируется «своей» функцией и не зависит от того, какие функции были вызваны до нее. Напротив, глобальные переменные может модифицировать кто угодно и когда угодно, значение глобальной переменной в произвольной точке программы не определено. Чтобы его выяснить, необходимо проанализировать все манипулирующие с ней функции и — более того — восстановить порядок их вызова. Подробнее этот вопрос будет рассмотрен в главе «Построение дерева вызовов», пока же разберемся с техникой восстановления перекрестных ссылок.

Техника восстановления перекрестных ссылок. В большинстве случаев с восстановлением перекрестных ссылок сполна справляется автоматический анализатор IDA, и делать это вручную практически никогда не приходится. Однако бывает, что IDA ошибается, да и не всегда (и не у всех!) она бывает под рукой. Поэтому совсем нелишне уметь справляться с глобальными переменными самому.

Отслеживание обращений к глобальным переменным контекстным поиском их смещения в сегменте кода [данных]. Непосредственная адресация глобальных переменных чрезвычайно облегчает поиск манипулирующих с ними машинных команд. Рассмотрим, например, такую конструкцию: `MOV EA, [0x41B904]`. После ассемблирования она будет выглядеть так: `A1 04 B9 41 00`. Смещение глобальной переменной записывается, «как есть» (естественно, с соблюдением обратного порядка следования байтов — старшие располагаются по большему адресу, а младшие — по меньшему).

Тривиальный контекстный поиск позволит выявить все обращения к интересующей вас глобальной переменной, достаточно лишь узнать ее смещение, пере-

писать его справа налево и... вместе с полезной информацией получить какое-то количество «мусора». Ведь не каждое число, совпадающее по значению со смещением глобальной переменной, обязано быть указателем на эту переменную. К тому же 04 B9 41 00 удовлетворяет, например, следующий контекст:

83EC04	sub	esp, 004
B941000000	mov	ecx, 000000041

Ошибка очевидна — искомое значение не является операндом инструкции, более того, оно «захватило» сразу две инструкции! Отбрасыванием всех вхождений, пересекающих границы инструкции, мы сразу же избавляемся от значительной части «мусора». Единственная проблема, как определить границы инструкций, по части инструкции о самой инструкции сказать ничего нельзя.

Вот, например, встречается нам следующее: ...8D 81 04 B9 41 00 00... Эту последовательность, за вычетом последнего нуля, можно интерпретировать так: `lea eax,[ecx+0x41B904]`, но если предположить, что 0x8D принадлежит «хвосту» предыдущей команды, то получится следующее: `add d,[ecx][edi]*4,000000041`, а может быть, здесь и вовсе несколько команд...

Самый надежный способ определения границ машинных команд — трассированное дизассемблирование, но, к сожалению, это чрезвычайно ресурсоемкая операция и далеко не всякий дизассемблер умеет трассировать код. Поэтому приходится идти другим путем...

Образно машинный код можно изобразить в виде машинописного текста, напечатанного без пробелов. Если попробовать читать с произвольной позиции, мы, скорее всего, попадем на середину слова и ничего не поймем. Может быть, волей случая, первые несколько слогов и сложатся в осмысленное слово (а то и в два!), но дальше пойдет сплошная чепуха. Например: мамылараму. Ага, «мамы» — множественное число от «мама», подходит? Подходит. Дальше — лараму. «Лараму» — это что, народный индийский герой такой со множеством родительниц? Или «Мама ла Раму»? А как вам «Мама Ла Ра Му» — в смысле три мамы «Ла, Ра и Му»? Да, скажете тоже, вот ерунда какая!!!

Смещаемся на одну букву вперед, оставляя «м» предыдущему слову. «А», что ж, вполне возможно, это и есть союз «А», тем более что за ним идет осмысленное местоимение «мы», получается — «А мы Лараму» или «А мы Лара Му». Кто такой этот Лараму?!

Сдвигаемся еще на одну букву и читаем «мыла», а за ним «раму». Заработало! А «ма», стало быть, хвост от «мама».

Вот примерно так читается и машинный код, причем такая аналогия весьма адекватна. Слово (русское) не может начинаться с некоторых букв (например, с «Ы», мягкого и твердого знака), существуют характерные суффиксы и окончания, с сочетанием букв, практически не встречающихся в других частях предложения. Соответственно видя в конце несколько подряд идущих нулей, можно с высокой степенью уверенности утверждать, что это непосредственное значение, а непосредственные значения располагаются в конце команды (см. раздел «Тонкости дизассемблирования»).

Отличия констант от указателей, или Продолжаем разгребать мусор дальше. Вот наконец мы избавились от ложных срабатываний, бессмысленность которых очевидна с первого взгляда. Куча мусора заметно приуменьшилась, но... в ней все еще продолжают встречаться такие штучки, как `PUSH 0x401010`. Что такое `0x401010` — константа или смещение? С равным успехом может быть и то и другое. Пока не доберемся до манипулирующего с ней кода, мы вообще не сможем сказать ничего вразумительного. Если манипулирующий код обращается к `0x401010` по значению — это константа (выражающая, например, скорость улетывания Пятачка от Слонопотама), а если по ссылке — это указатель (в данном контексте смещение).

Подробнее эту проблему мы еще обсудим в главе «Идентификация констант и смещений», пока же заметим с большим облегчением, что минимальный адрес загрузки файла в Windows 9x равен `0x400000`, и немного существует констант, выражаемых таким большим числом.

Замечание: минимальный адрес загрузки Windows NT равен `0x10000`, однако, чтобы программа могла успешно работать и под NT, и под 9x, она должна грузиться не ниже `0x400000`.



Кошмары 16-разрядного режима. В 16-разрядном режиме отличить константу от указателя не так-то просто, как в 32-разрядном режиме! В 16-разрядном режиме под данные отводится один (или несколько) сегментов размером `0x10000` байтов, и допустимые значения смещений заключены в узком интервале [`0x0`, `0xFFFF`], причем у большинства переменных смещения очень невелики и визуальны неотличимы от констант.

Другая проблема — один сегмент чаще всего не вмещает в себя всех данных и приходится заводить еще один (а то и больше). Два сегмента — это еще ничего: один адресуется через регистр DS, другой — через ES, и никаких трудностей в определении «это указатель на переменную **какого** сегмента» не возникает. Например, если нас интересуют все обращения к глобальной переменной X, расположенной в основном сегменте по смещению `0x666`, то команду `MOV AX, ES:[0x666]` мы сразу же откинем в мусорную корзину, так как основной сегмент адресуется через DS (по умолчанию), а здесь — ES. Правда, обращение может происходить и в два этапа. Например: `MOV BX, 0x666 / xxx — xxx / MOV AX, ES:[BX]`, увидев `MOV BX, 0x666`, мы не только не можем определить сегмент, но и даже сказать, смещение ли это вообще? Впрочем, это не сильно затрудняет анализ...

Хуже, если сегментов данных в программе добрый десяток (а что, может же потребоваться порядка 640 килобайтов статической памяти?). Никаких сегментных регистров на это не хватит, и их переназначения будут происходить многократно. Тогда, чтобы узнать, к какому именно сегменту происходит обращение, потребуется определить значение сегментного регистра. А как его определить? Самое простое — прокрутить экран дизассемблера немного вверх, ища глазами инициализацию данного сегментного регистра, помня о том, что она может осуществляться не только командой `MOV segREG, REG`, но довольно частенько и `POP`! Например, `PUSH ES / POP DS` равносильно `MOV DS, ES`, правда, команды

MOV segREG, segREG в «языке» микропроцессоров 80x86, увы, нет. Как нет и команды MOV segREG, CONST, а потому ее приходится эмулировать «вручную» либо так: MOV AX, 0x666/MOV ES,AX, а можно и так: PUSH 0x666/POP ES.

Как хорошо, что 16-разрядный режим практически полностью ушел в прошлое, унося в песок истории все свои проблемы. Не только программисты, но и хакеры с переходом на 32-разрядный режим вздыхают с облегчением.

Косвенная адресация глобальных переменных. Довольно часто приходится слышать утверждение, что глобальные переменные **всегда** адресуются непосредственно (исключая, конечно, ассемблерные вставки, на ассемблере программист может обращаться к переменным, как захочет). На самом же деле все далеко не так... Если глобальная переменная передается функции по ссылке (а почему бы программисту не передать глобальную переменную по ссылке?), она будет адресоваться косвенно — через указатель.

Автору могут возразить: а зачем вообще явно передавать глобальную переменную функции? Любая функция и без этого может к ней обратиться. Автор не спорит. Да, **может**, но только, если знает об этом заранее. Вот, скажем, есть у нас функция `xchg`, обменивающая свои аргументы местами, и есть две глобальные переменные, которые позарез приспичило обменять. Функции `xchg` доступны все глобальные переменные, но она не знает, какие из них необходимо обменивать (и необходимо ли это вообще), вот и приходится ей явно передавать глобальные переменные как аргументы. А это значит, что всех обращений к глобальным переменным простым контекстным поиском мы не найдем. Самое печальное — не найдет их и IDA Pro (да и как бы она могла их найти? Для этого ей потребовался бы полноценный эмулятор процессора или хотя бы основных команд), на чем мы и убедимся в следующем примере:

Листинг 121. Явная передача глобальных переменных

```
#include <stdio.h>

int a; int b; // Глобальные переменные а и b.

// Функция, обменивающая значения аргументов
xchg(int *a, int *b)
{
    int c; c=*a; *b=*a; *a=c;
    //      ^^^^^^^^^^^^^^^^^^^^^^ косвенное обращение к аругментам по указателю.
    // если аргументы функции - глобальные переменные, то они будут адресоваться
    // не прямо, а косвенно
}

main()
{
    a=0x666; b=0x777; // Здесь непосредственное обращение к глобальным переменным
    xchg(&a, &b);      // Передача глобальной переменной по ссылке
}
```

Результат компиляции компилятором Microsoft Visual C++ должен выглядеть так:

Листинг 122

```
main          proc near          ; CODE XREF: start+AFp
    push      ebp
    mov       ebp, esp
    ; Открываем кадр стека

    mov       dword_405428, 666h
    ; Инициализируем глобальную переменную dword_405428.
    ; На то, что это действительно глобальная переменная, указывает непосредственная
    ; адресация

    mov       dword_40542C, 777h
    ; Инициализируем глобальную переменную dword_40542C

    push      offset dword_40542C
    ; Смотрите! Передаем функции смещение глобальной переменной dword_40542C как
    ; аргумент (т. е., другими словами, передаем ее по ссылке).
    ; Это значит, что вызываемая функция будет обращаться к переменной косвенно,
    ; через указатель, точно так, как она обращается с локальными переменными

    push      offset dword_405428
    ; Передаем функции смещение глобальной переменной dword_405428

    call      xchg
    add       esp, 8

    pop       ebp
    retn
main          endp

xchg          proc near          ; CODE XREF: main+21p

var_4         = dword ptr -4
arg_0         = dword ptr 8
arg_4         = dword ptr 0Ch

    push      ebp
    mov       ebp, esp
    ; Открываем кадр стека

    push      ecx
    ; Выделяем память для локальной переменной var_4

    mov       eax, [ebp+arg_0]
    ; Загружаем в EAX содержимое аргумента arg_0

    mov       ecx, [eax]
    ; Смотрите!
    ; Косвенное обращение к глобальной переменной!
    ; А еще говорят, будто бы таких не бывает!
    ; Разумеется, определить, что обращение происходит именно к глобальной
    ; переменной (и какой именно глобальной переменной), можно только анализом
    ; кода вызывающей функции
```

```

mov     [ebp+var_4], ecx
; Копируем значение *arg_0 в локальную переменную var_4

mov     edx, [ebp+arg_4]
; Загружаем в EDX содержимое аргумента arg_4

mov     eax, [ebp+arg_0]
; Загружаем в EAX содержимое аргумента arg_0

mov     ecx, [eax]
; Копируем в ECX значение аргумента *arg_0

mov     [edx], ecx
; Копируем в [arg_4] значение arg_0[0]

mov     edx, [ebp+arg_4]
; Загружаем в EDX значение arg_4

mov     eax, [ebp+var_4]
; Загружаем в EAX значение локальной переменной var_4 (хранит *arg_0)

mov     [edx], eax
; Загружаем в *arg_4 значение *arg_0

mov     esp, ebp
pop     ebp
retn

xchg    endp

dword_405428 dd 0 ; DATA XREF: main+3w main+1Co
; ~~~~~

dword_40542C dd 0 ; DATA XREF: main+Dw main+17o
; ~~~~~

; IDA нашла все ссылки на обе глобальные переменные.
; Первые две: main+3w и main+Dw на код инициализации
; (w - от write - т. е. в обращение на запись).
; Вторые две: main+1Co и main+17o
; (o - от offset - т. е. получение смещения глобальной переменной).
```

Если среди перекрестных ссылок на глобальную переменную присутствуют ссылки с суффиксом *o*, обозначающие взятие смещения (аналог ассемблерной директивы *offset*), то сразу же вскидывайте свои ушки на макушку — раз присутствует *offset*, значит, имеет место передача глобальной переменной по ссылке. А ссылка — это косвенная адресация. А косвенная адресация — это ласты: утомительный «ручной» анализ и никаких чудес прогресса.

Статические переменные. Статические переменные — это разновидность глобальных переменных, но с ограниченной областью видимости — они доступны только из той функции, в которой и были объявлены. Во всем остальном статические и глобальные переменные полностью совпадают — обе размещаются в сегменте данных, обе непосредственно адресуются (исключая случаи обращения через ссылку), обе...

...есть лишь одна существенная разница — к глобальной переменной могут обращаться любые функции, а к статической — только одна. А как насчет глобальных переменных, используемых лишь одной функцией? Да какие же это

глобальные переменные?! Это не глобальность, это кривость исходного кода программы. Если переменная используется лишь одной функцией, нет никакой необходимости объявлять ее глобальной!

Всякая непосредственно адресуемая ячейка памяти — глобальная (статическая) переменная (см. исключения ниже), но не всякая глобальная (глобальная) переменная всегда адресуется непосредственно.

Идентификация констант и смещений

То, что для одного человека константа, для другого — переменная.

А. Перлис. Афоризмы программирования

Микропроцессоры серии 80x86 поддерживают операнды трех типов: **регистр**, **непосредственное значение**, **непосредственный указатель**. Тип операнда явно задается в специальном поле машинной инструкции, именуемом **mod**, поэтому никаких проблем в идентификации типов операндов не возникает. Регистр — ну, все мы знаем, как выглядят регистры; указатель по общепринятому соглашению заключается в угловые скобки, а непосредственное значение записывается без них. Например:

```
MOV ECX, EAX;           ← регистровые операнды
MOV ECX, 0x666;         ← левый операнд регистровый, правый - непосредственный
MOV [0x401020], EAX      ← левый операнд - указатель, правый - регистр
```

Кроме этого микропроцессоры серии 80x86 поддерживают два вида адресации памяти: **непосредственную** и **косвенную**. Тип адресации определяется типом указателя. Если операнд непосредственный указатель, то и адресация непосредственна. Если же операнд-указатель — регистр, то такая адресация называется косвенной. Например:

```
MOV ECX, [0x401020]      ← непосредственная адресация
MOV ECX, [EAX]           ← косвенная адресация
```

Для инициализации регистрового указателя разработчики микропроцессора ввели специальную команду — **LEA REG, [addr]**, — вычисляющую значение адресного выражения **addr** и присваивающую его регистру **REG**. Например:

```
LEA EAX, [0x401020]      ; регистру EAX присваивается значение указателя 0x401020
MOV ECX, [EAX]           ; косвенная адресация - загрузка в ECX двойного слова,
                        ; расположенного по смещению 0x401020
```

Правый операнд команды **LEA** всегда представляет собой ближний (**near**) указатель. (Исключение составляют случаи использования **LEA** для сложения констант — подробнее об этом см. в одноименном пункте.) И все было бы хорошо.... да вот, оказывается, внутреннее представление ближнего указателя экви-

валентно константе того же значения. Отсюда LEA EAX, [0x401020] равносильно MOV EAX, 0x401020. В силу определенных причин MOV значительно обогнал в популярности LEA, практически вытеснив последнюю инструкцию из употребления.

Изгнание LEA породило фундаментальную проблему ассемблирования — **проблему OFFSET'a**. В общих чертах ее суть заключается в синтаксической неразличимости констант и смещений (ближних указателей). Конструкция MOV EAX, 0x401020 может грузить в EAX и константу, равную 0x401020 (пример соответствующего Си-кода: `a=0x401020`), и указатель на ячейку памяти, расположенную по смещению 0x401020 (пример соответствующего Си-кода: `a=&x`). Согласитесь, `a=0x401020` совсем не одно и то же, что `a=&x`! А теперь представьте, что произойдет, если в заново ассемблированной программе переменная `x`, в силу некоторых обстоятельств, окажется расположена по иному смещению, а не 0x401020? Правильно, — программа рухнет, ибо указатель `a` по-прежнему указывает на ячейку памяти 0x401020, но здесь теперь «проживает» совсем другая переменная!

Почему переменная может изменить свое смещение? Основных причин тому две. Во-первых, язык ассемблера неоднозначен и допускает двоякую интерпретацию. Например, конструкции ADD EAX, 0x66 соответствуют две машинные инструкции: 83 C0 66 и 05 66 00 00 00 длиной три и пять байтов соответственно. Транслятор может выбрать любую из них, и не факт, что ту же самую, которая была в исходной программе (до дизассемблирования). Неверно «угаданный» размер вызовет уплывание всех остальных инструкций, а вместе с ними и данных. Во-вторых, уплывание не замедлит вызвать модификацию программы (разумеется, речь идет не о замене JZ на JNZ, а настоящей адаптации или модернизации), и все указатели тут же «посыплются».

Вернуть работоспособность программы помогает директива `offset`. Если MOV EAX, 0x401020 **действительно** загружает в EAX указатель, а не константу, по смещению 0x401020 следует создать *метку*, именуемую, скажем, `loc_401020`, и MOV EAX, 0x401020 заменить на MOV EAX, **offset** loc_401020. Теперь указатель EAX связан не с фиксированным смещением, а с меткой!

А что произойдет, если предварить директивой `offset` константу, ошибочно приняв ее за указатель? Программа откажет в работе или станет работать некорректно. Допустим, число 0x401020 выражало собой объем бассейна через одну трубу, в который что-то втекает, а через другую — вытекает. Если заменить константу указателем, то объем бассейна станет равен... смещению метки в заново ассемблированной программе и все расчеты полетят к черту.

Таким образом, очень важно определить типы всех непосредственных операндов и еще важнее определить их **правильно**. Одна ошибка может стоить программе жизни (в смысле работоспособности), а в типичной программе тысячи и десятки тысяч операндов!

Отсюда возникает два вопроса:

- Как вообще определяют типы операндов?
- Можно ли их определять автоматически (или на худой конец хотя бы полуавтоматически)?

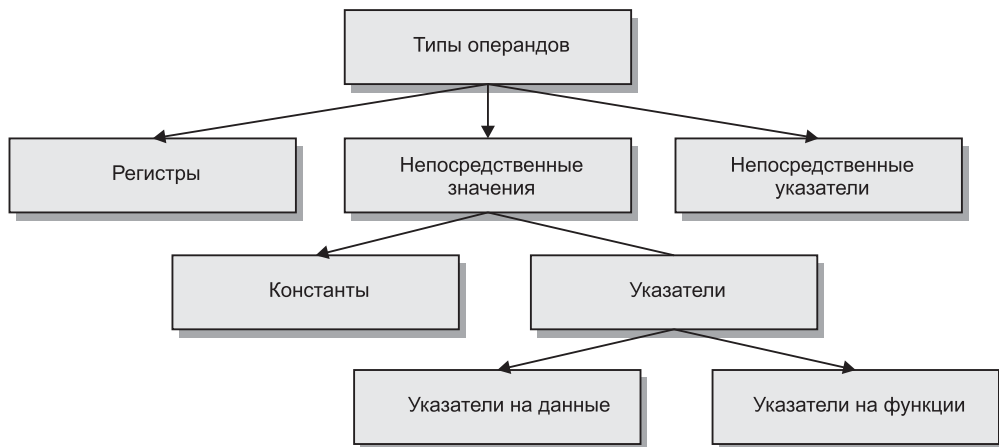


Рис. 18. Типы операндов

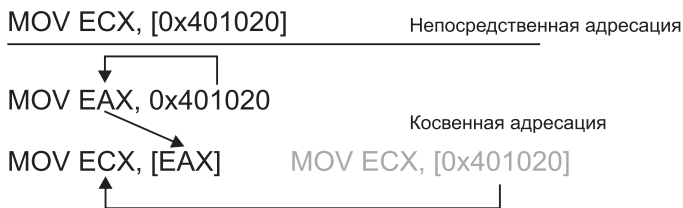


Рис. 19. Типы адресаций

Определение типа непосредственного операнда. Непосредственный операнд команды LEA — всегда указатель (исключение составляют ассемблерные «извращения»: чтобы сбить хакеров с толку, в некоторых защитах LEA используются для загрузки константы).

Непосредственные операнды команд MOV и PUSH могут быть как константами, так и указателями. Чтобы определить тип непосредственного операнда, необходимо проанализировать, *как* используется его значение в программе. Если для косвенной адресации памяти — это указатель, то в противном случае — константа.

Например, встретили мы в тексте программы команду MOV EAX, 0x401020 (рис 19). Что это такое: константа или указатель? Ответ на вопрос дает строка MOV ECX, [EAX], подсказывающая, что значение 0x401020 используется для косвенной адресации памяти, следовательно, непосредственный операнд не что иное, как указатель.

Существует два типа указателей — *указатели на данные* и *указатели на функцию*. Указатели на данные используются для извлечения значения ячейки памяти и встречаются в арифметических командах и командах пересылки (напри-

мер, MOV, ADD, SUB). Указатели на функцию используются в командах косвенного вызова и реже в командах косвенного перехода — CALL и JMP соответственно.

Рассмотрим следующий пример:

Листинг 123. Константы и указатели

```
main()
{
    static int a=0x777;
    int *b = &a;
    int c=b[0];
}
```

Результат его компиляции должен выглядеть приблизительно так:

Листинг 124

```
main                proc near
var_8               = dword ptr -8
var_4               = dword ptr -4

    push    ebp
    mov     ebp, esp
    sub     esp, 8
    ; Открываем кадр стека

    mov     [ebp+var_4], 410000h
    ; Загружаем в локальную переменную var_4 значение 0x410000.
    ; Пока мы не можем определить его тип - константа это или указатель.

    mov     eax, [ebp+var_4]
    ; Загружаем содержимое локальной переменной var_4 в регистр EAX

    mov     ecx, [eax]
    ; Загружаем в ECX содержимое ячейки памяти, на которую указывает указатель EAX.
    ; Ага! Значит, EAX все-таки указатель. Тогда локальная переменная var_4,
    ; откуда он был загружен, тоже указатель.
    ; И непосредственный операнд 0x410000 - указатель, а не константа!
    ; Следовательно, чтобы сохранить работоспособность программы, создадим по
    ; смещению 0x410000 метку loc_410000, ячейку памяти, расположенную по этому
    ; адресу, преобразует в двойное слово, и MOV [ebp+var_4], 410000h заменим на
    ; MOV [ebp+var_4], offset loc_410000

    mov     [ebp+var_8], ecx
    ; Присваиваем локальной переменной var_8 значение *var_4 ([offset loc_410000])

    mov     esp, ebp
    pop     ebp
    ; Закрываем кадр стека

    retn
main                endp
```

Рассмотрим теперь пример с косвенным вызовом процедуры:

Листинг 125. Пример, демонстрирующий косвенный вызов процедуры

```
func(int a, int b)
{
    return a+b;
};

main()
{
    int (*zzz) (int a, int b) = func;

    // Вызов функции происходит косвенно - по указателю zzz
    zzz(0x666, 0x777);
}
```

Результат компиляции должен выглядеть приблизительно так:

Листинг 126

```
.text:0040100B main    proc near                ; CODE XREF: start+AFp
.text:0040100B
.text:0040100B var_4  dword ptr -4
.text:0040100B
.text:0040100B         push     ebp
.text:0040100C         mov      ebp, esp
.text:0040100C         ; Открываем кадр стека
.text:0040100C
.text:0040100E         push     ecx
.text:0040100E         ; Выделяем память для локальной переменной var_4
.text:0040100E
.text:0040100F         mov      [ebp+var_4], 401000h
.text:0040100F         ; Присваиваем локальной переменной значение 0x401000
.text:0040100F         ; Пока еще мы не можем сказать, константа это или смещение
.text:0040100F
.text:00401016         push     777h
.text:00401016         ; Заносим значение 0x777 в стек. Константа это или указатель?
.text:00401016         ; Пока сказать невозможно - необходимо проанализировать
.text:00401016         ; вызываемую функцию
.text:00401016
.text:0040101B         push     666h
.text:0040101B         ; Заносим в стек непосредственное значение 0x666
.text:0040101B
.text:00401020         call    [ebp+var_4]
.text:00401020         ; Смотрите, косвенный вызов функции!
.text:00401020         ; Значит, переменная var_4 - указатель, раз так, то и
.text:00401020         ; присваиваемое ей непосредственное значение
.text:00401020         ; 0x401000 тоже указатель!
.text:00401020         ; А по адресу 0x401000 расположена вызываемая функция!
.text:00401020         ; Окрестим ее каким-нибудь именем, например MyFunc, и
.text:00401020         ; заменим mov [ebp+var_4], 401000h на
.text:00401020         ; mov [ebp+var_4], offset MyFunc
.text:00401020         ; после чего можно будет смело модифицировать программу
.text:00401020         ; Теперь-то она уже не "развалится"!
.text:00401020
```



```
.text:00401023      add     esp, 8
.text:00401023
.text:00401026      mov     esp, ebp
.text:00401028      pop     ebp
.text:00401028      ; Закрываем кадр стека
.text:00401028
.text:00401029      retn
.text:00401029 main      endp

.text:00401000 MyFunc      proc near
.text:00401000 ; А вот и косвенно вызываемая функция MyFunc
.text:00401000 ; Исследуем ее, чтобы определить тип передаваемых ей
.text:00401000 ; непосредственных значений
.text:00401000
.text:00401000 arg_0      = dword ptr 8
.text:00401000 arg_4      = dword ptr 0Ch
.text:00401000 ; Ага, вот они, наши аргументы!
.text:00401000
.text:00401000      push   ebp
.text:00401001      mov     ebp, esp
.text:00401001      ; Открываем кадр стека
.text:00401001
.text:00401003      mov     eax, [ebp+arg_0]
.text:00401003      ; Загружаем в EAX значение аргумента arg_0
.text:00401003
.text:00401006      add     eax, [ebp+arg_4]
.text:00401006      ; Складываем EAX (arg_0) со значением аргумента arg_0
.text:00401006      ; Операция сложения намекает, что по крайней мере один из
.text:00401006      ; двух аргументов не указатель, так как сложение двух указателей
.text:00401006      ; бессмысленно (см. "Сложные случаи адресации")
.text:00401006
.text:00401009      pop     ebp
.text:00401009      ; Закрываем кадр стека
.text:00401009
.text:0040100A      retn
.text:0040100A      ; Выходим, возвращая в EAX сумму двух аргументов
.text:0040100A      ; Как мы видим, ни здесь, ни в вызывающей функции
.text:0040100A      ; непосредственные значения 0x666 и 0x777 не использовались
.text:0040100A      ; для адресации памяти, - значит, это константы
.text:0040100A
.text:0040100A MyFunc      endp
.text:0040100A
```

Сложные случаи адресации или математические операции с указателями. С/С++ и некоторые другие языки программирования допускают выполнение над указателями различных арифметических операций, чем серьезно затрудняют идентификацию типов непосредственных операндов.

В самом деле, если бы такие операции с указателями были запрещены, то любая математическая инструкция, манипулирующая с непосредственным операндом, однозначно указывала бы на его константный тип.

К счастью, даже в тех языках, где это разрешено, над указателями выполняется ограниченное число математических операций. Так, совершенно бессмыс-

ленно сложение двух указателей, а уж тем более умножение или деление их друг на друга. Вычитание — дело другое. Используя тот факт, что компилятор располагает функции в памяти согласно порядку их объявления в программе, можно вычислить размер функции, отнимая ее указатель от указателя на следующую функцию (рис. 20). Такой трюк встречается в упаковщиках (распаковщиках) исполняемых файлов, защитах с самомодифицирующимся кодом, но в прикладных программах используется редко.

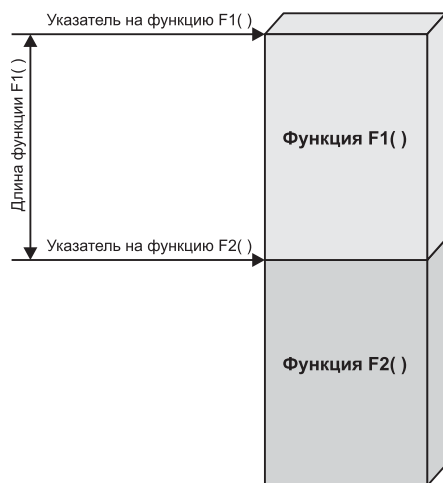


Рис. 20. Использование вычитания указателей для вычисления размера функции [структуры данных]

Сказанное выше относилось к случаям «указатель» + «указатель», между тем указатель может сочетаться и с константой. Причем такое сочетание настолько популярно, что микропроцессоры серии 80x86 даже поддерживают для этого специальную адресацию — *базовую*. Пусть, к примеру, имеется указатель на массив и индекс некоторого элемента массива. Очевидно, чтобы получить значение этого элемента, необходимо сложить указатель с индексом, умноженным на размер элемента.

Вычитание константы из указателя встречается гораздо реже, этому не только соответствует меньший круг задач, но и сами программисты избегают вычитания, поскольку оно нередко приводит к серьезным проблемам. Среди начинающих популярен следующий прием: если им требуется массив, начинающийся с единицы, они, объявив обычный массив, получают на него указатель и... уменьшают его на единицу! Элегантно, не правда ли? Нет, не правда, подумайте, что произойдет, если указатель на массив будет равен нулю. Правильно, «змея укусит» свой хвост, и указатель станет очень большим положительным числом. Вообще-то под Windows 9x/NT массив гарантированно не может быть размещен по нулевому смещению, но не стоит привыкать к трюкам, привязанным к одной платформе и не работающим на других.

«Нормальные» языки программирования запрещают смешение типов, и правильно! Иначе такая чехарда получается, не чехарда даже, а еще одна фундамен-

тельная проблема дизассемблирования — *определение типов в комбинированных выражениях*. Рассмотрим следующий пример:

```
MOV EAX, 0x...
MOV EBX, 0x...
ADD EAX, EBX
MOV ECX, [EAX]
```

Летающий Слонопотам! Сумма двух непосредственных значений используется для косвенной адресации. Ну, положим, оба они указателями быть не могут, исходя из самых общих соображений — никак не должны. Наверняка одно из непосредственных значений — указатель на массив (структуру данных, объект), а другое — индекс в этом массиве. Для сохранения работоспособности программы указатель необходимо заменить смещением метки, а вот индекс оставить без изменений (ведь индекс — это константа).

Как же различить, что есть что. Увы, нет универсального ответа, а в контексте приведенного выше примера это и вовсе *невозможно!*

Рассмотрим следующий пример:

Листинг 127. Пример, демонстрирующий определение типов в комбинированных выражениях

```
MyFunc(char *a, int i)
{
    a[i]='\n';
    a[i+1]=0;
}

main()
{
    static char buff[]="Hello, Sailor!";
    MyFunc(&buff[0], 5);
}
```

Результат компиляции Microsoft Visual C++ должен выглядеть так:

Листинг 128

```
main          proc near          ; CODE XREF: start+AFp
    push      ebp
    mov       ebp, esp
    ; Открываем кадр стека

    push      5
    ; Передаем функции MyFunc непосредственное значение 0x5

    push      405030h
    ; Передаем функции MyFunc непосредственное значение 0x405030

    call      MyFunc
    add       esp, 8
    ; Вызываем MyFunc(0x405030, 0x5)

    pop       ebp
    ; Закрываем кадр стека
```

```

    retn
main      endp

MyFunc    proc near                ; CODE XREF: main+Ap
arg_0     = dword ptr 8
arg_4     = dword ptr 0Ch

    push   ebp
    mov    ebp, esp
    ; Открываем кадр стека

    mov     eax, [ebp+arg_0]
    ; Загружаем в EAX значение аргумента arg_0
    ; (arg_0 содержит непосредственное значение 0x405030)

    add     eax, [ebp+arg_4]
    ; Складываем EAX со значением аргумента arg_4 (он содержит значение 0x5).
    ; Операция сложения указывает на то, что по крайней мере один из них
    ; константа, а другой - либо константа, либо указатель

    mov     byte ptr [eax], 0Ah
    ; Ага! Сумма непосредственных значений используется для косвенной адресации
    ; памяти, значит, это константа и указатель. Но кто есть кто?
    ; Для ответа на этот вопрос нам необходимо понять смысл кода программы -
    ; чего же добивался программист сложением указателей?
    ; Предположим, что значение 0x5 - указатель. Логично?
    ; Да, вот не очень-то логично, если это указатель, то указатель на что?
    ; Первые 64 килобайта адресного пространства Windows NT заблокированы для
    ; "отлавливания" нулевых и неинициализированных указателей.
    ; Ясно, что равным пяти указатель быть никак не может, разве что программки
    ; использовал какой-нибудь очень извращенный трюк.
    ; А если указатель 0x401000? Выглядит правдоподобным легальным смещением...
    ; Кстати, что там у нас расположено? Секундочку...
    ; 00401000 db 'Hello,Sailor!',0
    ;
    ; Теперь все сходится - функции передан указатель на строку "Hello, Sailor!"
    ; (значение 0x401000) и индекс символа этой строки (значение 0x5),
    ; функция сложила указатель со строкой и записала в полученную ячейку символ \n

    mov     ecx, [ebp+arg_0]
    ; В ECX заносится значение аргумента arg_0
    ; (как мы уже установили, это указатель)

    add     ecx, [ebp+arg_4]
    ; Складываем arg_0 с arg_4 (как мы установили, arg_4 - индекс)

    mov     byte ptr [ecx+1], 0
    ; Сумма ECX используется для косвенной адресации памяти, точнее, косвенно-базовой
    ; так как к сумме указателя и индекса прибавляется еще и единица и в эту ячейку
    ; памяти заносится ноль.
    ; Наши выводы подтверждаются - функции передается указатель на строку и
    ; индекс первого "отсекаемого" символа строки.
    ; Следовательно, для сохранения работоспособности программы по смещению 0x401000
    ; необходимо создать метку loc_s0, а PUSH 0x401000 в вызывающей функции
    ; заменить на PUSH offset loc_s0

```

```

        pop     ebp
        retn
MyFunc      endp

```

А теперь откомпилируем тот же самый пример компилятором Borland C++ 5.0 и сравним, чем он отличается от Microsoft Visual C++ (ниже для экономии места приведен код одной лишь функции MyFunc, функция main практически идентична предыдущему примеру):

Листинг 129

```

MyFunc      proc near          ; CODE XREF: _main+Dp
        push   ebp
        ; Отрываем пустой кадр стека - нет локальных переменных

        mov    byte ptr [eax+edx], 0Ah
        ; Ага, Borland C++ сразу сложил указатель с константой непосредственно в
        ; адресном выражении!
        ; Как определить, какой из регистров константа, а какой указатель?
        ; Как и в предыдущем случае, необходимо проанализировать их значение.

        mov    byte ptr [eax+edx+1], 0

        mov    ebp, esp
        pop    ebp
        ; Закрытие кадра стека

        retn
MyFunc      endp

```

Порядок индексов и указателей. Открою маленький секрет — при сложении указателя с константой большинство компиляторов на первое место помещают указатель, а на второе — константу, каким бы ни было их расположение в исходной программе.

Иначе говоря, выражения $a[i]$, $(a+i)[0]$, $*(a+i)$ и $*(i+a)$ компилируются в один и тот же код! Даже если извратиться и написать так: $(0)[i+a]$, компилятор все равно выдвинет a на первое место. Что это — ослиная упрямость, игра случая или фишка? Ответ до смешного прост — сложение указателя с константой дает указатель! Поэтому результат вычислений всегда записывается в переменную типа «указатель».

Вернемся к последнему рассмотренному примеру, применив для анализа наше новое правило:

Листинг 130

```

        mov    eax, [ebp+arg_0]
        ; Загружаем в EAX значение аргумента arg_0
        ; (arg_0 содержит непосредственное значение 0x405030)

        add    eax, [ebp+arg_4]
        ; Складываем EAX со значением аргумента arg_4 (он содержит значение 0x5).
        ; Операция сложения указывает на то, что по крайней мере один из них
        ; константа, а другой - либо константа, либо указатель

```

```
mov     byte ptr [eax], 0Ah
; Ага! Сумма непосредственных значений используется для косвенной адресации
; памяти, значит, это константа и указатель. Но кто из них кто?
; С большой степенью вероятности EAX - указатель, так как он стоит на первом
; месте, а var_4 - индекс, так как он стоит на втором.
```

Использование LEA для сложения констант. Инструкция LEA широко используется компиляторами не только для инициализации указателей, но и для сложения констант. Поскольку внутренне представление констант и указателей идентично, результат сложения двух указателей идентичен сумме тождественных им констант. То есть `LEA EBX, [EBX+0x666] == ADD EBX, 0x666`, однако по своим функциональным возможностям LEA значительно обгоняет ADD. Вот, например, `LEA ESI, [EAX*4+EBP-0x20]`, попробуйте то же самое «скормить» инструкции ADD!

Встретив в тексте программы команду LEA, не торопитесь навешивать на возвращенное ею значение ярлык «указатель», с не меньшим успехом он может оказаться и константой! Если «подозреваемый» ни разу не используется в выражении косвенной адресации — никакой это не указатель, а самая настоящая константа!

«Визуальная» идентификация констант и указателей. Вот несколько приемов, помогающих отличить указатели от констант.

1) В 32-разрядных Windows-программах указатели могут принимать ограниченный диапазон значений. Доступный процессорам регион адресного пространства начинается со смещения `0x1.00.00` и простирается до смещения `0x80.00.00.00`, а в среде Windows 9x/Me и того меньше — от `0x40.00.00` до `0x80.00.00.00`. Поэтому все непосредственные значения, меньшие `0x1.00.00` и больше `0x80.00.00`, представляют собой константы, а не указатели. Исключение составляет число ноль, обозначающее нулевой указатель некоторые защитные механизмы непосредственно обращаются к коду операционной системы, расположенному выше адреса `0x80.00.00`.

2) Если непосредственное значение смахивает на указатель, посмотрите, на что он указывает. Если по данному смещению находится пролог функции или осмысленная текстовая строка, скорее всего, мы имеем дело с указателем, хотя, может быть, это всего лишь совпадение.

3) Загляните в таблицу перемещаемых элементов (см. «Шаг четвертый. Знакомство с отладчиком. Способ 0. Бряк на оригинальный пароль»). Если адрес «подследственного» непосредственного значения есть в таблице, это, несомненно, указатель. Беда в том, что большинство исполняемых файлов — перемещаемы и такой прием актуален лишь для исследования DLL (а DLL перемещаемы по определению).

К слову сказать, дизассемблер IDA Pro использует все три описанных способа для автоматического опознавания указателей. Подробнее об этом рассказывается в книге автора «Образ мышления — дизассемблер IDA» (гл. «Настройки», с. 408).

Идентификация литералов и строк

Уже давно
Утихло поле боя,
Но сорок тысяч
Воинов Китая
Погибли здесь,
Пожертвовав собою...

*Ду Фо. Оплакиваю поражение
при Чэньтао*

Казалось бы, что может быть сложного в идентификации строк? Если то, на что ссылается указатель (см. «Идентификация указателей»), выглядит как строка, это и есть строка! Более того, в подавляющем большинстве случаев строки обнаруживаются и идентифицируются тривиальным просмотром дампа программы (при условии, конечно, что они не зашифрованы, но шифровка — тема отдельного разговора). Так-то оно так, да не все столь просто!

Задача номер один — автоматизированное выявление строк в программе — ведь не пролистывать же мегабайтовые дампы «вручную»? Существует множество алгоритмов идентификации строк. Самый простой (но не самый надежный) основан на двух следующих тезисах:

- *строка состоит из ограниченного ассортимента символов.* В грубом приближении — это цифры, буквы алфавита (включая пробелы), знаки препинания и служебные символы наподобие табуляции или возврата каретки;
- *строка должна состоять по крайней мере из нескольких символов.*

Условимся считать минимальную длину строки равной N байтам, тогда для автоматического выявления всех строк достаточно отыскать все последовательности из N и более «строковых» символов. Весь вопрос в том, чему должна быть равна N и какие символы включать в «строковые».

Если N имеет малое значение, порядка трех-четырех байтов, то мы получим очень большое количество ложных срабатываний. Напротив, когда N велико, порядка шести-восьми байтов, число ложных срабатываний близко к нулю и ими можно пренебречь, но все короткие строки, например OK, YES, NO, окажутся нераспознаваемыми! Другая проблема — помимо знакоцифровых символов в строках встречаются и элементы псевдографики (особенно часты они в консольных приложениях), и всякие там «мордашки», «стрелки», «карапузики» — словом, почти вся таблица ASCII. Чем же тогда строка отличается от случайной последовательности байтов? Частотный анализ здесь бессилён — ему для нормальной работы требуется как минимум сотня байтов текста, а мы говорим о строках из двух-трех символов!

Зайдем с другого конца — если в программе есть строка, значит, на нее кто-нибудь да ссылается. А раз так, можно поискать среди непосредственных значений указатель на распознанную строку. И если он будет найден, шансы на то, что это действительно именно строка, а не случайная последовательность байтов, резко возрастают. Все просто, не так ли?

Просто, да не совсем! Рассмотрим следующий пример:

Листинг 131

```
BEGIN
    WriteLn('Hello, Sailor!');
END.
```

Откомпилируем его любым подходящим Pascal-компилятором (например, Delphi или Free Pascal) и, загрузив откомпилированный файл в дизассемблер, пройдемся вдоль сегмента данных. Вскоре нам на глаза попадется следующее:

Листинг 132

```
.data:00404040 unk_404040      db 0Eh ;
.data:00404041                db 48h ; H
.data:00404042                db 65h ; e
.data:00404043                db 6Ch ; l
.data:00404044                db 6Ch ; l
.data:00404045                db 6Fh ; o
.data:00404046                db 2Ch ; ,
.data:00404047                db 20h ;
.data:00404048                db 53h ; S
.data:00404049                db 61h ; a
.data:0040404A                db 69h ; i
.data:0040404B                db 6Ch ; l
.data:0040404C                db 6Fh ; o
.data:0040404D                db 72h ; r
.data:0040404E                db 21h ; !
.data:0040404F                db 0 ;
.data:00404050 word_404050    dw 1332h
```

Вот она, искомая строка! (В том, что это строка, у нас никаких сомнений нет.) Попробуем найти, кто на нее ссылается. В IDA Pro для этого следует нажать **<ALT-I>** и в поле поиска ввести смещение начала строки — 0x404041...

Как это «ничего не найдено — Search Failed»? А что же тогда передается функции WriteLn? Может быть, это сбойнула IDA? Просматриваем дизассемблерный текст «вручную» — результат вновь нулевой.

Причина нашей неудачи в том, что в начале Pascal-строк идет байт, содержащий длину этой строки. Действительно, в дампе по смещению 0x404040 находится значение 0xE (четырнадцать в десятичной системе исчисления). А сколько символов строке «Hello, Sailor!»? Считаем: один, два, три... четырнадцать! Вновь нажимаем то же сочетание **<ALT-I>** и ищем непосредственный операнд, равный 0x404040. И в самом деле находим:

Листинг 133

```
.text:00401033      push    404040h
.text:00401038      push    [ebp+var_4]
.text:0040103B      push    0
.text:0040103D      call   FPC_WRITE_TEXT_SHORTSTR
.text:00401042      push    [ebp+var_4]
.text:00401045      call   FPC_WRITELN_END
.text:0040104A      push    offset loc_40102A
```



```
.text:0040104F      call     FPC_IOCHECK
.text:00401054      call     FPC_D0_EXIT
.text:00401059      leave
.text:0040105A      retn
```

Оказывается, мало идентифицировать строку, еще как минимум требуется определить ее границы.

Наиболее популярны следующие типы строк: **Си-строки**, завершающиеся нулем; **DOS-строки**, завершающиеся символом \$; **Pascal-строки**, предвараемые одно-, двух- или четырехбайтовым полем, содержащим длину строки. Рассмотрим каждый из этих типов подробнее:

С-строки, также именуемые ASCIIZ-строками (от Zero — ноль на конце), — весьма распространенный тип строк, широко использующийся в операционных системах семейств Windows и UNIX. Символ \0 (не путать с 0) имеет специальное предназначение и трактуется по-особому — как *завершитель строки*. Длина ASCIIZ-строк практически ничем не ограничена, ну разве что размером адресного пространства, выделенного процессу или протяженностью сегмента. Соответственно в Windows 9x/NT максимальный размер ASCIIZ-строки лишь немногим менее **2 гигабайтов**, а в Windows 3.1 и MS-DOS — около **64 килобайтов**. Фактическая длина ASCIIZ-строк лишь на байт длиннее исходной ASCII-строки. Несмотря на перечисленные выше достоинства, С-строкам присущи и некоторые недостатки. Во-первых, ASCIIZ-строка не может содержать нулевых байтов, и поэтому она непригодна для обработки бинарных данных. Во-вторых, операции копирования, сравнения и контакции С-строк сопряжены со значительными накладными расходами — современным процессорам невыгодно работать с отдельными байтами, им желательно иметь дело с двойными словами. Но, увы, длина ASCIIZ-строк наперед неизвестна и ее приходится вычислять на лету, проверяя **каждый байт** на символ завершения. Правда, разработчики некоторых компиляторов идут на хитрость — они завершают строку **семью нулями**, что позволяет работать с двойными словами, а это на порядок быстрее. Почему семью, а не четыремя? Ведь в двойном слове байтов четыре! Да, верно, четыре, но подумайте, что произойдет, если последний значимый символ строки придется на первый байт двойного слова? Верно, его конец заполнят три нулевых байта, но двойное слово из-за вмешательства первого символа уже не будет равно нулю! Вот поэтому следующему двойному слову надо предоставить еще четыре нулевых байта, тогда оно гарантировано будет равно нулю. Впрочем, семь служебных байтов на каждую строку — это уже перебор!

DOS-строки. В MS-DOS функция вывода строки воспринимает знак \$ как символ завершения, поэтому в программистских кулуарах такие строки называют DOS-строками. Термин не совсем корректен — все остальные функции MS-DOS работают исключительно с ASCIIZ-строками! Причина выбора столь странного выбора символа-разделителя восходит к тем древнейшим временам, когда никакого графического интерфейса еще и в помине не существовало, а консольный терминал считался весьма продвинутой системой взаимодействия с пользователем. Клавиша <Enter> не могла служить завершителем строки, так как подчас прихо-

дилось вводить в программу несколько строк сразу. Комбинации **<Ctrl-Z>** или **<Alt-000>** также не годились — на многих клавиатурах тех лет отсутствовали такие регистры! С другой стороны, компьютеры использовались главным образом для инженерных, а не бухгалтерских расчетов, и символ «бакса» был самым мало употребляемым символом — вот и решили использовать его для сигнализации о завершении пользователем ввода и как символ-завершитель строки. (Да, символ-завершитель вводился пользователем, а не добавлялся программой, как это происходит с ASCIIZ-строками.) В настоящее время DOS-строки практически вышли из употребления и читатель вряд ли с ними столкнется...

Pascal-строки. Pascal-строки не имеют завершающего символа, вместо этого они предваряются специальным полем, содержащим длину этой строки. Достоинства этого подхода: возможность хранения любых символов в строке (в том числе и нулевых байтов!) и высокая скорость обработки строковых переменных. Вместо постоянной проверки каждого байта на завершающий символ происходит лишь одно обращение к памяти — загрузка длины строки. Ну а раз длина строки известна, можно работать не с байтами, а двойными словами — «родным» типом данных 32-разрядных процессоров. Весь вопрос в том, сколько байтов отвести под поле размера. Один? Что ж, это экономно, но тогда максимальная длина строки будет ограничена 255 символами, что во многих случаях оказывается явно недостаточно! Этот тип строк используют практически все Pascal-компиляторы (например, Borland Turbo Pascal, Free Pascal), поэтому-то такие строки и называют Pascal-строками или, если более точно, *короткими Pascal-строками*.

Delphi-строки. Осознавая очевидную смехотворность ограничения длины Pascal-строк 255 символами, разработчики Delphi расширили поле размера до двух байтов, увеличив тем самым максимально возможную длину до 65 535 символов. Хотя такой тип строк поддерживают и другие компиляторы (тот же Free Pascal, к примеру), в силу сложившейся традиции их принято именовать *Delphi-строками* или Pascal-строками с двухбайтовым полем размера — *двухбайтовыми Pascal-строками*.

Ограничение в шестьдесят с гаком килобайтов и ограничением язык назвать не поворачивается. Большинство строк имеют гораздо меньшую длину, а для обработки больших массивов данных (текстовых файлов, к примеру) есть куча (динамическая память) и ряд специализированных функций. Накладные же расходы (два служебных байта на каждую строковую переменную) не столь велики, чтобы их брать в расчет. Словом, Delphi-строки, сочетая в себе лучше стороны C- и Pascal-строк (практически неограниченную длину и высокую скорость обработки соответственно), представляются самым удобным и практичным типом.

Wide-Pascal-строки. «Широкие» Pascal-строки отводят на поле размера аж четыре байта, «ограничивая» максимально возможную длину 4 294 967 295 символами или 4 гигабайтами, что даже больше того количества памяти, которое Windows NT/9x выделяют в «личное пользование» прикладному процессу! Однако за эту роскошь приходится дорого платить, отдавая каждой строке четыре лишние байта, три из которых в большинстве случаев будут попросту пустовать.

Накладные расходы на коротких строках становятся весьма велики, поэтому тип Wide-Pascal практически не используется.

Комбинированные типы. Некоторые компиляторы используют комбинированный C+Pascal тип, что позволяет им, с одной стороны, достичь высокой скорости обработки строк и хранить в строках любые символы, а с другой — обеспечить совместимость с огромным количеством C-библиотек, «заточенных» под ASCIIZ-строки. Каждая комбинированная строка принудительно завершается нулем, но этот ноль в саму строку не входит и штатные библиотеки (операторы) языка работают с ней как с Pascal-строкой. При вызове же функций Си-библиотек компилятор передает им указатель не на истинное начало строки, а на первый символ строки.

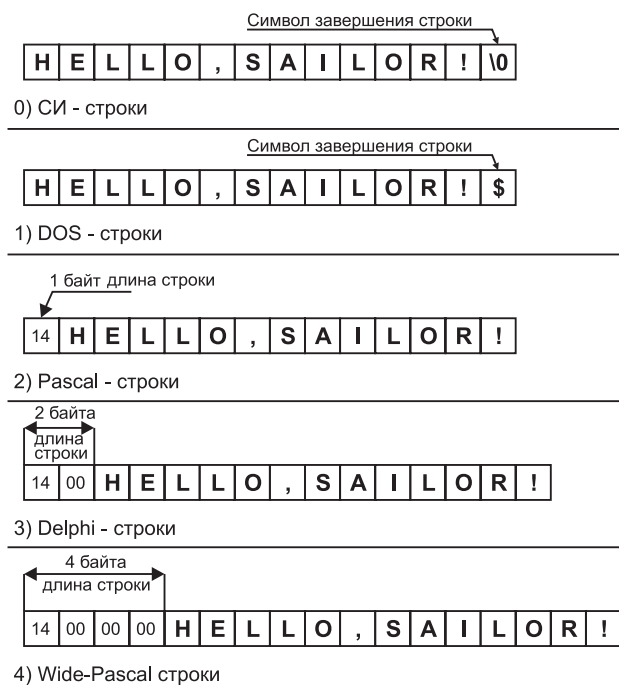


Рис. 21. Основные типы строк

Определение типа строк. По внешнему виду строки определить ее тип весьма затруднительно. Наличие завершающего нуля в конце строки еще не повод считать ее ASCIIZ-строкой (Pascal-компиляторы в конец строк частенько дописывают один или несколько нулей для выравнивания данных по кратным адресам), а совпадение предшествующего строке байта с ее длиной может действительно быть лишь случайным совпадением.

Грубо тип строки определяется по роду компилятора (Си или Pascal), а точно — по алгоритму обработки этой строки (т. е. анализом манипулирующего с ней кода). Рассмотрим следующий пример:

Листинг 134. Пример, демонстрирующий идентификацию типа строк

```

VAR
    s0, s1 : String;

BEGIN
    s0 := 'Hello, Sailor!';
    s1 := 'Hello, World!';
    IF s0=s1 THEN WriteLN('OK') ELSE WriteLn('Wooz1');
END.

```

Откомпилировав его компилятором Free Pascal, заглянем в сегмент данных. Там мы найдем следующую строку:

```
.data:00404050 aHelloWorld      db 0Dh,'Hello, World! ',0 ; DATA XREF: _main+2B↑o
```

Не правда ли, она очень похожа на ASCIIZ-строку? Кому не известен используемый компилятор, тому и на ум не придет, что 0xD — это поле длины, а не символ переноса! Чтобы проверить нашу гипотезу насчет типа, перейдем по перекрестной ссылке, любезно обнаруженной IDA Pro, или самостоятельно найдем в дизассемблированном тексте непосредственный операнд 0x404050 (смещение строки).

```

push    offset _S1                      ; Передаем указатель на строку-приемник
push    offset aHelloWorld ; "\rHello, World!" Передаем указатель на строку-источник
push    OFFh                            ; Макс. длина строки
call    FPC_SHORTSTR_COPY

```

Так-с, указатель на строку передается функции FPC_SHORTSTR_COPY. Из прилагаемой к Free Pascal документации можно узнать, что эта функция работает с короткими Pascal-строками, стало быть, байт 0xD никакой не символ переноса, а длина строки. А что бы мы делали, если бы у нас отсутствовала документация на Free Pascal? (В самом же деле, невозможно раздобыть все-все-все компиляторы!) Кстати, штатная поставка IDA Pro, вплоть до версии 4.17 включительно, не содержит сигнатур FPP-библиотек и их приходится создавать самостоятельно.

В тех случаях, когда строковая функция неопознана или отсутствует ее описание, путь один — исследовать код на предмет выяснения алгоритма его работы. Ну что, засучим рукава и приступим?

Листинг 135

```

FPC_SHORTSTR_COPY      proc near                      ; CODE XREF: sub_401018+21p

arg_0                  = dword ptr  8                  ; Макс. длина строки
arg_4                  = dword ptr  0Ch                ; Исходная строка
arg_8                  = dword ptr  10h                ; Целевая строка

    push    ebp
    mov     ebp, esp
    ; Открываем кадр стека

    push    eax
    push    ecx
    ; Сохраняем регистры

```

```
cld
; Сбрасываем флаг направления,
; т. е. заставляем команды LODS, STOS, MOVS инкрементировать регистр-указатель

mov     edi, [ebp+arg_8]
; Загружаем в регистр EDI значение аргумента arg_8
; (смещение целевого буфера)

mov     esi, [ebp+arg_4]
; Загружаем в регистр ESI значение аргумента arg_4
; (смещение исходной строки)

xor     eax, eax
; Обнуляем регистр EAX

mov     ecx, [ebp+arg_0]
; Загружаем в ECX значение аргумента arg_0 (макс. допустимая длина строки)

lodsb
; Загружаем в AL первый байт исходной строки, на которую указывает регистр ESI,
; и увеличиваем ESI на единицу

cmp     eax, ecx
; Сравниваем первый символ строки с макс. возможной длиной строки.
; Уже ясно, что первый символ строки - длина, однако притворимся, что мы
; не знаем назначения аргумента arg_0, и продолжим анализ

jbe     short loc_401168
; if (ESI[0] <= arg_0) goto loc_401168

mov     eax, ecx
; Копируем в EAX значение ECX

loc_401168:                                     ; CODE XREF: sub_401150+14j
stosb
; Записываем первый байт исходной строки в целевой буфер
; и увеличиваем EDI на единицу

cmp     eax, 7
; Сравниваем длину строки с константой 0x7

jl      short loc_401183
; Длина строки меньше семи байтов?
; Тогда и копируем ее побайтно!

mov     ecx, edi
; Загружаем в ECX значение указателя на целевой буфер, увеличенный на единицу
; (его увеличила команда STOSB при записи байта)

neg     ecx
; Дополняем ECX до нуля, NEG(0xFFFF) = 1;
; ECX :=1

and     ecx, 3
; Оставляем в ECX три младших бита, остальные - сбрасываем
; ECX :=1

sub     eax, ecx
; Отнимаем от EAX (содержит первый байт строки) "кастрированный" ECX
```

```

repe    movsb
; Копируем ECX байт из исходной строки в целевой буфер, передвигая ESI и EDI.
; В нашем случае мы копируем 1 байт

mov     ecx, eax
; Теперь ECX содержит значение первого байта строки, уменьшенное на единицу

and     eax, 3
; Оставляем в EAX три младших бита, остальные - сбрасываем

shr     ecx, 2
; Циклическим сдвигом, делим ECX на четыре (22 = 4)

repe    movsd
; Копируем ECX двойных байтов из ESI в EDI.
; Теперь становится ясно, что ECX содержит длину строки, а поскольку
; в ECX загружается значение первого байта строки, можно с полной уверенностью
; сказать, что первый байт строки (причем именно байт, а не слово) содержит
; длину этой строки.
; Таким образом, это короткая Pascal-строка
;

loc_401183:                                ; CODE XREF: sub_401150+10j
mov     ecx, eax
; Если длина строки менее семи байтов, то EAX содержит длину строки для ее
; побайтового копирования (см. условный переход jbe short loc_401168).
; В противном случае EAX содержит остаток "хвоста" строки, который не смог
; заполнить собой последнее двойное слово.
; В общем, так или иначе, в ECX загружается количество байтов для копирования

repe    movsb
; Копируем ECX байт из ESI в EDI

pop     ecx
pop     eax
; Восстанавливаем регистры

leave
; Закрываем кадр стека

ret     0Ch
FPC_SHORTSTR_COPY    endp

```

А теперь познакомимся с Си-строками, для чего нам пригодится следующий пример:

Листинг 136

```

#include <stdio.h>
#include <string.h>

main()
{
    char s0[]="Hello, World!";
    char s1[]="Hello, Sailor!";
    if (strcmp(&s0[0],&s1[0])) printf("Woozl\n"); else printf("OK\n");
}

```

Откомпилируем его любым подходящим Си-компилятором, например Borland C++ 5.0 (внимание — Microsoft Visual C++ для этой цели не подходит, см. «*Turbo-инициализация строковых переменных*»), и поищем наши строки в сегменте данных.

Долго искать не приходится, вот они:

```
DATA:00407074 aHelloWorld      db 'Hello, World!',0      ; DATA XREF: _main+16↑o
DATA:00407082 aHelloSailor     db 'Hello, Sailor!',0      ; DATA XREF: _main+22↑o
DATA:00407091 aWooz1          db 'Wooz1',0Ah,0          ; DATA XREF: _main+4F↑o
DATA:00407098 aOk              db 'OK',0Ah,0              ; DATA XREF: _main+5C↑o
```

Обратите внимание, строки следуют вплотную друг к другу, каждая из них завершается символом нуля, и значение первого байта строки не совпадает с ее длиной. Несомненно, перед нами ASCIIZ-строки, однако не мешает лишний раз убедиться в этом, тщательно проанализировав манипулирующий с ними код:

Листинг 137

```
_main      proc near                ; DATA XREF: DATA:00407044o
var_20     = byte ptr -20h
var_10     = byte ptr -10h

    push    ebp
    mov     ebp, esp
    ; Открываем кадр стека

    add     esp, 0FFFFFFE0h
    ; Резервируем место для локальных переменных

    mov     ecx, 3
    ; Заносим в регистр ECX значение 0x3

    lea     eax, [ebp+var_10]
    ; Загружаем в EAX указатель на локальный буфер var_10

    lea     edx, [ebp+var_20]
    ; Загружаем в EDX указатель на локальный буфер var_20

    push    esi
    ; Сохраняем регистр ESI.
    ; Именно сохраняем, а не передаем функции, так как ESI еще не был инициализирован!

    push    edi
    ; Сохраняем регистр EDI

    lea     edi, [ebp+var_10]
    ; Загружаем в EDI указатель на локальный буфер var_10

    mov     esi, offset aHelloWorld ; "Hello, World!"
    ; IDA распознала в непосредственном операнде смещение строки "Hello,World!".
    ; А если бы и не распознала, это бы сделали мы сами, основываясь на том, что:
    ; 1) непосредственный операнд совпадает со смещением строки;
    ; 2) следующая команда неявно использует ESI для косвенной адресации памяти,
    ; следовательно, в ESI загружается указатель
```

```

repe    movsd
; Копируем ECX двойных слов из ESI в EDI.
; Чему равно ECX? Оно равно 0x3.
; Для перевода из двойных слов в байты умножаем 0x3 на 0x4 и получаем 0xC,
; что на байт короче копируемой строки "Hello,World!", на которую указывает ESI

movsw
; Копируем последний байт строки "Hello, World!" вместе с завершающим нулем

lea     edi, [ebp+var_20]
; Загружаем в регистр EDI указатель на локальный буфер var_20

mov     esi, offset aHelloSailor      ; "Hello, Sailor!"
; Загружаем в регистр ESI указатель на строку "Hello, Sailor!"

mov     ecx, 3
; Загружаем в ECX количество полных двойных слов в строке "Hello, Sailor!"

repe    movsd
; Копируем 0x3 двойных слова

movsw
; Копируем слово

movsb
; Копируем последний завершающий байт

; // Функция сравнения строк
loc_4010AD:                                ; CODE XREF: _main+4Bj
mov     cl, [eax]
; Загружаем в CL содержимое очередного байта строки "Hello, World!"

cmp     cl, [edx]
; CL равен содержимому очередного байта строки "Hello, Sailor!"?

jnz     short loc_4010C9
; Если символы обеих строк не равны, переходим к метке loc_4010C9

test    cl, cl
jz      short loc_4010D8
; Регистр CL равен нулю? (В строке встретился нулевой символ?)
; Если так, то прыгаем на loc_4010D8
; Теперь мы можем безошибочно определить тип строки,
; во-первых, первый байт строки содержит первый символ строки,
; а не хранит ее длину,
; во-вторых, каждый байт строки проверяется на завершающий нулевой символ.
; Значит, это ASCIIZ-строки!

mov     cl, [eax+1]
; Загружаем в CL следующий символ строки "Hello, World!"

cmp     cl, [edx+1]
; Сравняем его со следующим символом "Hello, Sailor!"

jnz     short loc_4010C9
; Если символы не равны - закончить сравнение

add     eax, 2
; Переместить указатель строки "Hello, World!" на два символа вперед

```



```

    add     edx, 2
    ; Переместить указатель строки "Hello, Sailor!" на два символа вперед

    test    cl, cl
    jnz     short loc_4010AD
    ; Повторять сравнение, пока не будет достигнут символ-завершитель строки

loc_4010C9:                                ; CODE XREF: _main+35j      _main+41j
    jz      short loc_4010D8
    ; (см. "Идентификация if - then - else")

    ; // Вывод строки "Woozl"
    push    offset aWoozl ; format
    call    _printf
    pop     ecx
    jmp     short loc_4010E3

loc_4010D8:                                ; CODE XREF: _main+39j      _main+4Dj
    ; // Вывод строки "OK"
    push    offset aOk ; format
    call    _printf
    pop     ecx

loc_4010E3:                                ; CODE XREF: _main+5Aj
    xor     eax, eax
    ; Функция возвращает ноль

    pop     edi
    pop     esi
    ; Восстанавливаем регистры

    mov     esp, ebp
    pop     ebp
    ; Закрываем кадр стека

    retn

_main     endp

```

Turbo-инициализация строковых переменных. Не всегда, однако, различить строки так просто. Чтобы убедиться в этом, достаточно откомпилировать предыдущий пример компилятором Microsoft Visual C++ и заглянуть в полученный файл любым подходящим дизассемблером, скажем, IDA Pro.

Так, переходим в секцию данных, прокручиваем ее вниз до тех пор, пока не устанет рука (а когда устанет — кирпич на Page Down!) и... Woozl! — никаких следов присутствия строк «Hello, Sailor!» и «Hello, World!». Зато обращает на себя внимание какая-то странная града двойных слов, смотрите:

```

.data:00406030 dword_406030      dd 6C6C6548h          ; DATA XREF: main+6↑r
.data:00406034 dword_406034      dd 57202C6Fh          ; DATA XREF: main +E↑r
.data:00406038 dword_406038      dd 646C726Fh          ; DATA XREF: main +17↑r
.data:0040603C word_40603C       dw 21h              ; DATA XREF: main +20↑r
.data:0040603E                                align 4
.data:00406040 dword_406040      dd 6C6C6548h          ; DATA XREF: main +2A↑r
.data:00406044 dword_406044      dd 53202C6Fh          ; DATA XREF: main +33↑r
.data:00406048 dword_406048      dd 6F6C6961h          ; DATA XREF: main +3C↑r

```

```
.data:0040604C word_40604C      dw 2172h          ; DATA XREF: main +44↑r
.data:0040604E byte_40604E     db 0              ; DATA XREF: main +4F↑r
```

Что бы это значило? Это не указатели — они никуда не указывают, это не переменные типа `int` — мы не объявляли таких в программе. Жмем **<F4>** для перехода в `hex`-режим, и что мы видим? Вот они, наши строки, вот они, родимые:

```
.data:00406030 48 65 6C 6C 6F 2C 20 57-6F 72 6C 64 21 00 00 00 "Hello, World!..."
.data:00406040 48 65 6C 6C 6F 2C 20 53-61 69 6C 6F 72 21 00 00 "Hello, Sailor!..."
.data:00406050 57 6F 6F 7A 6C 0A 00 00-4F 4B 0A 00 00 00 00 00 "Woozle...OK...."
```

Хм, почему же тогда IDA Pro посчитала их двойными словами? Ответить на вопрос поможет анализ манипулирующего со строкой кода, но прежде чем приступить к его исследованию, превратим эти двойные слова в нормальную ASCII-строку. (клавиша **<U>** для преобразования двойных слов в цепочку бестиповых байт и клавиша **<A>** для преобразования ее в строку.) Затем подведем курсор к первой перекрестной ссылке и нажмем клавишу **<Enter>**:

Листинг 138

```
main          proc near          ; CODE XREF: start+AFp

var_20        = byte ptr -20h
var_1C        = dword ptr -1Ch
var_18        = dword ptr -18h
var_14        = word ptr -14h
var_12        = byte ptr -12h
var_10        = byte ptr -10h
var_C         = dword ptr -0Ch
var_8         = dword ptr -8
var_4         = word ptr -4

; Откуда взялось столько локальных переменных?!

    push     ebp
    mov      ebp, esp
    ; Открываем кадр стека

    sub      esp, 20h
    ; Резервируем память для локальных переменных

    mov      eax, dword ptr aHelloWorld ; "Hello, World!"
    ; Загружаем в EAX... нет, не указатель на строку "Hello, World!", а
    ; четыре первых байта этой строки! Теперь понятно, почему ошиблась IDA Pro
    ; и оригинальный код (до преобразования строки в строку) выглядел так:
    ; mov eax, dword_406030
    ; Не правда ли, не очень наглядно? И если бы мы изучали не свою, а чужую
    ; программу, этот трюк дизассемблера ввел бы нас в заблуждение!

    mov      dword ptr [ebp+var_10], eax
    ; Копируем четыре первых байта строки в локальную переменную var_10

    mov      ecx, dword ptr aHelloWorld+4
    ; Загружаем байты с четвертого по восьмой строки "Hello, World!" в ECX

    mov      [ebp+var_C], ecx
    ; Копируем их в локальную переменную var_C. Но мы-то уже знаем, что это
```

```

; никакая не переменная var_C, а часть строкового буфера

mov     edx, dword ptr aHelloWorld+8
; Загружаем байты с восьмого по двенадцатый строки "Hello, World!" в EDX

mov     [ebp+var_8], edx
; Копируем их в локальную переменную var_8, точнее, в строковой буфер

mov     ax, word ptr aHelloWorld+0Ch
; Загружаем оставшийся двухбайтовый хвост строки в AX

mov     [ebp+var_4], ax
; Записываем его в локальную переменную var_4.
; Итак, строка копируется по частям в следующие локальные переменные:
; int var_10; int var_0C; int var_8; short int var_4
; Следовательно, на самом деле есть только одна локальная переменная -
; char var_10[14]

mov     ecx, dword ptr aHelloSailor      ; "Hello, Sailor!"
; Прodelываем ту же самую операцию копирования над строкой "Hello, Sailor!"

mov     dword ptr [ebp+var_20], ecx
mov     edx, dword ptr aHelloSailor+4
mov     [ebp+var_1C], edx
mov     eax, dword ptr aHelloSailor+8
mov     [ebp+var_18], eax
mov     cx, word ptr aHelloSailor+0Ch
mov     [ebp+var_14], cx
mov     dl, byte_40604E
mov     [ebp+var_12], dl
; Копируем строку "Hello, Sailor!" в локальную переменную char var_20[14]

lea     eax, [ebp+var_20]
; Загружаем в регистр EAX указатель на локальную переменную var_20,
; которая (как мы помним) содержит строку "Hello, Sailor!".

push     eax                ; const char *
; Передаем ее функции strcmp.
; Из этого можно заключить, что var_20 действительно хранит строку,
; а не значение типа int

lea     ecx, [ebp+var_10]
; Загружаем в регистр ECX указатель на локальную переменную var_10,
; хранящую строку "Hello, World!",

push     ecx                ; const char *
; Передаем ее функции strcmp

call     _strcmp
add     esp, 8
; strcmp("Hello, World!", "Hello, Sailor!")

test     eax, eax
jz       short loc_40107B
; Строки равны?

; // Вывод на экран строки "Wooz1"
push     offset aWooz1 ; "Wooz1\n"

```

```

    call    _printf
    add     esp, 4
    jmp     short loc_401088

; // Вывод на экран строки "OK"
loc_40107B:                                ; CODE XREF: sub_401000+6Aj
    push    offset a0k ; "OK\n"
    call    _printf
    add     esp, 4

loc_401088:                                ; CODE XREF: sub_401000+79j
    mov     esp, ebp
    pop     ebp
    ; Закрываем кадр стека

    retn

main                                         endp

```

Идентификация условных операторов if-then-else

Значение каждого элемента текста определяется контекстом его употребления. Текст не описывает мир, а вступает в сложные взаимоотношения с миром.

Тезис аналитической философии

Существует два вида алгоритмов — *безусловные* и *условные*. Порядок действий безусловного алгоритма всегда постоянен и не зависит от входных данных. Например, $a=b+c$. Порядок действий условных алгоритмов, напротив, зависит от входных данных. Например: **если** s не равно нулю, **то**: $a=b/c$; **иначе**: вывести сообщение об ошибке.

Обратите внимание на выделенные жирным шрифтом ключевые слова «если», «то» и «иначе», называемые *операторами условия* или *условными операторами*. Без них не обходится ни одна программа (вырожденные примеры наподобие «Hello, World!» — не в счет). Условные операторы — сердце любого языка программирования. Поэтому чрезвычайно важно уметь их правильно идентифицировать.

В общем виде (не углубляясь в синтаксические подробности отдельных языков программирования) оператор условия схематично изображается так:

```
IF (условие) THEN { оператор1; оператор2; } ELSE { оператора; операторb; }
```

Задача компилятора — преобразовать эту конструкцию в последовательность машинных команд, выполняющих оператор₁, оператор₂, если условие истинно, и соответственно оператор_а, оператор_б — если оно ложно. Однако микропроцессоры серии 80x86 поддерживают весьма скромный набор условных команд, ограниченный фактически одними условными переходами (касательно исключений см. «Оптимизация ветвлений»). Программистам, знакомым лишь с IBM PC, та-

кое ограничение не покажется чем-то неестественным, между тем существует масса процессоров, поддерживающих *префикс условного выполнения инструкции*. То есть, вместо того, чтобы писать: `TEST ECX,ECX/JNZ xxx/MOV EAX,0x666`, там поступают так: `TEST ECX,ECX/IFZ MOV EAX,0x666`. IFZ и есть префикс условного выполнения, разрешающий выполнение следующей команды только в том случае, если установлен флаг нуля.

В этом смысле микропроцессоры 80x86 можно сравнить с ранними диалектами языка Бейсика, не разрешающими использовать в условных выражениях никакой другой оператор, кроме GOTO. Сравните:

Листинг 139. Новый диалект Бейсика. Старый диалект Бейсика

IF A=B THEN PRINT "A=B"	10 IF A=B THEN GOTO 30
	20 GOTO 40
	30 PRINT "A=B"
	40 ... // прочий код программы

Если вы когда-нибудь программировали на старых диалектах Бейсика, то, вероятно, помните, что гораздо выгоднее выполнять **GOTO**, если условие ложно, а в противном случае продолжать нормальное выполнение программы. (Как видите, вопреки расхожему мнению, навыки программирования на Бейсике отнюдь не бесполезны, особенно в дизассемблировании программ.)

Большинство компиляторов (даже не оптимизирующих) инвертируют истинность условия, транслируя конструкцию **IF (условие) THEN {оператор₁; оператор₂}** в следующий псевдокод:

Листинг 140

```
IF (NOT условие) THEN continue
оператор1;
оператор2;
continue;
...
```

Следовательно, для восстановления исходного текста программы нам придется снова инвертировать *условие* и «подцепить» блок операторов {оператор₁; оператор₂} к ключевому слову THEN. Иначе говоря, если откомпилированный код выглядит так:

Листинг 141

```
10 IF A<>B THEN 30
20 PRINT "A=B"
30 ...// прочий код программы
```

можно с уверенностью утверждать, что в исходном тексте присутствовали следующие строки: `IF A=B THEN PRINT «A=B»`. А если, программист, наоборот, проверял переменные A и B на неравенство, т. е. `IF A<>B THEN PRINT «A<>B»`? Все равно компилятор инвертирует истинность условия и сгенерирует следующий код:

Листинг 142

```
10 IF A=B THEN 30
20 PRINT "A<>B"
30 ...// прочий код программы
```

Конечно, встречаются и дебильные компиляторы, страдающие многословием. Их легко распознать по безусловному переходу, следующему сразу же после условного оператора:

Листинг 143

```
IF (условие) THEN do
GOTO continue
do:
оператор1;
оператор2;
continue:
```

В таком случае инвертировать условие не нужно. Впрочем, если это сделать, ничего страшного не произойдет, разве что код программы станет менее понятным, да и то не всегда.

Рассмотрим теперь, как транслируется полная конструкция **IF (условие) THEN {оператор₁; оператор₂;} ELSE {оператор_а; оператор_б;}** . Одни компиляторы поступают так:

Листинг 144

```
IF (условие) THEN do_it
// Ветка ELSE
оператора;
операторб
GOTO continue

do_it:
//Ветка IF
оператор1;
оператор2;
continue:
```

А другие так:

```
IF (NOT условие) THEN else
//Ветка IF
оператор1;
оператор2;
GOTO continue

else:
// Ветка ELSE
оператора;
операторб
continue:
```

Разница между ними в том, что вторые инвертируют истинность условия, а первые — нет. Поэтому, не зная «нрава» компилятора, определить, как выглядел подлинный исходный текст программы, невозможно! Однако это не создает проблем, ибо условие всегда можно записать так, как это удобно. Допустим, не нравится вам конструкция `IF (c<>0) THEN a=b/c ELSE PRINT «Ошибка!»` Пишите ее так: `IF (c==0) THEN PRINT «Ошибка!» ELSE a=b/c` — и никаких гвоздей!

Типы условий. Условия делятся на *простые (элементарные)* и *сложные (составные)*. Пример первых — `if (a==b)...`, вторых — `if ((a==b) && (a!=0))...` Очевидно, что *любое сложное условие можно разложить на ряд простых условий*. Вот с простых условий мы и начнем.

Существуют два основных типа элементарных условий: *условия отношений* («меньше», «равно», «больше», «меньше или равно», «не равно», «больше или равно», соответственно обозначаемые как: «<», «==», «>», «<=», «!=», «>=») и *логические условия* («И», «ИЛИ», «НЕ», «И исключаящее ИЛИ», в Си-нотации соответственно обозначаемые так: «&», «|», «!», «^»). Известный хакерский авторитет Мэтт Питрек приплетает сюда и проверку битов, однако несколько некорректно смешивать в одну кучу людей и коней, даже если они чем-то и взаимосвязаны. Поэтому о битовых операциях мы поговорим отдельно в одноименной главе.

Если условие истинно, оно возвращает булево значение TRUE, соответственно если ложно — FALSE. Внутреннее (физическое) представление булевых переменных зависит от конкретной реализации и может быть любым. По общепринятому соглашению FALSE **равно** нулю, а TRUE **не равно** нулю. Часто (но не всегда) TRUE равно единице, но на это нельзя полагаться! Так, код `IF ((a>b)!=0)...` абсолютно корректен, а: `IF ((a>b)==1)...` привязан к конкретной реализации и поэтому нежелателен.

Обратите внимание: `IF ((a>b)!=0)...` проверяет на неравенство нулю отнюдь не значения самих переменных `a` и `b`, а именно результат их сравнения. Рассмотрим следующий пример: `IF ((666==777)==0) printf("Wooz!")`. Как вы думаете, что отобразится на экране, если его запустить? Правильно — "Wooz!" Почему? Ведь ни 666, ни 777 не равно нулю! Да, но ведь `666 != 777`, следовательно, условие `(666==777)` — ложно, следовательно, равно нулю. Кстати, если записать `IF ((a==b)==0)...` получится совсем иной результат — значение переменной `b` будет присвоено переменной `a` и потом проверено на равенство нулю.

Логические условия чаще всего используются для связывания двух или более элементарных условий отношения в составное. Например, `IF ((a==b) && (a!=0))...` При трансляции программы компилятор всегда выполняет развертку составных условий в простые. В данном случае это происходит так: `IF a==b THEN IF a=0 THEN...` На втором этапе выполняется замена условных операторов на оператор `GOTO`:

Листинг 145

```
IF a!=b THEN continue
IF a==0 THEN continue
...// код условия
```

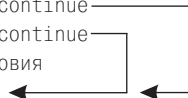
```
:continue
...// прочий код
```

Порядок вычисления элементарных условий в сложном выражении зависит от прихотей компилятора, гарантируется лишь, что условия, «связанные» операцией логического «И», проверяются слева направо в порядке их объявления в программе. Причем если первое условие ложно, то следующее за ним вычислено **не будет!** Это дает возможность писать код наподобие следующего: `if ((filename) & (f=fopen(&filename[0], "rw")))`... — если указатель `filename` указывает на невыделенную область памяти (т. е., попросту говоря, содержит ноль — логическое FALSE), функция `fopen` не вызывается и ее краха не происходит. Такой способ вычислений получил название *быстрых булевых операций* (теперь-то вы знаете, что подразумевается под «быстротой»).

Перейдем теперь к вопросу идентификации логических условий и анализу сложных выражений. Вернемся к уже облюбованному нами выражению `if ((a==b) && (a!=0))`... и взглянемся в результат его трансляции:

Листинг 146

```
IF a!=b THEN continue
IF a==0 THEN continue
...// код условия
:continue
...// прочий код
```

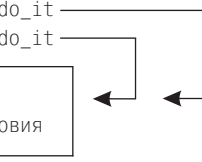


Легко видеть — он выдает себя серией условных переходов к одной и той же метке, причем, обратите внимание, выполняется проверка на **неравенство** каждого из элементарных условий, а сама метка расположена **позади** кода условия.

Идентификация логической операции «ИЛИ» намного сложнее в силу неоднозначности ее трансляции. Рассмотрим это на примере выражения `if ((a==b) || (a!=0))`... Его можно разбить на элементарные операции и так:

Листинг 147

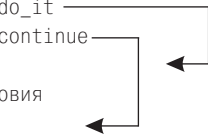
```
IF a==b THEN do_it
IF a!=0 THEN do_it
goto continue
:do_it
...// код условия
:continue
...// прочий код
```



Итак:

Листинг 148

```
IF a==b THEN do_it
IF a==0 THEN continue
:do_it
...// код условия
:continue
...// прочий код
```



Первый вариант обладает весьма запоминающейся внешностью — серия проверок (без инверсии условия) на одну и ту же метку, расположенную перед кодом условия, а в конце этой серии — безусловный переход на метку, расположенную позади кода условия.

Однако оптимизирующие компиляторы выкидывают безусловный переход, инвертируя проверку последнего условия в цепочке и соответственно меняя адрес перехода. По неопытности эту конструкцию часто принимают за смесь OR и AND. Кстати, о смещенных операциях. Рассмотрим результат трансляции следующего выражения: `if ((a==b) || (a==c) && a!=0)...:`

Листинг 149

```
IF a==b THEN check_null
IF a!=c THEN continue
check_null:
IF a==0 THEN continue
...// код условия
continue:
...// прочий код
```

Как из непроходимого леса элементарных условий получить одно удобочитаемое составное условие? Начинаем плясать от печки, т. е. от первой операции сравнения. Смотрите, если условие `a==b` окажется истинно, оно «выводит из игры» проверку условия `a!=c`. Такая конструкция характерна для операции OR, т. е. достаточно выполнения хотя бы одного условия из двух для срабатывания кода. Пишем в уме или карандашом: `if ((a==b) || ...)`, далее — если условие `(a!=c)` истинно, все дальнейшие проверки прекращаются, и происходит передача управления на метку, расположенную позади условного кода. Логично предположить, что мы имеем дело с последней операцией OR в цепочке сравнений — это ее почерк. Значит, мы инвертируем условие выражения и продолжаем писать: `if ((a==b) || (a==c)...)...`. Последний бастион — проверка условия `a==0`. Выполнить условный код, миновав условие, не удастся, следовательно, это не OR, а AND! А AND всегда инвертирует условие срабатывания, и поэтому оригинальный код должен был выглядеть так: `if ((a==b) || (a==c) && (a!=0))`. Ура! У нас получилось!

Впрочем, как любил поговаривать Дмитрий Николаевич, не обольщайтесь, то, что мы рассмотрели, — это простейший пример. В реальной жизни оптимизирующие компиляторы такого понаворочают...

Наглядное представление сложных условий в виде дерева. Конструкцию, состоящую из трех-четырех элементарных условий, можно проанализировать и в уме (да и то, если есть соответствующие навыки), но хитросплетения пяти и более условий образуют самый настоящий лабиринт — его с лету не возьмешь. Неоднозначность трансляции сложных условий порождает неоднозначность интерпретации, что приводит к многовариантному анализу, причем с каждым шагом в голове приходится держать все больше и больше информации. Так недолго и крышей поехать или окончательно запутаться и получить неверный результат.

Выход — в использовании двухуровневой системы ретрансляции. На первом этапе элементарные условия преобразуются к некоторой промежуточной форме записи, наглядно и непротиворечиво отображающей взаимосвязь элементарных операций. Затем осуществляется окончательная трансляция в любую подходящую нотацию (например, Си, Бейсик или Pascal).

Единственная проблема — выбрать удачную промежуточную форму. Существует множество решений, но в книге по соображениям экономии бумажного пространства мы рассмотрим только одно — *деревья*.

Изобразим каждое элементарное условие в виде узла с двумя ветвями, соответствующими состояниям: *условие истинно* и *условие ложно*. Для наглядности обозначим «ложь» равнобедренным треугольником, а «истину» — квадратом и условимся всегда располагать ложь на левой, а истину на правой ветке. Получившуюся конструкцию назовем «гнездом» (*nest*).



Рис. 22. Схематическое представление гнезда (*nest*)

Гнезда могут объединяться в деревья, соединяясь узлами с ветками другого узла. Причем каждый узел может соединяться только с одним гнездом, но всякое гнездо может соединяться с несколькими узлами. Непонятно? Не волнуйтесь, сейчас со всем этим мы самым внимательным образом разберемся.

Рассмотрим объединение двух элементарных условий логической операцией AND на примере выражения $((a==b) \ \&\& \ (a!=0))$. Извлекаем первое слева условие $(a==b)$, «усаживаем» его в гнездо с двумя ветвями: левая соответствует случаю, когда $a!=b$ (т. е. условие $a==b$ — ложно), а правая соответственно наоборот. Затем то же самое делаем и со вторым условием $(a!=0)$. У нас получаются два очень симпатичных гнездышка, остается лишь связать их меж собой операцией логического AND. Как известно, AND выполняет второе условие только в том случае, если истинно первое. Значит, гнездо $(a!=0)$ следует прицепить к правой ветке гнезда $(a==b)$. Тогда правая ветка гнезда $(a!=0)$ будет соответствовать истинности выражения $((a==b) \ \&\& \ (a!=0))$, а обе левые ветки — его ложности. Обозначим первую ситуацию меткой *do_it*, а вторую — *continue*. В результате дерево должно принять вид, изображенный на рис. 23.

Для наглядности отметим маршрут из вершины дерева к метке *do_it* жирной красной стрелкой. Как видите, в пункт *do_it* можно попасть только одним путем. Вот так графически выглядит операция AND.

Обратите внимание — в пункт *do_it* можно попасть только одним путем!

Перейдем теперь к операции логического OR. Рассмотрим конструкцию $((a==b) \ || \ (a!=0))$. Если условие $(a==b)$ истинно, то и все выражение считается истинным. Следовательно, правая ветка гнезда $(a==b)$ связана с меткой *do_it*.

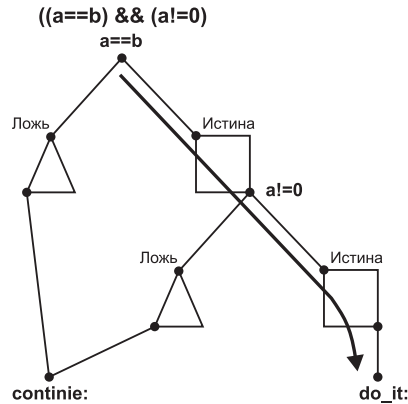


Рис. 23. Графическое представление операции AND в виде двоичного дерева

Если же условие $(a==b)$ ложно, то выполняется проверка следующего условия. Значит, левая ветка гнезда $(a==b)$ связана с гнездом $(a!=b)$. Очевидно, если условие $(a!=b)$ истинно, то истинно и все выражение $((a==b) \vee (a!=0))$, напротив, если условие $(a!=b)$ ложно, то ложно и все выражение, так как проверка условия $(a!=b)$ выполняется только в том случае, если условие $(a==b)$ ложно. Отсюда мы заключаем, что левая ветка гнезда $(a!=b)$ связана с меткой `continue:`, а правая — с `do_it:` (рис. 24). Обратите внимание, в пункт `do_it` можно попасть двумя различными путями! Вот так графически выглядит операция OR.

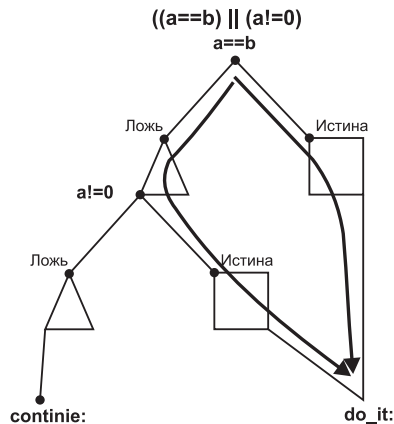


Рис. 24. Графическое представление операции OR в виде двоичного дерева

До сих пор мы отображали логические операции на деревья, но ведь деревья создавались как раз для противоположной цели — преобразованию последовательности элементарных условий к интуитивно понятному представлению. Займемся этим? Пусть в тексте программы встретится следующий код:

Листинг 150

```

IF a==b THEN check_null
IF a!=c THEN continue
check_null:
IF a==0 THEN continue
...// код условия
continue:
...// прочий код

```

Извлекаем условие ($a==b$) и сажаем его в «гнездо», смотрим: если оно ложно, то выполняется проверка ($a!=c$), значит, гнездо ($a!=c$) связано с левой веткой гнезда ($a==b$). Если же условие ($a==b$) истинно, то управление передается метке `check_null`, проверяющей истинность условия ($a==0$), следовательно, гнездо ($a==0$) связано с правой веткой гнезда ($a==b$). В свою очередь, если условие ($a!=c$) истинно, управление получает метка `continue`, в противном случае — `check_null`. Значит, гнездо ($a!=0$) связано одновременно и с правой веткой гнезда ($a==b$), и с левой веткой гнезда ($a!=c$).

Конечно, это проще рисовать, чем описывать! Если вы все правильно зарисовали, у вас должно получиться дерево, очень похожее на изображенное на рис. 25.

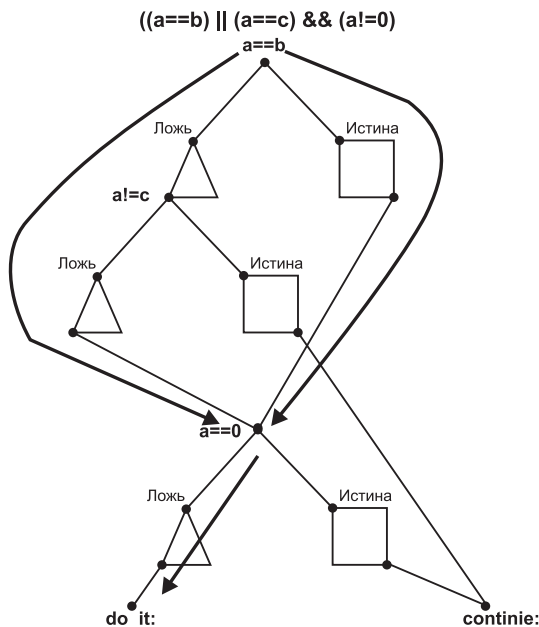


Рис. 25. Графическое представление сложного выражения

Смотрите: к гнезду ($a==0$) можно попасть двумя путями — либо через гнездо ($a==b$), либо через цепочку двух гнезд ($a==b$) \rightarrow ($a!=c$). Следовательно, эти гнезда связаны операцией OR. Записываем: `if ((a==b) || !(a!=c)....)` Откуда взялся NOT? Так ведь гнездо ($a==0$) связано с левой веткой гнезда ($a!=c$), т. е. проверя-

ется ложность его истинности! (Кстати, «ложность истинности» очень хорошо звучит). Избавляемся от NOT, инвертируя условие: if ((a==b) || (a==c)...)... Далее, из гнезда (a==0) до пункта do_it можно добраться только одним путем, значит, оно связано операцией AND. Записываем: if (((a==b) || (a==c)) && !(a==0))... Теперь избавляемся от лишних скобок и операции NOT. В результате получается: if ((a==b) || (a==c) && (a!=0)) { // Код условия }.

Не правда ли все просто? Причем вовсе необязательно строить деревья «вручную», при желании можно написать программу, берущую эту работу на себя.

Исследование конкретных реализаций. Прежде чем приступить к отображению конструкции IF (сложное условие) THEN оператор₁:оператор₂ ELSE оператор_а:оператор_б на машинный язык, вспомним, что, во-первых, агрегат IF-THEN-ELSE можно выразить через IF-THEN, во-вторых, THEN оператор₁:оператор₂ можно выразить через THEN GOTO do_it, в-третьих, любое сложное условие можно свести к последовательности элементарных условий отношения. Таким образом, на низком уровне мы будем иметь дело лишь с конструкциями IF (простое условие отношения) THEN GOTO do_it, а уже из них, как из кирпичиков, можно сложить что угодно.

Итак, *условия отношения*, или, другими словами, *результат операции сравнения двух чисел*. В микропроцессорах Intel 80x86 сравнение целочисленных значений осуществляется командой CMP, а вещественных — одной из следующих инструкций сопроцессора: FCOM, FCOMP, FCOMPP, FCOMI, FCOMIP, FUCOMI, FUCOMIP. Предполагается, что читатель уже знаком с языком ассемблера, поэтому не будем подробно останавливаться на этих инструкциях и рассмотрим их лишь вкратце.

CMP. Команда CMP эквивалентна операции целочисленного вычитания SUB, за одним исключением — в отличие от SUB, CMP не изменяет операндов, а воздействует лишь на флаги основного процессора: флаг **нуля**, флаг **переноса**, флаг **знака** и флаг **переполнения**.

Флаг нуля устанавливается в единицу, если результат вычитания равен нулю, т. е. операнды равны друг другу.

Флаг переноса устанавливается в единицу, если в процессе вычитания произошел заем из старшего бита уменьшаемого операнда, т. е. уменьшаемое меньше вычитаемого.

Флаг знака равен старшему — знаковому — биту результата вычислений, т. е. результат вычислений — отрицательное число.

Флаг переполнения устанавливается в единицу, если результат вычислений «залез» в старший бит, приводя к потере знака числа.

Для проверки состояния флагов существует множество команд *условных переходов*, выполняющихся в случае, если определенный флаг (набор флагов) установлен (сброшен). Инструкции, используемые для анализа результата сравнения целых чисел, перечислены в табл. 16.

В общем случае конструкция IF (элементарное условие отношения) THEN do_it транслируется в следующие команды процессора:

```
CMP    A,B
Jxx    do_it
continue:
```

Между инструкциями CMP и Jxx могут находиться и другие команды, не изменяющие флагов процессора, например MOV, LEA.

Таблица 16. Соответствие операций отношения командам процессора

Условие		Состояние флагов			Инструкция		
		Zero flag	Carry Flag	Sing Flag			
a == b		1	?	?	JZ	JE	
a != b		0	?	?	JNZ	JNE	
a < b	беззнаковое	?	1	?	JC	JB	JNAE
	знаковое	?	?	!=OF	JL	JNGE	
a > b	беззнаковое	0	0	?	JA	JNBE	
	знаковое	0	?	==OF	JG	JNLE	
a >= b	беззнаковое	?	0	?	JAE	JNB	JNC
	знаковое	?	?	==OF	JGE	JNL	
a <= b	беззнаковое	(ZF == 1) (CF == 1)		?	JBE	JNA	
	знаковое	1	?		JLE	JNG	

Сравнение вещественных чисел. Команды сравнения вещественных чисел FCOMxx (табл. 18), в отличие от команд целочисленного сравнения, воздействуют на регистры сопроцессора, а не основного процессора. На первый взгляд — логично, но весь камень преткновения в том, что инструкций условного перехода, управляемых флагами сопроцессора, не существует! К тому же флаги сопроцессора непосредственно недоступны, чтобы прочитать их статус, необходимо выгрузить регистр состояния сопроцессора **SW** в память или регистр общего назначения основного процессора.

Хуже всего — анализировать флаги «вручную»! Если при сравнении целых чисел можно и не задумываться, какими именно флагами управляется условный переход, достаточно написать, скажем, CMP A,B; JGE do_it. (*Jump [if] Great /or/ Equal* — прыжок, если A больше или равно B), то теперь этот номер не пройдет! Правда, можно схитрить и скопировать флаги сопроцессора в регистр флагов основного процессора, а затем использовать родные инструкции условного перехода из серии Jxx.

Конечно, непосредственно скопировать флаги из сопроцессора в основной процессор нельзя, и эту операцию приходится осуществлять в два этапа. Сначала флаги FPU выгрузить в память или регистр общего назначения, а уже оттуда за-
талкивать в регистр флагов CPU. Непосредственно модифицировать регистр фла-

гов CPU умеет только одна команда — POPF. Остается только выяснить, каким флагам сопроцессора какие флаги процессора соответствуют. И вот что удивительно — флаги 8, 10 и 14-й сопроцессора совпадают с 0, 2 и 6-ым флагами процессора — CF, PF и ZF соответственно (см. табл. 17). То есть, старший байт регистра флагов сопроцессора можно безо всяких преобразований затолкать в младший байт регистра флагов процессора, и это будет работать, но... при этом исказятся 1, 3 и 5-й биты флагов CPU, никак не используемые в текущих версиях процессора, но зарезервированные на будущее. Менять значение зарезервированных битов **нельзя**! Кто знает, вдруг завтра один из них будут отвечать за самоуничтожение процессора? Шутка, конечно, но в ней есть своя доля истины.

К счастью, никаких сложных манипуляций нам проделывать не придется — разработчики процессора предусмотрели специальную команду SAHF, копирующую 8, 10, 12, 14 и 1-й биты регистра AX в 0, 2, 4, 6 и 7-й биты регистра флагов CPU соответственно. Сверяясь по табл. 17, мы видим, что 7-й бит регистра флагов CPU содержит флаг знака, а соответствующий ему флаг FPU — признак занятости сопроцессора!

Отсюда следует, что для анализа результата сравнения вещественных чисел использовать знаковые условные переходы (JL, JG, JLE, JNL, JNLE, JGE, JNGE) **нельзя**! Они работают с флагами знака и переполнения, естественно, если вместо флага знака им подсовывают флаг занятости сопроцессора, а флаг переполнения оставляют в подвешенном состоянии, условный переход будет срабатывать не так, как вам бы этого хотелось! Применяйте лишь беззнаковые инструкции перехода — JE, JB, JA и др. (см. табл. 16).

Разумеется, это не означает, что сравнивать знаковые вещественные значения нельзя, можно, еще как! Но для анализа результатов сравнения обязательно всегда использовать только беззнаковые условные переходы!

Таблица 17. Соответствие флагов CPU и FPU

CPU	7	6	5	4	3	2	1	0
	SF	ZF	—	AF	—	PC	—	CF
FPU	15	14	13	12	11	10	9	8
	Busy!	C3 (ZF)	TOP			C2 (PF)	C1	C0 (CF)

Таким образом, вещественная конструкция IF (элементарное условие отношения) THEN do_it транслируется в одну из двух следующих последовательностей инструкций процессора:

Листинг 151

```
fld          [a]          fld          [a]
fcomp        [b]          fcomp        [b]
fnstsw ax    fnstsw ax
sahf                     test          ah, bit_mask
jxx          do_it        jnz do_it
```

Первый вариант более нагляден, зато второй работает быстрее. Однако такой код (из всех известных мне компиляторов) умеет генерировать один лишь Microsoft Visual C++. Borland C++ и хваленый WATCOM C испытывают неопределимую тягу к инструкции SAHF, чем вызывают небольшие тормоза, но чрезвычайно упрощают анализ кода, ибо, встретив команду наподобие JNA, мы и спросонок скажем, что переход выполняется, когда $a \leq b$, а вот проверка битовой маски TEST AH, 0x41/JNZ do_it заставит нас крепко задуматься или машинально потянуться к справочнику за разъяснениями (см. табл. 16).

Команды семейства FUCOMIxx в этом смысле гораздо удобнее в обращении, так как возвращают результат сравнения непосредственно в регистры основного процессора, но, увы, их понимает только Pentium Pro, а в более ранних микропроцессорах они отсутствуют. Поэтому вряд ли читателю доведется встретиться с ними в реальных программах, так что не имеет никакого смысла останавливаться на этом вопросе. Во всяком случае, всегда можно обратиться к с. 3—112 руководства «*Instruction Set Reference*», где эти команды подробно описаны.

Таблица 18. Команды сравнения вещественных значений

Инструкция	Назначение	Результат
FCOM	Сравнивает вещественное значение, находящееся на вершине стека сопроцессора, с операндом, находящимся в памяти или стеке FPU	Флаги FPU
FCOMP	То же самое, что и FCOM, но с выталкиванием вещественного значения с вершины стека	
FCOMPP	Сравнивает два вещественных значения, лежащих на вершине стека сопроцессора, затем выталкивает их из стека	
FCOMI	Сравнивает вещественное значение, находящееся на вершине стека сопроцессора с другим вещественным значением, находящимся в стеке FPU	Флаги CPU
FCOMIP	Сравнивает вещественное значение, находящееся на вершине стека сопроцессора с другим вещественным значением, находящимся в стеке FPU, затем выталкивает верхнее значение из стека	
FUCOMI	Неупорядоченно сравнивает вещественное значение, находящееся на вершине стека сопроцессора с другим вещественным значением, находящимся в стеке FPU	
FUCOMIP	Неупорядоченно сравнивает вещественное значение, находящееся на вершине стека сопроцессора с другим вещественным значением, находящимся в стеке FPU, затем выталкивает верхнее значение из стека	

Таблица 19. Назначение и битовые маски флагов сопроцессора

Флаги FPU	Назначение		Битовая маска
OE	Флаг переполнения	Overfull Flag	#0x0008
C0	Флаг переноса	Carry Flag	#0x0100
C1	—		#0x0200
C2	Флаг четности	Parity Flag	#0x0400
C3	Флаг нуля	Zero Flag	#0x4000

Таблица 20. Состояние регистров флагов для различных операций отношения.
a — левый, b — правый операнд команды сравнения вещественных значений

Отношение	Состояние флагов FPU		SAHF	Битовая маска
a<b	C0 == 1		JB	#0x0100 == 1
a>b	C0 == 0	C3 == 0	JNBE	#0x4100 == 0
a==b	C3 == 1		JZ	#0x4000 == 1
a!=b	C3 == 0		JNZ	#0x4000 == 0
a>=b	C0 == 0		JNB	#0x0100 == 0
a<=b	C0 == 1	C3 == 1	JNA	#0x4100 == 1

Таблица 21. «Характер» некоторых компиляторов

Компилятор	Алгоритм анализа флагов FPU
Borland C++	Копирует флаги сопроцессора в регистр флагов основного процессора
Microsoft Visual C++	Тест битовой маски
WATCOM C	Копирует флаги сопроцессора в регистр флагов основного процессора
Free Pascal	Копирует флаги сопроцессора в регистр флагов основного процессора

Условные команды булевой установки. Начиная с 80386 чипа, язык микропроцессоров Intel обогатился командой условной установки байта — SETxx, устанавливающей свой единственный операнд в единицу (булево TRUE), если условие xx равно, и соответственно сбрасывающей его в ноль (булево FALSE), если условие xx ложно.

Команда SETxx широко используются оптимизирующими компиляторами для устранения ветвлений, т. е. избавления от условных переходов, так как последние очищают конвейер процессора, чем серьезно снижают производительность программы.

Подробнее об этом рассказывается в главе «Оптимизация ветвлений», здесь же мы не будем останавливаться на этом сложном вопросе (см. там же «Булевы сравнения» и «Идентификация условного оператора (*условие*)?do_it:continue»).

Таблица 22. Условные команды булевой установки

Команда			Отношение		Условие
SETA	SETNBE		a>b	беззнаковое	CF == 0 && ZF == 0
SETG	SETNLE			знаковое	ZF == 0 && SF == OF
SETAE	SETNC	SETNB	a>=b	беззнаковое	CF == 0
SETGE	SETNL			знаковое	SF == OF
SETB	SETC	SETNAE	a<b	беззнаковое	CF == 1
SETL	SETNGE			знаковое	SF != OF
SETBE	SETNA		a<=b	беззнаковое	CF == 1 ZF == 1
SETLE	SETNG			знаковое	ZF == 1 SF != OF
SETE	SETZ		a==b	—	ZF == 1
SETNE	SETNZ		a!=0	—	ZF == 0

Прочие условные команды. Микропроцессоры серии 80x86 поддерживают множество условных команд, в общем случае не отображающихся на операции отношения, а потому и редко используемые компиляторами (можно даже сказать — вообще не используемые), но зато часто встречающиеся в ассемблерных вставках. Словом, они заслуживают хотя бы беглого упоминания.

Команды условного перехода. Помимо описанных в табл. 16, существует еще восемь других условных переходов — JCXZ, JECXZ, JO, JNO, JP (он же JPE), JNP (он же JPO), JS и JNS. Из них только JCXZ и JECXZ имеют непосредственное отношение к операциям сравнения. Оптимизирующие компиляторы могут заменять конструкцию CMP [E]CX, 0\JZ do_it на более короткий эквивалент J[E]CX do_it, однако чаще всего они (в силу ограниченности интеллекта и лени своих разработчиков) этого не делают.

Условные переходы JO и JNS используются в основном в математических библиотеках для обработки чисел большой разрядности (например, 1024 битных целых).

Условные переходы JS и JNS помимо основного своего предназначения часто используются для быстрой проверки значения старшего бита.

Условные переходы JP и JNP вообще практически не используются, ну разве что в экзотичных ассемблерных вставках.

Таблица 23. Вспомогательные условные переходы

Команда		Переход, если:	Флаги
JCXZ		Регистр CX равен нулю	CX == 0
JECXZ		Регистр ECX равен нулю	ECX == 0
JO		Переполнение	OF == 1
JNO		Нет переполнения	OF == 0
JP	JPE	Число бит младшего байта результата четно	PF == 1
JNP	JPO	Число бит младшего байта результата нечетно	PF == 0
JS		Знаковый бит установлен	SF == 1
JNS		Знаковый бит сброшен	SF == 0

Команды условной пересылки. Старшие процессоры семейства Pentium (Pentium Pro, Pentium II, CELERON) поддерживают команду условной пересылки CMOVxx, пересылающей значение из источника в приемник, если условие xx истинно. Это позволяет писать намного более эффективный код, не содержащий ветвлений и укладывающийся в меньшее число инструкций.

Рассмотрим конструкцию IF a<b THEN a=b. Сравните, как она транслируется с использованием условных переходов (1) и команды условной пересылки (2):

Листинг 152

```
CMP A, B          CMP A, B
JAE continue:     CMOVB A, B
MOV A, B
continue:
1)                2)
```

К сожалению, ни один из известных автору компиляторов на момент написания этих строк никогда не использовал CMOVxx при генерации кода, однако выигрыш от нее настолько очевиден, что появления усовершенствованных оптимизирующих компиляторов следует ожидать в самом ближайшем будущем. Вот почему эта команда включена в настоящий обзор. В табл. 24 дано ее краткое, но вполне достаточное для дизассемблирования программ описание. За более подробными разъяснениями обращайтесь к с. 3 — 59 справочного руководства «*Instruction Set Reference*» от Intel.

Таблица 24. Основные команды условной пересылки

Команда			Отношение	Условие
CMOVA	CMOVNBE		a>b	беззнаковое CF == 0 && ZF == 0
CMOVG	CMOVNLE			знаковое ZF == 0 && SF == OF
CMOVAE	CMOVNC	CMOVNB	a>=b	беззнаковое CF == 0
CMOVGE	CMOVNL			знаковое SF == OF

Команда			Отношение		Условие
CMOVB	CMOVC	CMOVNAE	a<b	беззнаковое	CF == 1
CMOVL	CMOVNGE			знаковое	SF != OF
CMOVBE	CMOVNA		a<=b	беззнаковое	CF == 1 ZF == 1
CMOVL	CMOVNG			знаковое	ZF == 1 SF != OF
CMOVE	CMOVZ		a==b	—	ZF == 1
CMOVNE	CMOVNZ		a!=0	—	ZF == 0

Булевы сравнения. Логической лжи (FALSE) соответствует значение ноль, а логической истине (TRUE) — любое ненулевое значение. Таким образом, булевы отношения сводятся к операции сравнения значения переменной с нулем. Конструкция IF (a) THEN do_it транслируется в IF (a!=0) THEN do_it.

Практически все компиляторы заменяют инструкцию CMP A, 0 более короткой командой TEST A,A или OR A,A. Во всех случаях, если A==0, устанавливается флаг нуля, и соответственно наоборот.

Поэтому, встретив в дизассемблеровом тексте конструкцию a la TEST EAX, EAX\ JZ do_it, можно с уверенностью утверждать, что мы имеем дело с булевым сравнением.

Идентификация условного оператора «(условие)?do_it:continue». Конструкция a=(условие)?do_it:continue языка Си в общем случае транслируется так: IF (условие) THEN a=do_it ELSE a=continue, однако результат компиляции обеих конструкций, вопреки распространенному мнению, не всегда идентичен.

В силу ряда обстоятельств оператор «?» значительно легче поддается оптимизации, чем ветвление IF-THEN-ELSE.

Покажем это на следующем примере:

Листинг 153

```
main()
{
    int a;           // Переменная специально не инициализирована,
    int b;           // чтобы компилятор не заменил ее константой.

    a=(a>0)?1:-1;    // Условный оператор.

    if (b>0)         // Ветвление.
        b=1;
    else
        b=-1;

    return a+b;
}
```

Если пропустить эту программу сквозь компилятор Microsoft Visual C++, на выходе мы получим такой код:

Листинг 154

```

push    ebp
mov     ebp, esp
; Открываем кадр стека

sub     esp, 8
; Резервируем место для локальных переменных.

; // Условный оператор "?"
; Начало условного оператора "?"
xor     eax, eax
; Обнуляем EAX

cmp     [ebp+var_a], 0
; Сравниваем переменную a с нулем

setle   al
; Поместить в al значение 0x1, если var_a <= 0.
; Соответственно поместить в al значение 0, если var_a>0

dec     eax
; Уменьшить EAX на единицу.
; Теперь, если var_a > 0, то EAX := -1,
; если var_a <=0, то EAX := 0

and     eax, 2
; Сбросить все биты, кроме второго слева, считая от одного.
; Теперь, если var_a > 0, то EAX := 2,
; если var_a <=0, то EAX := 0

add     eax, 0FFFFFFFh
; Отнять от EAX 0x1
; Теперь, если var_a > 0, то EAX := 1,
; если var_a <=0, то EAX := -1
mov     [ebp+var_a], eax
; Записать результат в переменную var_a.
; Конец оператора "?"
; Обратите внимание, для трансляции условного оператора не потребовалось ни
; одного условного перехода, компилятор сумел обойтись без ветвлений!

; // Ветвление
; Начало ветвления IF - THEN - ELSE
cmp     [ebp+var_b], 0
; Сравнение переменной var_b с нулем

jle     short else
; Переход, если var_b <= 0.

; Ветка "var_b > 0"
mov     [ebp+var_b], 1
; Записываем в переменную var_b значение 1

jmp     short continue
; Переход к метке continue.

; Ветка "var_b > 0"

```

```

mov     [ebp+var_b], 0FFFFFFFh
; Записываем в переменную var_b значение -1

continue:                                ; CODE XREF: _main+26j
; Конец ветвления IF-THEN-ELSE.
; Обратите внимание - представление ветвления IF-THEN-ELSE намного компактнее
; условного оператора "?", однако содержит в себе условные переходы, ощутимо
; снижающие быстродействие программы

mov     eax, [ebp+var_a]
; Загружаем в EAX значение переменной var_a

add     eax, [ebp+var_b]
; Складываем значение переменной var_a со значением переменной var_b
; и помещаем результат в EAX

mov     esp, ebp
pop     ebp
; Закрываем кадр стека
ret

```

Таким образом, мы видим, что нельзя априори утверждать, будто бы результат трансляции условного оператора «?» всегда эквивалентен результату трансляции конструкции IF-THEN-ELSE. Однако тот же Microsoft Visual C++ в режиме агрессивной оптимизации в обоих случаях генерирует идентичный код. Смотрите:

Листинг 155

```

_main      proc near
push       ecx
; Резервируем место для локальных переменных a и b.
; Поскольку они никогда не используются вместе, а только поочередно,
; компилятор помещает их в одну ячейку памяти

mov        edx, [esp+0]; команда № 1 оператора "?"
; Загрузка в EDX значения переменной a

xor        eax, eax; команда № 2 оператора ?
; Обнуляем EAX
; Поскольку команда setle al изменяет содержимое одного лишь al и не трогает
; остальную часть регистра, нам приходится очищать его самостоятельно

test       edx, edx; команда № 3 оператора "?"
; Проверка переменной a на равенство нулю

mov        edx, [esp+0]; команда № 1 ветвления IF
; Загрузка в EDX значения переменной b

setle      al; команда № 4 оператора "?"
; Поместить в al значение 0x1, если a <= 0.
; Соответственно поместить в al значение 0, если a > 0

dec        eax; команда № 5 оператора "?"
; Уменьшить EAX на единицу
; Теперь,      если a > 0, то EAX := -1,
;              если a <= 0, то EAX := 0

```

```

xor     ecx, ecx; команда № 2 ветвления IF
; Обнулить ECX

and     eax, 2; команда № 6 оператора "?"
; Сбросить все биты, кроме второго слева, считая от одного.
; Теперь,      если a > 0, то EAX := 2,
;              если a <=0, то EAX := 0

dec     eax; команда № 7 оператора "?"
; Уменьшить EAX на единицу
; Теперь,      если a > 0, то EAX := 1,
;              если a <=0, то EAX := -1

test    edx, edx; команда № 3 ветвления IF
; Проверка переменной b на равенство нулю

setle   cl; команда № 4 ветвления IF
; Поместить в cl значение 0x1, если b <= 0.
; Соответственно поместить в cl значение 0, если b>0

dec     ecx; команда № 5 ветвления IF
; Уменьшить ECX на единицу.
; Теперь,      если b > 0, то ECX := -1,
;              если b <=0, то ECX := 0

and     ecx, 2; команда № 6 ветвления IF
; Сбросить все биты, кроме второго слева, считая от первого
; Теперь,      если b > 0, то ECX := 2
;              если b <=0, то ECX := 0

dec     ecx; команда № 7 ветвления IF
; Уменьшить ECX на единицу.
; Теперь,      если b > 0, то ECX := -1,
;              если b <=0, то ECX := 0

add     eax, ecx
; Сложить переменную a с переменной b

pop     ecx
; Закрыть кадр стека

retn

_main   endp

```

Компилятор некоторым образом перемешал команды, относящиеся к условному оператору «?», с командами ветвления IF-THEN-ELSE (это было сделано для лучшего спаривания инструкций), однако если их сравнить, то выяснится — реализации обеих конструкций абсолютно идентичны друг другу!

Однако с точки зрения языка условный оператор «?» выгодно отличается от ветвления тем, что может непосредственно использоваться в выражениях, например:

Листинг 156

```

main()
{
    int a;
    printf("Hello, %s\n", (a>0)?"Sailor":"World!");
}

```

Попробуйте также компактно реализовать это с помощью ветвлений! Но на самом деле это удобство лишь внешнее, а компилятор транслирует приведенный пример так:

Листинг 157

```
main()
{
    int a;
    char *p;
    static char s0[]="Sailor";
    static char s1[]="World";
    if (a>0) p=s0; else p=s1;

    printf("Hello, %s\n", p);
}
```

Откомпилируйте оба листинга и дизассемблируйте полученные файлы, они должны быть идентичны. Таким образом, при декомпиляции Си/Си++ программ в общем случае невозможно сказать, использовалось ли в них ветвление или условный оператор, однако все же есть некоторые зацепки, помогающие восстановить истинный вид исходного текста в некоторых частных случаях.

Например, маловероятно, чтобы программист строил свой листинг, как показано в последнем примере. Зачем вводить статические переменные и сложным образом манипулировать с указателем, когда проще использовать условный оператор вместо ветвления?

Таким образом, если условный оператор гладко ложится в декомпилируемую программу, а ветвление не лезет в нее никаким боком, то, очевидно, что в исходном тексте использовался именно условный оператор, а не ветвление.

Идентификация типов. Условные команды — ключ к идентификации типов. Поскольку анализ результата сравнения знаковых и беззнаковых переменных осуществляется различными группами инструкций, можно уверенно и однозначно отличить `signed int` от `unsigned int`.

Впрочем, идентификация типов — тема отдельного разговора, поэтому не будет отклоняться в сторону, а рассмотрим ее чуточку позже в одноименной главе.



16-разрядный режим. Одна из неприятных особенностей 16-разрядного режима — ограниченная «дальнобойность» команд условного перехода. Разработчики микропроцессора в стремлении добиться высокой компактности кода отвели на целевой адрес всего один байт, ограничив тем самым длину прыжка интервалом в 255 байтов. Это так называемый *короткий (short)* переход, адресуемый относительным знаковым смещением, отсчитываемым от начала следующей за инструкцией перехода командой (рис. 26). Такая схема адресации ограничивает длину прыжка «вперед» (т. е. «вниз») всего 128 байтами, а «назад» (т. е. «вверх») и того меньше — 127! (Прыжок вперед короче

потому, что ему требуется «пересечь» и саму команду перехода.) Этих ограничений лишен *ближний (near)* безусловный переход, адресуемый двумя байтами и действующий в пределах всего сегмента.

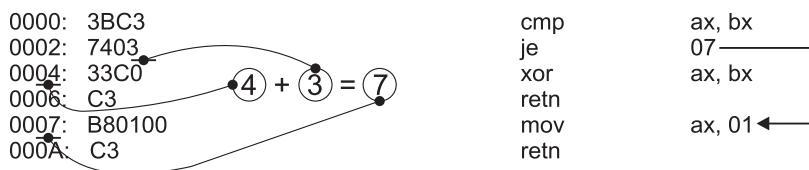


Рис. 26. Внутреннее представление короткого (short) перехода

Короткие переходы усложняют трансляцию ветвлений — ведь не всякий целевой адрес находится в пределах 128 байтов! Существует множество способов обхода этого ограничения. Наиболее популярен следующий прием: если транслятор видит, что целевой адрес выходит за пределы досягаемости условного перехода, он инвертирует условие срабатывания и совершает короткий (short) переход на метку *continue*, а на *do_it* передает управление ближним (near) переходом, действующим в пределах одного сегмента (рис. 27).

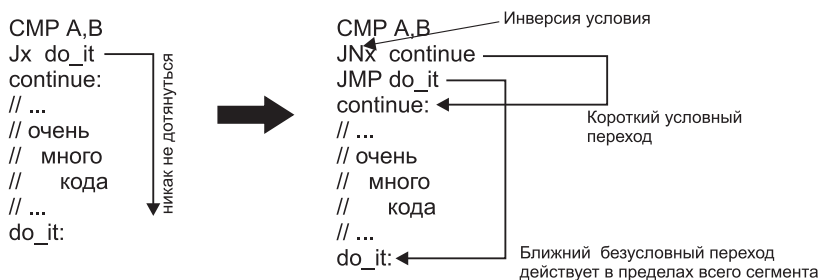


Рис. 27. Трансляция коротких переходов

Аналогичным образом можно выкрутиться и в тех ситуациях, когда целевой адрес расположен совсем в другом сегменте, достаточно лишь заменить ближний безусловный переход на дальний. Вот, собственно, и все.

К великому счастью разработчиков компиляторов и не меньшей радости хакеров, дизассемблирующих программы, в 32-разрядном режиме условный переход «бьет» в пределах всего четырехгигабайтового адресного пространства, и все эти проблемы исчезают, как бородавки после сеанса Кашпировского.

Листинги примеров. А теперь для лучшего уяснения материала, рассмотренного в этой главе, давайте рассмотрим несколько живых примеров, откомпилированных различными компиляторами. Начнем с исследования элементарных целочисленных отношений:

Листинг 158

```
#include <stdio.h>

main()
{
    int a; int b;
    if (a<b) printf("a<b");
    if (a>b) printf("a>b");
    if (a==b) printf("a==b");
    if (a!=b) printf("a!=b");
    if (a>=b) printf("a>=b");
    if (a<=b) printf("a<=b");
}
```

Результат компиляции этого примера компилятором Microsoft Visual C+ должен выглядеть так:

Листинг 159

```
main          proc near          ; CODE XREF: start+AFp
var_b         = dword ptr -8
var_a         = dword ptr -4

    push      ebp
    mov       ebp, esp
    ; Открываем кадр стека

    sub       esp, 8
    ; Резервируем память для локальных переменных var_a и var_b

    mov       eax, [ebp+var_a]
    ; Загружаем в EAX значение переменной var_a

    cmp       eax, [ebp+var_b]
    ; Сравниваем значение переменной var_a со значением переменной var_b

    jge       short loc_40101B
    ; Если var_a >= var_b, то переход на continue иначе - печать строки.
    ; Обратите внимание, что оригинальный код выглядел так:
    ; if (a<b) printf("a<b");
    ; То есть условие отношения было инвертировано компилятором!
    ; Знаковая операция JGE говорит о том, что и сравниваемые переменные
    ; var_a и var_b - также знаковые.

    ; // ВЕТКА DO_IT
    push      offset aAB_4 ; "a<b"
    call      _printf
    add       esp, 4
    ; Печать строки "a<b".

    ; // ВЕТКА CONTINUE
loc_40101B:    ; CODE XREF: main+Cj
    mov       ecx, [ebp+var_a]
    ; Загружаем в ECX значение переменной var_a
```

```
    cmp     ecx, [ebp+var_b]
; Сравниваем значение переменной var_a с переменной var_b

    jle     short loc_401030
; Переход, если var_a <= var_b, иначе печать строки
; Следовательно, строка печатается, когда !(var_a <= var_b) или
; var_a > var_b. Тогда исходный код программы должен выглядеть так:
; if (a>b) printf("a>b");

    push    offset aAB_3 ; "a>b"
    call    _printf
    add     esp, 4
;

loc_401030:                                ; CODE XREF: main+21j
    mov     edx, [ebp+var_a]
; Загружаем в EDX значение переменной var_a

    cmp     edx, [ebp+var_b]
; Сравниваем значение переменной var_a с переменной var_b

    jnz     short loc_401045
; Переход, если var_a!=var_b, иначе печать строки.
; Следовательно, оригинальный код программы выглядел так:
; if (a==b) printf("a==b");

    push    offset aAB ; "a==b"
    call    _printf
    add     esp, 4

loc_401045:                                ; CODE XREF: main+36j
    mov     eax, [ebp+var_a]
; Загружаем в EAX значение переменной var_a

    cmp     eax, [ebp+var_b]
; Сравниваем значение переменной var_a со значением переменной var_b

    jz      short loc_40105A
; Переход, если var_a==var_b, иначе печать строки.
; Следовательно, оригинальный код программы выглядел так:
; if (a!=b) printf("a!=b");

    push    offset aAB_0 ; "a!=b"
    call    _printf
    add     esp, 4

loc_40105A:                                ; CODE XREF: main+4Bj
    mov     ecx, [ebp+var_a]
; Загружаем в ECX значение переменной var_a

    cmp     ecx, [ebp+var_b]
; Сравниваем значение переменной var_a с переменной var_b

    jl      short loc_40106F
; Переход, если var_a < var_b, иначе печать строки.
; Следовательно, оригинальный код программы выглядел так:
; if (a>=b) printf("a>=b");
```

```

    push    offset aAB_1 ; "a>=b"
    call    _printf
    add     esp, 4

loc_40106F:                                ; CODE XREF: main+60j
    mov     edx, [ebp+var_a]
    ; Загружаем в EDX значение переменной var_a

    cmp     edx, [ebp+var_b]
    ; Сравниваем значение переменной var_a с переменной var_b

    jg      short loc_401084
    ; Переход, если var_a>var_b, иначе печать строки
    ; Следовательно, оригинальный код программы выглядел так:
    ; if (a<=b) printf("a<=b");

    push    offset aAB_2 ; "a<=b"
    call    _printf
    add     esp, 4

loc_401084:                                ; CODE XREF: main+75j
    mov     esp, ebp
    pop     ebp
    ; Закрываем кадр стека

    retn

main     endp

```

А теперь сравним этот 32-разрядный код с 16-разрядным кодом, сгенерированным компилятором Microsoft C++ 7.0 (ниже для экономии места приведен лишь фрагмент):

Листинг 160

```

    mov     ax, [bp+var_a]
    ; Загрузить в AX значение переменной var_a

    cmp     [bp+var_b], ax
    ; Сравнить значение переменной var_a со значением переменной var_b

    jl      loc_10046
    ; Переход на код печати строки, если var_a < var_b

    jmp     loc_10050
    ; Безусловный переход на continue.
    ; Смотрите! Компилятор, не будучи уверен, что "дальнобойности" короткого
    ; условного перехода хватит для достижения метки continue, вместо этого
    ; прыгнул на метку do_it, расположенную неподалеку - в гарантированной
    ; досягаемости, а передачу управления на continue взял на себя
    ; безусловный переход.
    ; Таким образом, инверсия истинности условия сравнения имела место дважды,
    ; первый раз при трансляции условия отношения, второй раз - при генерации
    ; машинного кода. А NOT на NOT можно сократить!
    ; Следовательно, оригинальный код выглядел так:
    ; if (a<b) printf("a<b");

```

```

loc_10046:                                ; CODE XREF: _main+11j
        mov     ax, offset aAB ; "a<b"
        push    ax
        call    _printf
        add     sp, 2

loc_10050:                                ; CODE XREF: _main+13j
        ; // прочий код

```

А теперь заменим тип сравниваемых переменных с `int` на `float` и посмотрим, как это повлияет на сгенерированный код. Результат компиляции Microsoft Visual C++ должен выглядеть так (ниже приведен лишь фрагмент):

Листинг 161

```

        fld     [ebp+var_a]
        ; Загрузка значения вещественной переменной var_a на вершину стека сопроцессора

        fcomp   [ebp+var_b]
        ; Сравнение значения переменной var_a с переменной var_b
        ; с сохранением результата сравнения во флагах сопроцессора

        fnstsw  ax
        ; Скопировать регистр флагов сопроцессора в регистр AX

        test    ah, 1
        ; Нулевой бит регистра AH установлен?
        ; Соответственно восьмой бит регистра флагов сопроцессора установлен?
        ; А что у нас хранится в восьмом бите?
        ; Ага, восьмой бит содержит флаг переноса.

        jz      short loc_20
        ; Переход, если флаг переноса сброшен, т. е. это равносильно конструкции jnc
        ; при сравнении целочисленных значений. Смотрим по табл. 16 – синоним jnc.
        ; Команда jnb.
        ; Следовательно, оригинальный код выглядел так:
        ; if (a<b) printf("a<b");

        push    offset $SG339 ; "a<b"
        call    _printf
        add     esp, 4

loc_20:                                ; CODE XREF: _main+11j

```

Гораздо нагляднее код, сгенерированный компилятором Borland C++ или WATCOM C. Смотрите:

Листинг 162

```

        fld     [ebp+var_a]
        ; Загрузка значения вещественной переменной var_a
        ; на вершину стека сопроцессора

        fcomp   [ebp+var_b]
        ; Сравнение значения переменной var_a с переменной var_b
        ; с сохранением результата сравнения во флагах сопроцессора

```

```

fnstsw ax
; Скопировать регистр флагов сопроцессора в регистр AX

sahf
; Скопировать соответствующие биты регистра AH во флаги основного процессора

jnb     short loc_1003C
; Переход, если !(a<b), иначе печать строки printf("a<b").
; Теперь, не копаясь ни в каких справочных таблицах, можно восстановить
; оригинальный код: if (a<b) printf("a<b");

push    offset unk_100B0 ; format
call    _printf
pop     ecx

loc_1003C:                                ; CODE XREF: _main+Fj

```

Теперь, «насобачившись» на идентификации элементарных условий, перейдем к вещам по-настоящему сложным.

Рассмотрим следующий пример:

Листинг 163

```

#include <stdio.h>

main()
{
    unsigned int a; unsigned int b; int c; int d;
    if (d) printf("TRUE"); else if (((a>b) && (a!=0)) || ((a==c) && (c!=0)))
        printf("OK\n");
    if (c==d) printf("+++\n");
}

```

Результат его компиляции должен выглядеть приблизительно так:

Листинг 164

```

_main          proc near
var_d          = dword ptr -10h
var_C          = dword ptr -0Ch
var_b          = dword ptr -8
var_a          = dword ptr -4

    push    ebp
    mov     ebp, esp
    ; Открытие кадра стека

    sub     esp, 10h
    ; Резервирование места для локальных переменных

    cmp     [ebp+var_d], 0
    ; Сравнение значения переменной var_d с нулем

    jz      short loc_1B
    ; Если переменная var_d равна нулю, переход к метке loc_1B, иначе
    ; печать строки TRUE. Схематически это можно изобразить так:
    ;

```

```

1      var_d == 0
2
3      loc_1B
4
5      printf("TRUE");
6
7
8

```

```
push    offset $SG341 ; "TRUE"
call    _printf
add     esp, 4
jmp     short loc_44
; Ага, говорим мы голосом Пятачка, искушающего Кенгу!
; Вносим этот условный переход в наше дерево
```

```
graph TD; A["var_d == 0"] --> B["loc_1B"]; A --> C["printf(\"TRUE\");"]; C --> D["loc_44"];
```

```
loc_1B:                                ; CODE XREF: _main+Aj
```

```
mov     eax, [ebp+var_a]
```

; Загружаем в EAX значение переменной var_a

```
cmp      eax, [ebp+var_b]
; Сравниваем переменную var_a с переменной var_b
```

```
jbe     short loc_29
; Если var_a меньше или равна переменной var_b, то переход на loc_29.
; Прививаем новое гнездо к нашему дереву, попутно обращая внимание не то, что
; var_a и var_b - беззнаковые переменные!
```

```

graph TD
    var_d == 0 --> loc_1B
    var_d == 0 --> printf_TRUE_printfTRUE["printf(\\\"TRUE\\\");"]
    loc_1B --> var_a <= var_b
    var_a <= var_b --> continue
    var_a <= var_b --> loc_29
    printf_TRUE_printfTRUE --> loc_44

```

```
cmp     [ebp+var_a], 0
; Сравниваем значение переменной var_a с нулем
```

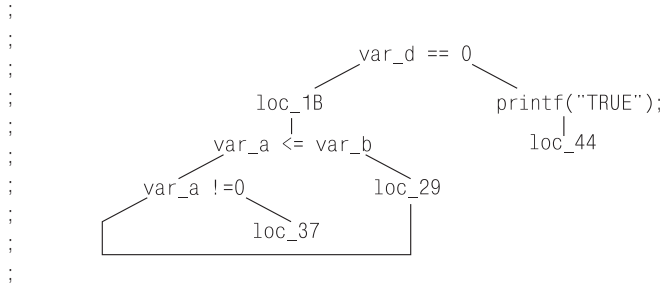
```
jnz      short loc_37
; Переход на loc_37, если var_a не равна нулю
```

```

graph TD
    var_d == 0 --> loc_1B
    var_d == 0 --> printf_TRUE[printf("TRUE");]
    loc_1B --> var_a <= var_b
    var_a <= var_b --> loc_29
    var_a <= var_b --> var_a != 0
    var_a != 0 --> continue
    var_a != 0 --> loc_37
    loc_29 --> loc_44
    loc_44 --> loc_44

```

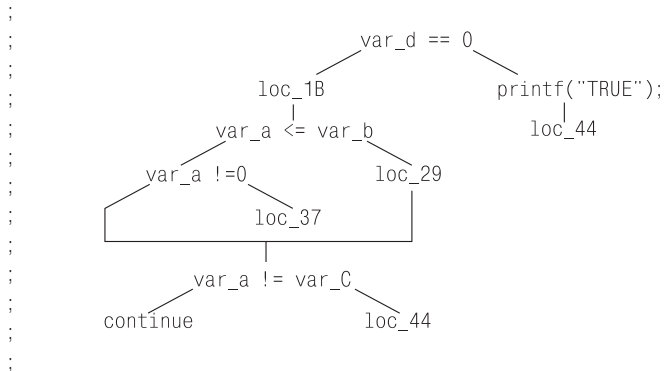
```
loc_29:                                ; CODE XREF: _main+21j
; Смотрите, в нашем дереве уже есть метка loc_29! Корректируем его!
```



```
mov     ecx, [ebp+var_a]
; Загружаем в ECX значение переменной var_a

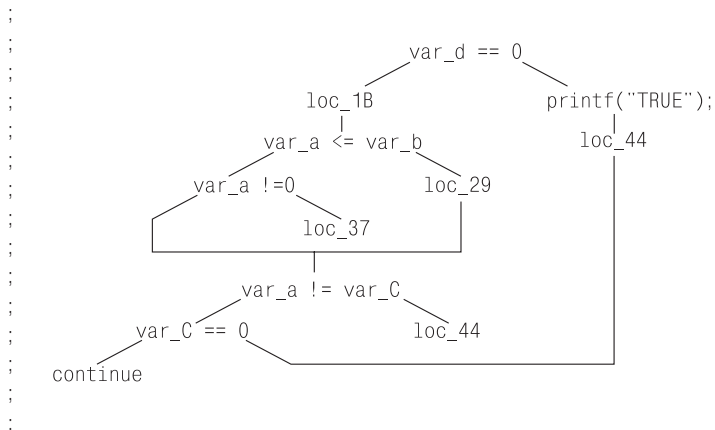
cmp     ecx, [ebp+var_C]
; Сравниваем значение переменной var_a с переменной var_C

jnz     short loc_44
; переход, если var_a != var_C
```

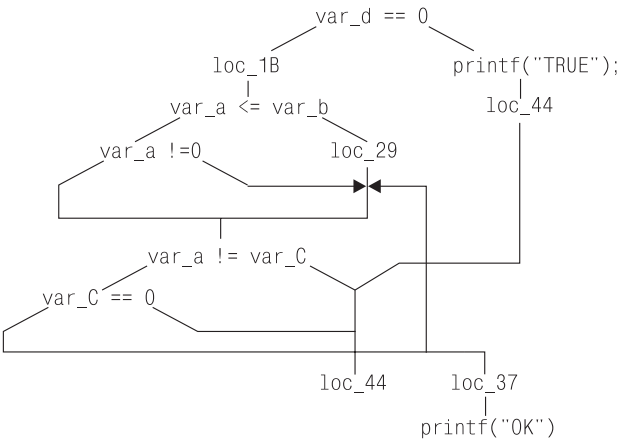


```
cmp     [ebp+var_C], 0
; Сравнение значения переменной var_C с нулем

jz      short loc_44
; Переход на loc_44, если var_C == 0
```

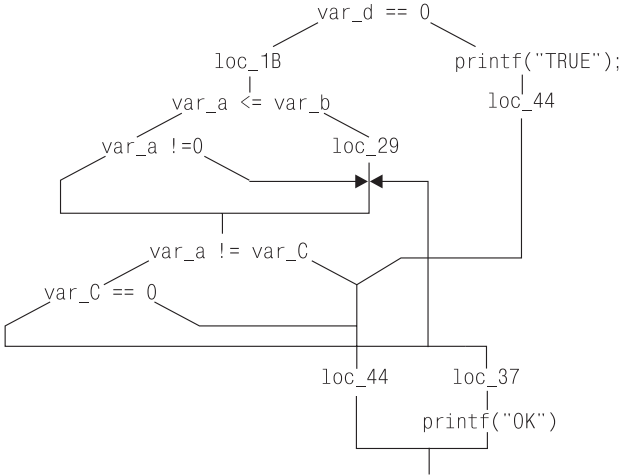



```
loc_37:                                ; CODE XREF: _main+27j
; Смотрим, метка loc_37 уже есть в дереве! Прививаем!
```



```
push    offset $SG346 ; "OK\n"
call    _printf
add     esp, 4
```

```
loc_44:                                ; CODE XREF: _main+19j _main+2Fj ...
; Смотрите, ветки loc_44 и loc_37 смыкаются!
```



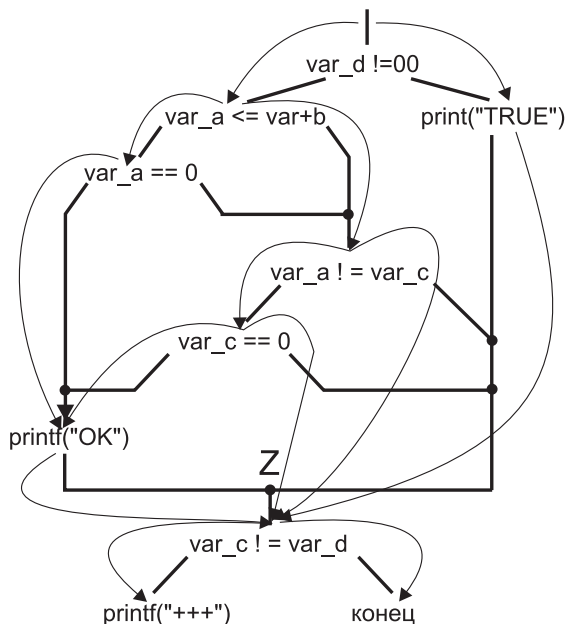


Рис. 28. Логическое дерево

От гнезда `var_d != 0` отходят две ветки — правая ведет к `printf («OK»)` и далее к завершению конструкции IF-THEN [ELSE], а левая, прежде чем выйти к точке Z, минует целое полчище гнезд. В переводе на русский язык ситуация выглядит так: *если переменная var_d не равна нулю, то печатаем «OK» и сваливаем, иначе выполняем дополнительные проверки*. Проще говоря: IF (`var_d != 0`) THEN `printf("OK")` ELSE ... Короче говоря, левая ветка гнезда (`var_d != 0`) есть ветка ELSE. Изучим ее?

От гнезда (`var_a <= var_b`) к узлу `printf («OK»)` ведут два пути: `!(var_a <= var_b) → !(var_a == 0)` и `!(var_a != var_c) → !(var_c == 0)`. Где есть альтернатива — там всегда есть OR. То есть либо первый путь, либо второй. В то же время узлы обоих путей *последовательно* связаны друг с другом, значит, они объединены операций AND. Таким, образом, эта ветка должна выглядеть так: IF ((`var_a > var_b`) && (`var_0 != 0`)) || (`var_a == var_c`) && (`var_c != 0`)) `printf("OK")`, прививаем ELSE к первому IF и получаем: IF (`var_d != 0`) THEN `printf("OK")` ELSE IF((`var_a > var_b`) && (`var_0 != 0`)) || (`var_a == var_c`) && (`var_c != 0`)) `printf("OK")`.

Ну а разбор второго дерева вообще тривиален: IF (`var_c == var_d`) `printf("+++")`. Итак, исходный текст дизассемблируемой программы выглядел так:

Листинг 165

```

u_int a; u_int b; ?_int c; ?_int d;
if (d) printf("TRUE");
    else
if (((a>b) && (a!=0)) || ((a==c) && (c!=0))) printf("OK\n");
if (c==d) printf("+++n");

```

Тип переменных `a` и `b` мы определили как `unsigned int`, так как результат сравнения анализировался беззнаковой условной командой — `jnb`. А вот тип переменных `c` и `d`, увы, определить так и не удалось. Однако это не умаляет значимости того факта, что мы смогли ретранслировать сложное условие, в котором без деревьев было бы немудрено и запутаться...

Больше всего следует опасаться идей, которые переходят в дела.

Ф. Херберт. Мессия дюны

Оптимизация ветвлений. Какое коварство — под флагом оптимизации сделать каждую строчку кода головоломкой. Тьфу ты, тут ящика пива не хватит, чтобы с этим справиться (а с этим лучше справляться вообще без пива — на трезвую голову). Итак, предположим, встретился вам код следующего содержания. На всякий случай, чтобы избавить вас от копания по справочникам (хотя покопаться в них лишний раз только на пользу), отмечу, что команда `SETGE` устанавливает выходной операнд в 1, если флаги состояния `SF` и `OF` равны (т. е. `SF==OF`). Иначе выходной операнд устанавливается в ноль.

Листинг 166

```
mov eax, [var_A]
xor ecx, ecx
cmp eax, 0x666
setge cl
dec ecx
and ecx, 0xFFFFFC00
add ecx, 0x300
mov [var_zzz], ecx
```

На первый взгляд этот фрагмент заимствован из какого-то хитро-запутанного защитного механизма, но нет, перед вами результат компиляции следующего тривиального выражения: *if ($a < 0x666$) zzz=0x200 else zzz=0x300*, которое в не оптимизированном виде выглядит так:

Листинг 167

```
mov eax, [var_A]
cmp eax, 0x666
jge Label_1
mov ecx, 0x100
jmp lable_2
Label_1:
mov ecx, 0x300
Label_2:
mov [var_zzz], ecx
```

Чем же компилятору не понравился такой вариант? Между прочим, он даже короче. Короче-то он короче, но содержит **ветвления**, т. е. внеплановые изменения нормального хода выполнения программы. А ветвления отрицательно сказываются на производительности хотя бы уже потому, что они приводят к очистке конвейера. Конвейер же в современных процессорах очень длинный и быстро его не заполнишь... Поэтому избавление от ветвлений путем хитроумных математических вычислений вполне оправданно и горячо приветствуется. Попутно это усложняет анализ программы, защищая ее от всех посторонних личностей типа хакеров (т. е. нас с вами).

Впрочем, если хорошенько подумать... Начнем пошагово исполнять программу, мысленно комментируя каждую строчку.

Листинг 168

```
mov eax, [var_A]
    ; eax == var_A

xor ecx,ecx
    ; ecx=0;

cmp eax, 0x666
    ; if eax<0x666 { SF=1; OF=0} else {SF=0; OF=0}

setge cl
    ; if eax<0x666 (т. е. SF==1, OF ==0) cl=0 else cl=1

dec ecx
    ; if eax<0x666 ecx=-1 else ecx=0

and ecx, 0xFFFFC00
    ; if eax<0x666 (т. е. ecx== -1) ecx=0xFFFFC00 (-0x400) else ecx=0;

add ecx, 0x300
    ; if eax<0x666 (т. е. ecx=-0x400) ecx=0x100 else ecx=0x300;

mov [esp+0x66],ecx
```

Получилось! Мы разобрались с этим алгоритмом и успешно реверсировали его! Теперь видно, что это довольно простой пример (в жизни будут нередко попадаться и более сложные). Но основная идея ясна: если встречается команда SETxx — держите нос по ветру: пахнет условными переходами! В вырожденных случаях SETxx может быть заменена на SBB (вычитание с заемом). По этому поводу решим вторую задачу:

Листинг 169

```
SUB EBX,EAX
SBB ECX,ECX
AND ECX,EBX
ADD EAX,ECX
```

Что этот код делает? Какие-то сложные арифметические действия? Посмотрим...

Листинг 170

```
SUB EBX,EAX
    ; if (EBX<EAX) SF=1 else SF=0

SBB ECX,ECX
    ; if (EBX<EAX) ECX=-1 else ECX=0

AND ECX,EBX
    ; if (EBX<EAX) ECX=EBX else ECX=0

ADD EAX,ECX
    ; if (EBX<EAX) EAX=EAX+(EBX-EAX) else EAX=EAX
```

Раскрывая скобки в последнем выражении (мы ведь не забыли, что от EBX отняли EAX?) получаем: if (EBX<EAX) EAX=EBX, т. е. это классический алгоритм поиска минимума среди двух знаковых чисел. А вот еще один пример:

Листинг 171

```
CMP EAX,1
SBB EAX,EAX
AND ECX,EAX
XOR EAX,-1
AND EAX,EBX
OR EAX,ECX
```

Попробуйте решить его сами и только потом загляните в ответ:

Листинг 172

```
CMP EAX,1
    ; if (EAX!=0) SF=0 else SF=1

SBB EAX,EAX
    ; if (EAX!=0) EAX=-1 else EAX=0

AND ECX,EAX
    ; if (EAX!=0) ECX=ECX else ECX=0

XOR EAX,-1
    ; if (EAX!=0) EAX=0 else EAX=-1

AND EAX,EBX
    ; if (EAX!=0) EAX=0 else EAX=EBX

OR EAX,ECX
    ; if (EAX!=0) EAX=ECX else EAX=EBX
```

Да... после таких упражнений тов. Буль будет во сне сниться! Но... таковы уж издержки цивилизации. К слову сказать, подавляющее большинство компиляторов достаточно лояльно относятся к условным переходам и не стремятся к их тотальному изгнанию. Так что особо напрягаться при анализе оптимизированного кода не приходится (правда, к ручной оптимизации это не относится — профессиональные разработчики выкидывают переходы в первую очередь).

Идентификация операторов switch-case-break

...когда вы видите все целиком, то у вас нет выбора, вам не из чего выбирать. Тогда вы имеете два пути одновременно, следуете одновременно этим двум направлениям.

Ошо. Пустая лодка.

Беседы по высказываниям Чжуан Цзы

Для улучшения читабельности программ в язык Си был введен оператор множественного выбора — switch. В Паскале с той же самой задачей справляется оператор CASE, кстати, более гибкий, чем его Си-аналог, но об их различиях мы поговорим позднее.

Легко показать, что switch эквивалентен конструкции IF (a == x₁) THEN оператор₁ ELSE IF (a == x₂) THEN оператор₂ IF (a == x₂) THEN оператор₂ ELSE ... оператор по умолчанию. Если изобразить это ветвление в виде логического дерева, то образуется характерная «косичка», прозванная так за сходство с завитой в косу прядью волос (рис. 29).

Казалось бы, идентифицировать switch никакого труда не составит, — даже не строя дерева, невозможно не обратить внимания на длинную цепочку гнезд, проверяющих истинность условия равенства некоторой переменной с серией непосредственных значений (сравнения переменной с другой переменной switch не допускает).

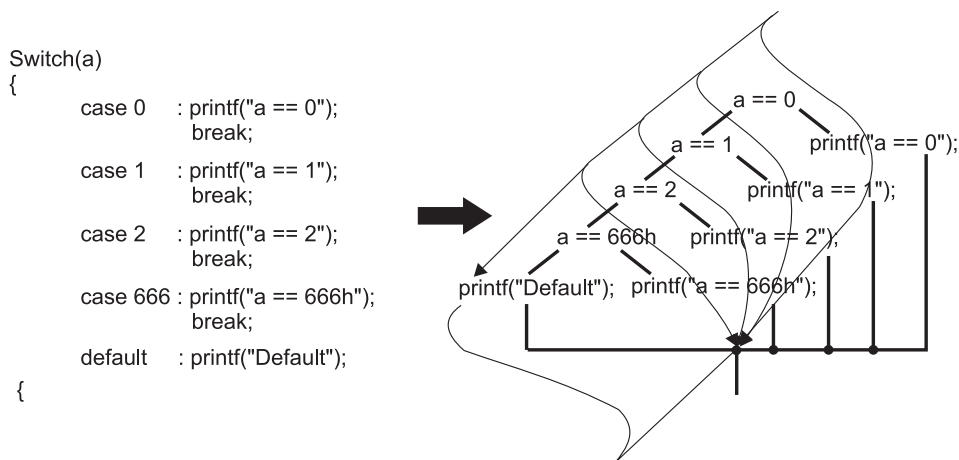


Рис. 29. Трансляция оператора switch в общем случае

Однако в реальной жизни все происходит совсем не так. Компиляторы (даже не оптимизирующие) транслируют switch в настоящий «мясной рулет», доверху нашпигованных всевозможными операциями отношений. Давайте откомпилируем приведенный выше пример компилятором Microsoft Visual C++ и посмотрим, что из этого выйдет:

Листинг 173

```

main          proc near          ; CODE XREF: start+AFp
var_tmp       = dword ptr -8
var_a         = dword ptr -4

    push     ebp
    mov      ebp, esp
    ; Открываем кадр стека

    sub      esp, 8
    ; Резервируем место для локальных переменных

    mov      eax, [ebp+var_a]
    ; Загружаем в EAX значение переменной var_a

    mov      [ebp+var_tmp], eax
    ; Обратите внимание, switch создает собственную временную переменную!
    ; Даже если значение сравниваемой переменной в каком-то ответвлении CASE
    ; будет изменено, это не повлияет на результат выборов!
    ; В дальнейшем во избежание путаницы мы будем условно называть
    ; переменную var_tmp переменной var_a

    cmp      [ebp+var_tmp], 2
    ; Сравниваем значение переменной var_a с двойкой.
    ; Хм-хм, в исходном коде CASE начинался с нуля, а заканчивался 0x666.
    ; При чем же тут двойка?!

    jg       short loc_401026
    ; Переход, если var_a > 2.
    ; Обратите на этот момент особое внимание - ведь в исходном тексте такой
    ; операции отношения не было!
    ; Причем этот переход не ведет к вызову функции printf, т. е. этот фрагмент
    ; кода получен не прямой трансляцией некой ветки case, а как-то иначе!

    cmp      [ebp+var_tmp], 2
    ; Сравниваем значение var_a с двойкой.
    ; Очевидный "прокол" компилятора - мы же только что проделывали эту
    ; операцию и с того момента не меняли никакие флаги!

    jz       short loc_40104F
    ; Переход к вызову printf("a == 2"), если var_a == 2.
    ; ОК, этот код явно получен трансляцией ветки CASE 2: printf("a == 2")

    cmp      [ebp+var_tmp], 0
    ; Сравниваем var_a с нулем

    jz       short loc_401031
    ; Переход к вызову printf("a == 0"), если var_a == 0
    ; Этот код получен трансляцией ветки CASE 0: printf("a == 0")

    cmp      [ebp+var_tmp], 1
    ; Сравниваем var_a с единицей

    jz       short loc_401040
    ; Переход к вызову printf("a == 1"), если var_a == 1.
    ; Этот код получен трансляцией ветки CASE 1: printf("a == 1")

```



```

        jmp     short loc_40106D
        ; Переход к вызову printf("Default").
        ; Этот код получен трансляцией ветки Default: printf("a == 0")

loc_401026:                                ; CODE XREF: main+10j
        ; Эта ветка получает управление, если var_a > 2
        cmp     [ebp+var_tmp], 666h
        ; Сравниваем var_a со значением 0x666

        jz      short loc_40105E
        ; Переход к вызову printf("a == 666h"), если var_a == 0x666.
        ; Этот код получен трансляцией ветки CASE 0x666: printf("a == 666h")

        jmp     short loc_40106D
        ; Переход к вызову printf("Default").
        ; Этот код получен трансляцией ветки Default: printf("a == 0")

loc_401031:                                ; CODE XREF: main+10cj
        ; // printf("A == 0")
        push    offset aA0 ; "A == 0"
        call    _printf
        add     esp, 4
        jmp     short loc_40107A
        ; ~~~~~ - а вот это оператор break, выносящий управление
        ; за пределы switch, - если бы его не было, то начали бы выполняться все
        ; остальные ветки CASE, независимо от того, к какому значению var_a они
        ; принадлежат!

loc_401040:                                ; CODE XREF: main+22j
        ; // printf("A == 1")
        push    offset aA1 ; "A == 1"
        call    _printf
        add     esp, 4
        jmp     short loc_40107A
        ; ~ break

loc_40104F:                                ; CODE XREF: main+16j
        ; // printf("A == 2")
        push    offset aA2 ; "A == 2"
        call    _printf
        add     esp, 4
        jmp     short loc_40107A
        ; ~ break

loc_40105E:                                ; CODE XREF: main+2Dj
        ; // printf("A == 666h")
        push    offset aA666h ; "A == 666h"
        call    _printf
        add     esp, 4
        jmp     short loc_40107A
        ; ~ break

loc_40106D:                                ; CODE XREF: main+24j main+2Fj
        ; // printf("Default")
        push    offset aDefault ; "Default"
        call    _printf

```

```

    add     esp, 4
loc_40107A:                                ; CODE XREF: main+3Ej main+4Dj ...
    ; // КОНЕЦ SWITCH
    mov     esp, ebp
    pop     ebp
    ; Закрываем кадр стека
    retn
main      endp

```

Построив логическое дерево (см. раздел «Идентификация IF-THEN-ELSE»), мы получим следующую картину (рис. 30). При ее изучении бросается в глаза, во-первых, условие $a > 2$, которого не было в исходной программе, а во-вторых, изменение порядка обработки case. В то же время вызовы функций printf следуют один за другим строго согласно их объявлению. Зачем же компилятор так чудит? Чего он рассчитывает этим добиться?

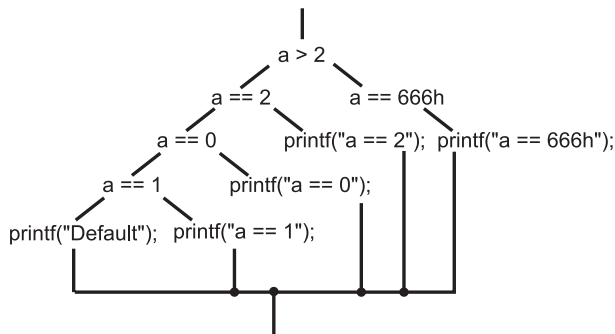


Рис. 30. Пример транслации оператора switch компилятором Microsoft Visual C

Назначение гнезда ($a > 2$) объясняется очень просто — последовательная обработка всех операторов case крайне непроизводительная. Хорошо, если их всего четыре-пять штук, а если программист натолкает в switch сотню-другую case? Процессор совсем запарится, пока их все проверит (а по закону бутерброда нужный case будет в самом конце). Вот компилятор и «утрамбовывает» дерево, уменьшая его высоту. Вместо одной ветви, изображенной на рис. 30, транслятор в нашем случае построил две, поместив в левую только числа не больше двух, а в правую — все остальные. Благодаря этому ветвь 666h из конца дерева была перенесена в его начало. Данный метод оптимизации поиска значений называют «методом вилки», но не будем сейчас на нем останавливаться, а лучше разберем его в разделе «Обрезка длинных деревьев».

Изменение порядка сравнений — право компилятора. Стандарт ничего об этом не говорит, и каждая реализация вольна поступать так, как ей это заблагорассудится. Другое дело — case-обработчики (т. е. тот код, которому case передает управление в случае истинности отношения). Они обязаны располагаться так, как были объявлены в программе, так как при отсутствии закрывающего оператора break они должны выполняться строго в порядке, замышленном программистом, хотя эта возможность языка Си используется крайне редко.

Таким образом, идентификация оператора switch не сильно усложняется: если после уничтожения узлового гнезда и прививки правой ветки к левой (или наоборот) мы получаем эквивалентное дерево и это дерево образует характерную «косичку», мы имеем дело с оператором множественного выбора или его аналогом.

Весь вопрос в том: правомерны ли мы удалять гнездо, не нарушит ли эта операция структуры дерева? Смотрим — на левой ветке узлового гнезда расположены гнезда ($a == 2$), ($a == 0$) и ($a == 1$), а на правом — ($a == 0x666$). Очевидно, если $a == 0x666$, то $a != 0$ и $a != 1$! Следовательно, прививка правой ветки к левой вполне безопасна и после такого преобразования дерево принимает вид, типичный для конструкции switch (рис. 31).

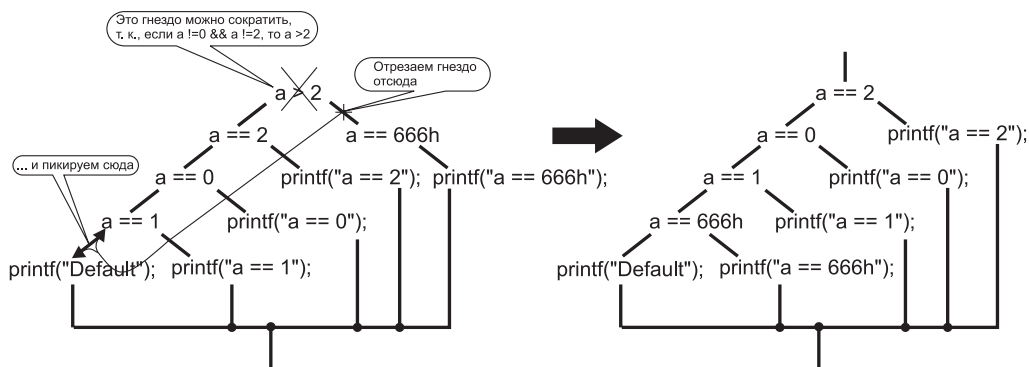


Рис. 31. Усечение логического дерева

Увы, такой простой прием идентификации срабатывает не всегда! Иные компиляторы такого наворотят, что волосы в разных местах дыбом встанут! Если откомпилировать наш пример компилятором Borland C++ 5.0, то код будет выглядеть так:

Листинг 174

```
; int __cdecl main(int argc,const char **argv,const char *envp)
_main      proc near      ; DATA XREF: DATA:00407044o

    push    ebp
    mov     ebp, esp
    ; Открываем кадр стека
    ; Компилятор помещает нашу переменную a в регистр EAX.
    ; Поскольку она не была инициализирована, то заметить этот факт
    ; не так-то легко!

    sub     eax, 1
    ; Уменьшает EAX на единицу! Что бы этого значило? Хвост Тиггера?
    ; Никакого вычитания в нашей программе не было!

    jb      short loc_401092
    ; Если EAX < 1, то переход на вызов printf("a == 0")
```

```

; (мы ведь помним, что CMP та же команда SUB, только не изменяющая операндов?).
; Ага, значит, этот код сгенерирован в результате трансляции
; ветки CASE 0: printf("a == 0").
; Внимание! Задумайтесь, какие значения может принимать EAX, чтобы
; удовлетворять условию этого отношения? На первый взгляд EAX < 1,
; в частности, 0, -1, -2,... СТОП! Ведь jb - это беззнаковая инструкция
; сравнения! А -0x1 в беззнаковом виде выглядит, как 0xFFFFFFFF
; 0xFFFFFFFF много больше единицы, следовательно, единственным подходящим
; значением будет ноль.
; Таким образом, данная конструкция - просто завуалированная проверка EAX на
; равенство нулю! (0x, и хитрый же этот Borland-компилятор!)
;

jz      short loc_40109F
; Переход, если установлен флаг нуля.
; Он будет установлен в том случае, если EAX == 1.
; И действительно, переход идет на вызов printf("a == 1")

dec     eax
; Уменьшаем EAX на единицу

jz      short loc_4010AC
; Переход, если установлен флаг нуля, а он будет установлен, когда после
; вычитания единицы командой SUB в EAX останется ровно единица,
; т. е. исходное значение EAX должно быть равно двум.
; И точно, управление передается ветке вызова printf("a == 2")!

sub     eax, 664h
; Отнимаем от EAX число 0x664

jz      short loc_4010B9
; Переход, если установлен флаг нуля, т. е. после двукратного уменьшения EAX
; равен 0x664, следовательно, исходное значение - 0x666

jmp     short loc_4010C6
; Прыгаем на вызов printf("Default"). Значит, это конец switch

loc_401092:                                ; CODE XREF: _main+6j
; // printf("a==0");
push    offset aA0 ; "a == 0"
call    _printf
pop     ecx
jmp     short loc_4010D1

loc_40109F:                                ; CODE XREF: _main+8j
; // printf("a==1");
push    offset aA1 ; "a == 1"
call    _printf
pop     ecx
jmp     short loc_4010D1

loc_4010AC:                                ; CODE XREF: _main+Bj
; // printf("a==2");
push    offset aA2 ; "a == 2"
call    _printf
pop     ecx

```

```

        jmp     short loc_4010D1

loc_4010B9:                                ; CODE XREF: _main+12j
        ; // printf("a==666");
        push    offset aA666h ; "a == 666h"
        call    _printf
        pop     ecx
        jmp     short loc_4010D1

loc_4010C6:                                ; CODE XREF: _main+14j
        ; // printf("Default");
        push    offset aDefault ; "Default"
        call    _printf
        pop     ecx

loc_4010D1:                                ; CODE XREF: _main+21j      _main+2Ej ...
        xor     eax, eax
        pop     ebp
        retn

_main    endp

```

Код, сгенерированный компилятором, модифицирует сравниваемую переменную в процессе сравнения! Оптимизатор посчитал, что DEC EAX короче, чем сравнение с константой, да и работает быстрее. Вот только нам, хакерам, от этого утешения ничуть не легче! Ведь прямая ретрансляция кода (см. раздел «Идентификация условных операторов IF-THEN-ELSE») дает конструкцию вроде: if (a- == 0) printf("a == 0"); else if (a==0) printf("a == 1"); else if (-a == 0) printf("a == 2"); else if ((a-0x664)==0) printf("a == 666h"); else printf("Default"), в которой совсем не угадывается оператор switch! Впрочем, почему это не угадывается?! Угадывается, еще как! Где есть длинная цепочка IF-THEN-ELSE-IF-THEN-ELSE... там и до switch'a недалеко! Узнать оператор множественного вы-

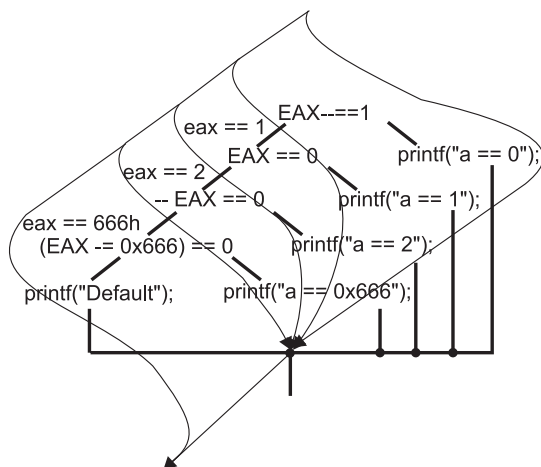


Рис. 32. Построение логического дерева с гнездами, модифицирующими саму сравниваемую переменную

бора будет еще легче, если изобразить его в виде дерева, смотрите (рис. 32), вот она, характерная «косичка»!

Другая характерная деталь — case-обработчики, точнее, оператор break традиционно замыкающий каждый из них. Они-то и образуют правую половину «косички», сходясь все вместе в точке Z. Правда, многие программисты питают паталогическую любовь к case-обработчикам размером в два-три экрана, включая в них, помимо всего прочего, и циклы (о них речь еще впереди — см. раздел «Идентификация *for\while*»), и ветвления, и даже вложенные операторы множественного выбора! В результате правая часть «косички» превращается в непроходимый таежный лес, сквозь который не проберется и стадо слонопотамов. Но даже если и так левая часть «косички», все равно останется достаточно простой и легко распознаваемой!

В заключение темы рассмотрим последний компилятор — WATCOM C. Как и следует ожидать, здесь нас подстерегают свои тонкости и «вкусности». Итак, откомпилированный им код предыдущего примера должен выглядеть так:

Листинг 175

```
main_      proc near          ; CODE XREF: __CMain+40p
    push    8
    call    __CHK
    ; Проверка стека на переполнение

    cmp     eax, 1
    ; Сравнение регистровой переменной EAX, содержащей в себе переменную a
    ; со значением 1

    jb      short loc_41002F
    ; Если EAX == 0, то переход к ветви с дополнительными проверками

    jbe     short loc_41003A
    ; Если EAX == 1 (т. е. условие bellow уже обработано выше), то переход
    ; к ветке вызова printf("a == 1");

    cmp     eax, 2
    ; Сравнение EAX со значением 2

    jbe     short loc_410041
    ; Если EAX == 2 (условие EAX <2 уже было обработано выше), то переход
    ; к ветке вызова printf("a == 2");

    cmp     eax, 666h
    ; Сравнение EAX со значением 0x666

    jz      short loc_410048
    ; Если EAX == 0x666, то переход к ветке вызова printf("a == 666h");

    jmp     short loc_41004F
    ; Что ж, ни одно из условий не подошло - переходит к ветке "Default"

loc_41002F:          ; CODE XREF: main_+Dj
    ; // printf("a == 0");
    test    eax, eax
    jnz     short loc_41004F
```

```

; Совершенно непонятно, зачем здесь дополнительная проверка?!
; Это ляп компилятора - она ни к чему!

push    offset aA0 ; "A == 0"
; Обратите внимание, WATCOM сумел обойтись всего одним вызовом printf!
; Обработчики case всего лишь передают ей нужный аргумент!
; Вот это действительно оптимизация!
jmp     short loc_410054

loc_41003A:                                ; CODE XREF: main_+Fj
; // printf("a == 1");
push    offset aA1 ; "A == 1"
jmp     short loc_410054

loc_410041:                                ; CODE XREF: main_+14j
; // printf("a == 2");
push    offset aA2 ; "A == 2"
jmp     short loc_410054

loc_410048:                                ; CODE XREF: main_+1Bj
; // printf("a == 666h");
push    offset aA666h ; "A == 666h"
jmp     short loc_410054

loc_41004F:                                ; CODE XREF: main_+1Dj main_+21j
; // printf("Default");
push    offset aDefault ; "Default"

loc_410054:                                ; CODE XREF: main_+28j main_+2Fj ...
call    printf_
; A вот он, наш printf, получающий аргументы из case-обработчиков!

add     esp, 4
; Закрытие кадра стека

retn

main_    endp

```

В общем, WATCOM генерирует более хитрый, но, как ни странно, весьма наглядный и читабельный код.

Отличия switch от оператора case языка Pascal Оператор CASE языка Pascal практически идентичен своему Си-собрату — оператору switch, хотя и близнецами их не назовешь: оператор CASE выгодно отличается поддержкой *наборов и диапазонов значений*.

Ну, если обработку наборов можно реализовать и посредством switch, правда, не так элегантно, как на Pascal (листинг 176), то проверка вхождения значения в диапазон на Си организуется исключительно с помощью конструкции IF-THEN-ELSE. Зато в Паскале каждый case-обработчик принудительно завершается неявным break, а Си-программист волен ставить (или не ставить) его по своему усмотрению.

Однако оба языка накладывают жесткое ограничение на выбор сравниваемой переменной: она должна принадлежать к перечисленному типу, а все наборы (диапазоны) значений представлять собой константы или константные выраже-

ния, вычисляемые на стадии компиляции. Подстановка переменных или вызовов функций не допускается.

Листинг 176

CASE a OF	switch(a)
begin	{
1 : WriteLn('a == 1');	case 1 : printf("a == 1");
	break;
2,4,7 : WriteLn('a == 2 4 7');	case 2 :
	case 4 :
	case 7 : printf("a == 2 4 7");
	break;
9 : WriteLn('a == 9');	case 9 : printf("a == 9");
	break;
end;	

Любопытно было бы посмотреть, как Pascal транслирует проверку диапазонов, и сравнить его с компиляторами Си. Рассмотрим следующий пример:

Листинг 177

```

VAR
    a : LongInt;
BEGIN
    CASE a OF
        2          :      WriteLn('a == 2');
        4, 6       :      WriteLn('a == 4 | 6 ');
        10..100    :      WriteLn('a == [10,100]');
    END;
END.

```

Результат его компиляции компилятором Free Pascal должен выглядеть так (для экономии места приведена лишь левая часть «косички»):

Листинг 178

```

mov     eax, ds:_A
; Загружаем в EAX значение сравниваемой переменной

cmp     eax, 2
; Сравниваем EAX со значением 0x2

j1      loc_CA      ; Конец CASE
; Если EAX < 2, то конец CASE

sub     eax, 2
; Вычитаем из EAX значение 0x2

jz      loc_9E      ; WriteLn('a == 2');
; Переход на вызов WriteLn('a == 2') если EAX == 2

sub     eax, 2
; Вычитаем из EAX значение 0x2

```



```
jz      short loc_72 ; WriteLn('a == 4 | 6');
; Переход на вызов WriteLn('a == 4 | 6') если EAX == 2 (соответственно a == 4)

sub     eax, 2
; Вычитаем из EAX значение 0x2

jz      short loc_72 ; WriteLn('a == 4 | 6');
; Переход на вызов WriteLn('a == 4 | 6') если EAX == 2 (соответственно a == 6)

sub     eax, 4
; Вычитаем из EAX значение 0x4

jl      loc_CA       ; Конец CASE
; Переход на конец CASE, если EAX < 4 (соответственно a < 10)

sub     eax, 90
; Вычитаем из EAX значение 90

jle     short loc_46 ; WriteLn('a = [10..100]');
; Переход на вызов WriteLn('a = [10..100]'), если EAX <= 90 (соответственно a <= 100)
; Поскольку случай a > 10 уже был обработан выше, то данная ветка
; срабатывает при условии a>=10 && a<=100.

jmp     loc_CA       ; Конец CASE
; Прыжок на конец CASE - ни одно из условий не подошло.
```

Как видно, Free Pascal генерирует практически тот же самый код, что и компилятор Borland C++ 5.x, поэтому его анализ не должен вызвать никаких сложностей.

Обрезка (балансировка) длинных деревьев. В некоторых (хотя и редких) случаях операторы множественного выбора содержат сотни (а то и тысячи) наборов значений, и если решать задачу сравнения в лоб, то высота логического дерева окажется гигантской до неприличия, а его прохождение займет весьма длительное время, что не лучшим образом скажется на производительности программы.

Но задумайтесь: чем, собственно, занимается оператор switch? Если отвлечься от устоявшейся идиомы: *оператор SWITCH дает специальный способ выбора одного из многих вариантов, который заключается в проверке совпадения значения данного выражения с одной из заданных констант и соответствующем ветвлении*, — то можно сказать, что switch — оператор поиска соответствующего значения. В таком случае каноническое switch-дерево представляет собой тривиальный алгоритм последовательного поиска — самый неэффективный алгоритм из всех.

Пусть, например, исходный текст программы выглядел так:

Листинг 179

```
switch (a)
{
    case 98 : ...;
    case 4  : ...;
    case 3  : ...;
    case 9  : ...;
```

```

case 22 : ...;
case 0  : ...;
case 11 : ...;
case 666: ...;
case 096: ...;
case 777: ...;
case 7  : ...;
}

```

Тогда соответствующее ему не оптимизированное логическое дерево будет достигать в высоту одиннадцати гнезд (рис. 33, слева). Причем на левой ветке корневого гнезда окажется аж десять других гнезд, а на правой вообще ни одного (только соответствующий ему case-обработчик).

Исправить «перекос» можно, разрезав одну ветку на две и привив образовавшиеся половинки к новому гнезду, содержащему условие, определяющее, в какой из веток следует искать сравниваемую переменную. Например, левая ветка может содержать гнезда с четными значениями, а правая — с нечетными. Но это плохой критерий: четных и нечетных значений редко бывает поровну, и вновь образуется перекас. Гораздо надежнее поступить так: берем наименьшее из всех значений и бросаем его в кучу А, затем берем наибольшее из всех значений и бросаем его в кучу В. Так повторяем до тех пор, пока не рассортируем все имеющиеся значения.

Поскольку оператор множественного выбора требует уникальности каждого значения, т. е. каждое число может встречаться в наборе (диапазоне) значений лишь однажды, легко показать, что: а) в обеих кучах будет содержаться равное количество чисел (в худшем случае — в одной куче окажется на число больше); б) все числа кучи А меньше наименьшего из чисел кучи В. Следовательно, достаточно выполнить только одно сравнение, чтобы определить в какой из двух куч следует искать сравниваемые значения.

Высота нового дерева будет равна $\frac{N+1}{2} + 1$, где N — количество гнезд старого дерева. Действительно, мы же делим ветвь дерева надвое и добавляем новое гнездо — отсюда и берется $\frac{N}{2}$ и +1, а (N+1) необходимо для округления результата деления в большую сторону. К примеру, если высота не оптимизированного дерева достигала 100 гнезд, то теперь она уменьшилась до 51. Что? Говорите, 51 все равно много? А что нам мешает разбить каждую из двух ветвей еще на две? Это уменьшит высоту дерева до 27 гнезд! Аналогично последующее уплотнение даст $16 \rightarrow 12 \rightarrow 11 \rightarrow 9 \rightarrow 8 \dots$ и все! Более плотная упаковка дерева невозможна (подумайте почему, на худой конец постройте само дерево). Но согласитесь, восемь гнезд — это не 100! Полное прохождение оптимизированного дерева потребует менее девяти сравнений!

«Трамбовать» логические деревья оператора множественного выбора умеют практически все компиляторы, даже не оптимизирующие! Это увеличивает производительность, но затрудняет анализ откомпилированной программы. Взгляните еще раз на рис. 33 — левое несбалансированное дерево наглядно и интуитивно

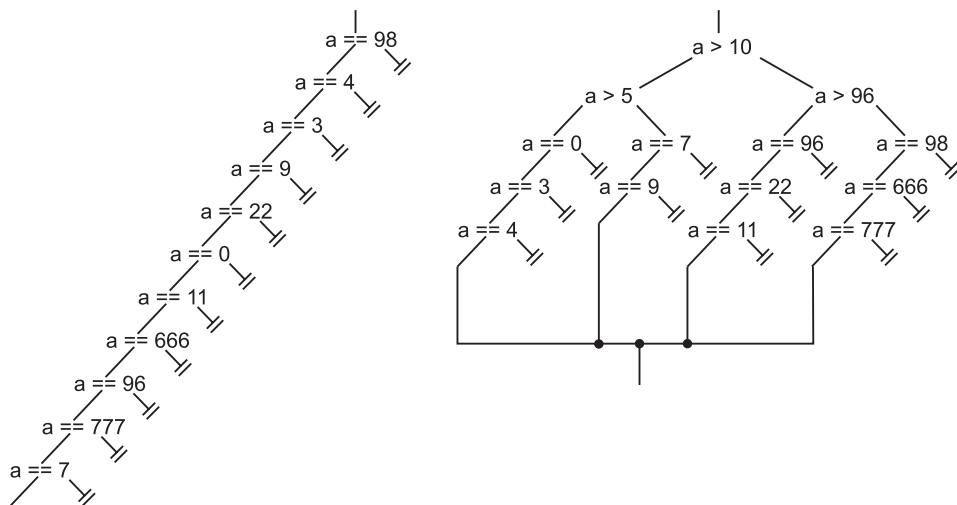


Рис. 33. Логическое дерево до утрамбовки (слева) и после (справа)

но понятно. После балансировки же (правое дерево) — в нем сам Тиггер хвост обломит.

К счастью, балансировка дерева допускает эффективное обращение. Но прежде чем засучить рукава и приготовиться к лазанью по деревьям (а Тиггеры по деревьям лазают лучше всех!), введем понятие *балансировочного узла*. Балансировочный узел не изменяет логики работы двоичного дерева и является факультативным узлом, единственная функция которого — укорачивание длины ветвей. Балансировочный узел без потери функциональности дерева может быть замещен любой из своих ветвей. Причем каждая ветвь балансирующего узла должна содержать одно или более гнезд.

Рассуждая от противного — если все узлы логического дерева, правая ветка которых содержит одно или более гнезд, могут быть замещены на эту самую правую ветку без потери функциональности дерева, то данная конструкция представляет собой оператор switch. Почему именно правая ветка? Так ведь оператор множественного выбора в развернутом состоянии представляет цепочку гнезд, соединенных левыми ветвями друг с другом, а на правых, держащих case-обработчики, вот мы и пытаемся подцепить все правые гнезда на левую ветвь. Если это удастся, мы имеем дело с оператором множественного выбора, а нет — с чем-то другим.

Рассмотрим обращение балансировки на примере следующего дерева (рис. 34, слева). Двигаясь от левой нижней ветви, мы будем продолжать взбираться на дерево до тех пор, пока не встретим узел, держащий на своей правой ветви одно или более гнезд. В нашем случае это узел ($a > 5$). Смотрите, если данный узел заменить его гнездами ($a == 7$) и ($a == 9$), функциональность дерева не нарушится! (рис. 34, посередине). Аналогично узел ($a > 10$) может быть безболезненно заменен гнездами ($a > 96$), ($a == 96$), ($a == 22$) и ($a == 11$), а узел ($a > 96$), в свою очередь, гнездами ($a == 98$), ($a == 666$) и ($a == 777$). В конце концов, обра-

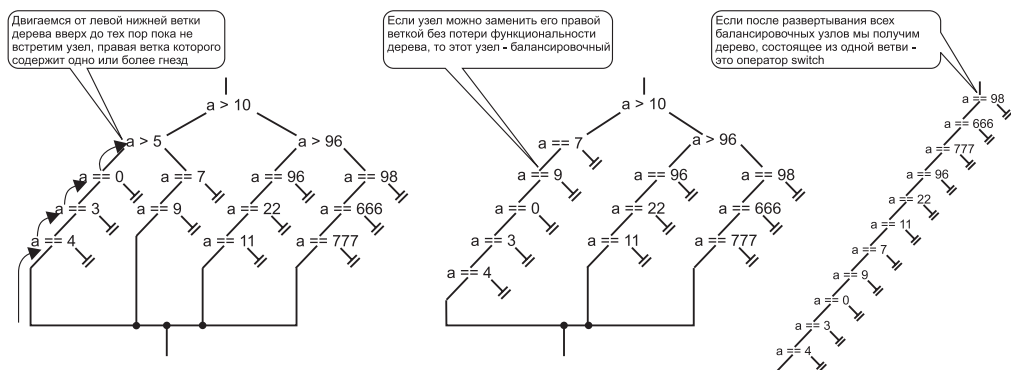


Рис. 34. Обращение балансировки логического дерева

зуется классическое switch-дерево, в котором оператор множественного выбора распознается с первого взгляда.

Сложные случаи балансировки или оптимизирующая балансировка. Для уменьшения высоты «утрамбовываемого» дерева хитрые трансляторы стремятся замещать уже существующие гнезда балансирующими узлами. Рассмотрим следующий пример (рис. 35). Для уменьшения высоты дерева транслятор разбивает его на две половины — в левую идут гнезда со значениями, меньшими или равными единице, а в правую — все остальные. Казалось бы, на правой ветке узла ($a > 1$) должно висеть гнездо ($a == 2$), а нет! Здесь мы видим узел ($a > 2$), к левой ветке которого прицеплен case-обработчик :2! А что, вполне логично — если ($a > 1$) и !($a > 2$), то $a == 2$!

Легко видеть, что узел ($a > 2$) жестко связан с узлом ($a > 1$) и работает на пару с последним. Нельзя выкинуть один из них, не нарушив работоспособности другого! Обратить балансировку дерева по описанному выше алгоритму без нарушения его функциональности невозможно! Отсюда может создаться мнение, что мы имеем дело вовсе не с оператором множественного выбора, а чем-то другим.

Чтобы развеять это заблуждение, придется предпринять ряд дополнительных шагов. Первое — у switch-дерева все case-обработчики всегда находятся на правой ветви. Смотрим, можно ли трансформировать наше дерево так, чтобы case-обработчик 2 оказался на левой ветви балансирующего узла? Да, можно: заменив ($a > 2$) на ($a < 3$) и поменяв ветви местами (другими словами, выполнив *инверсию*). Второе — все гнезда switch-дерева содержат в себе условия равенства,

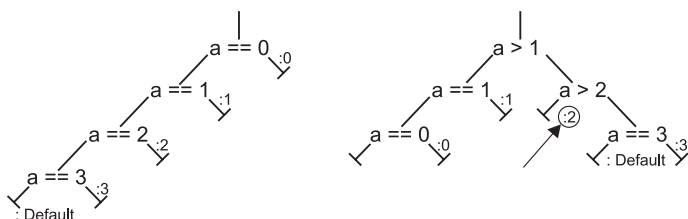


Рис. 35. Хитрый случай балансировки

смотрим, можем ли мы заменить неравенство ($a < 3$) на аналогичное ему равенство? Ну конечно же можем — ($a == 2$)!

Вот после всех этих преобразований обращение балансировки дерева удастся выполнить без труда!

Ветвления в case-обработчиках. В реальной жизни case-обработчики прямо-таки кишат ветвлениями, циклами и прочими условными переходами всех мастей. Как следствие — логическое дерево приобретает вид ничуть не напоминающий оператор множественного выбора, а скорее смахивающий на заросли чертополоха, так любимые И-и. Понятное дело, идентифицировав case-обработчики, мы могли бы решить эту проблему, но как их идентифицировать?!

Очень просто: за редкими клиническими исключениями, case-обработчики не содержат ветвлений относительно сравниваемой переменной. Действительно, конструкции `switch(a) ... case 666 : if (a == 666) ...` или `switch(a) ... case 666 : if (a > 66) ...` абсолютно лишены смысла. Таким образом, мы можем смело удалить из логического дерева все гнезда с условиями, не касающимися сравниваемой переменной (переменной корневого гнезда).

Хорошо, а если программист в порыве собственной глупости или стремлении затруднить анализ программы впяет в case-обработчики ветвления относительно сравниваемой переменной?! Оказывается, это ничуть не затруднит анализ! Впаянные ветвления элементарно распознаются и обрезаются либо как избыточные, либо как никогда не выполняющиеся. Например, если к правой ветке гнезда ($a == 3$) прицепить гнездо ($a > 0$) — его можно удалить как не несущее в себе никакой информации. Если же к правой ветке того же самого гнезда прицепить гнездо ($a == 2$), его можно удалить как никогда не выполняющееся — если $a == 3$, то заведомо $a != 2$!

Идентификация циклов

Связь между элементами системы носит трансуровневый характер и проявляется себя в виде повторяющихся единиц разных уровней (мотивов).

*Тезис классического постструктурализма
в его отечественном изводе*

Циклы — единственная (за исключением неприличного GOTO) конструкция языков высокого уровня, имеющая ссылку «назад», т. е. в область более младших адресов. Все остальные виды ветвлений — будь то IF-THEN-ELSE или оператор множественного выбора switch — всегда направлены «вниз», в область старших адресов. Вследствие этого логическое дерево, изображающее цикл, настолько характерно, что легко опознается с первого взгляда.

Существуют три основных типа цикла: *циклы с условием вначале* (рис. 36, слева), *циклы с условием в конце* (рис. 36, в центре) и *циклы с условием в сере-*

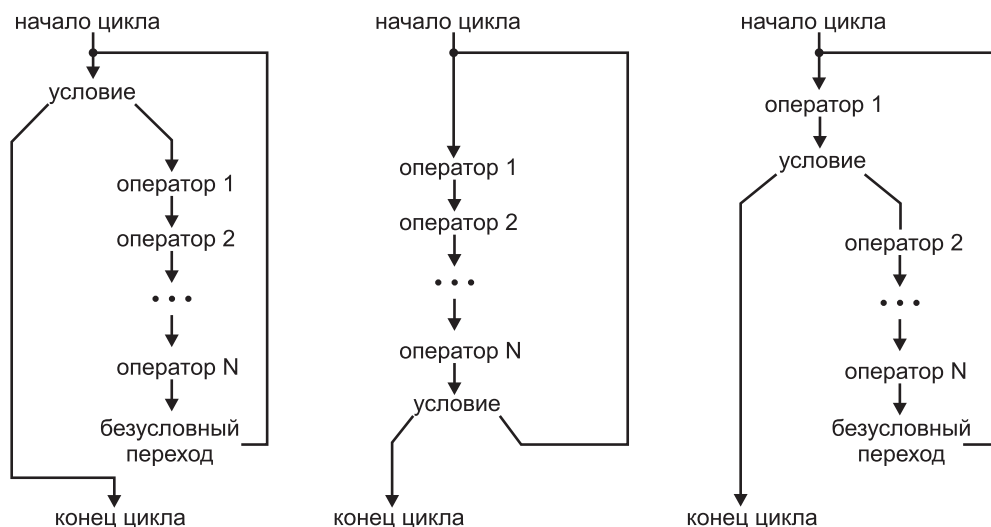


Рис. 36. Логическое дерево цикла с условием в начале (слева) и условием в конце (справа)

дине (рис. 36, справа). Комбинированные циклы имеют несколько условий в разных местах, например, в начале и конце одновременно.

В свою очередь, условия бывают двух типов: *условия завершения цикла* и *условия продолжения цикла*.

В первом случае: если условие завершения истинно происходит переход в конец цикла, иначе — его продолжение.

Во втором: если условие продолжения цикла ложно происходит переход в конец цикла, в противном случае — его продолжения. Легко показать, что условия продолжения цикла представляют собой инвертированные условия завершения. Таким образом, со стороны транслятора вполне достаточно поддержки условий одного типа. И действительно, операторы циклов `while`, `do` и `for` языка Си работают исключительно с условиями продолжения цикла. Оператор `while` языка Pascal также работает с условием продолжения цикла, и исключение составляет один лишь `repeat-until`, ожидающий условие завершения цикла.

Циклы с условиями в начале (также называемые циклами с предусловием). В языках Си и Pascal поддержка циклов с предусловием обеспечивается оператором `while` (условие), где «условие» — условие продолжения цикла. То есть цикл `while (a < 10) a++;` выполняется до тех пор, пока условие `(a > 10)` остается истинным. Однако транслятор при желании может инвертировать условие продолжения цикла на условие завершения цикла. На платформе Intel 80x86 такой трюк экономит от одной до двух машинных команд. Смотрите: на листинге 180, слева, приведен цикл с условием завершения, а справа — с условием продолжения. Как видно, цикл с условием завершения на одну команду короче! Поэтому практически все компиляторы (даже не оптимизирующие) всегда генерируют левый вариант. (А некоторые, особо одаренные, даже умеют превра-

щать циклы с предусловием в еще более эффективные циклы с постусловием (см. пункт «Циклы с условием в конце»).

Листинг 180

```
while:                                while:
    CMP A, 10                        CMP A, 10
    JAE end                          JB continue
    INC A                            JMP end
    JMP while                        continue:
end:                                  INC A
                                    JMP while
                                    end:
```

Слева показан цикл с условием завершения цикла, а справа — тот же цикл, но с условием продолжения цикла. Как видно, цикл с условием завершения на одну команду короче.

Цикл с условием завершения не может быть непосредственно отображен на оператор `while`. Кстати, об этом часто забывают начинающие, допуская ошибку «что вижу, то пишу»: `while (a >= 10) a++`. С таким условием данный цикл вообще не выполнится ни разу! Но как выполнить инверсию условия и при этом гарантированно не ошибиться? Казалось бы, что может быть проще, а вот попросите знакомого хакера назвать операцию, обратную «больше». Очень может быть (даже наверняка!) ответом будет... «меньше». А вот и нет, правильный ответ «меньше или равно». Полный перечень обратных операций отношений можно найти в табл. 25, приведенной ниже.

Таблица 25. Обратные операции отношения

Логическая операция	Обратная логическая операция
==	!=
!=	==
>	<=
<	>=
<=	>
>=	<

Циклы с условием в конце (также называемые циклами с пост-условием). В языке Си поддержка циклов с постусловием обеспечивается парой операторов `do` — `while`, а в языке Pascal — `repeat` \ `until`. Циклы с постусловием без каких-либо проблем непосредственно отображаются с языка высокого уровня на машинный код, и соответственно наоборот. То есть, в отличие от циклов с предусловием, инверсии условия не происходит.

Например, `do a++; while (a<10)` в общем случае компилируется в следующий код (обратите внимание, в переходе использовалась та же самая операция

отношения, что и в исходном цикле, красота и никаких ошибок при декомпиляции):

Листинг 181

```
repeat:
    INC A
    CMP A, 10
    JB repeat
end:
```

Вернувшись на страницу назад, сравним код цикла с постусловием с кодом цикла с предусловием. Не правда ли, цикл с условием в конце компактнее и быстрее? Некоторые компиляторы (например, Microsoft Visual C++) умеют транслировать циклы с предусловием в циклы с постусловием. На первый взгляд это вопиющая самодетельность компилятора, — если программист хочет проверять условие в начале, то какое право имеет транслятор ставить его в конце?! На самом же деле разница между «до» и «после» не столь значительна. Если компилятор уверен, что цикл выполняется хотя бы один раз, то он вправе выполнять проверку когда угодно. Разумеется, при этом необходимо несколько скорректировать условие проверки: `while (a < b)` не эквивалентно `do ... while (a < b)`, так как в первом случае при `(a == b)` уже происходит выход из цикла, а во втором цикл выполняет еще одну итерацию. Однако этой беде легко помочь: увеличим `a` на единицу (`do ... while ((a+1) < b)`) или вычтем эту единицу из `b` (`do ... while (a < (b-1))`), и... теперь все будет работать!

Спрашивается: и на кой все эти извращения, значительно раздувающие код? Дело в том, что блок статического предсказания направления ветвлений процессоров Pentium оптимизирован именно под переходы, направленные назад, т. е. в область младших адресов. Поэтому циклы с постусловием должны выполняться несколько быстрее аналогичных им циклов с предусловием.

Циклы со счетчиком. Циклы со счетчиком (`for`) не являются самостоятельным типом циклов, а представляют собой всего лишь синтаксическую разновидность циклов с предусловием. В самом деле, `for (a = 0; a < 10; a++)` в первом приближении то же самое, что и: `a = 0; while (a < 10) {...; a++;}`. Однако результаты компиляции двух этих конструкций не обязательно должны быть идентичны друг другу!

Оптимизирующие компиляторы (да и значительная часть не оптимизирующих) поступают хитрее, передавая после инициализации переменной-счетчика управление на команду проверки условия выхода из цикла. Образовавшаяся конструкция, во-первых, характерна и при анализе программы — сразу бросается в глаза, — а во-вторых, не может быть непосредственно отображена на циклы `while` языка высокого уровня. Смотрите:

Листинг 182

```
MOV A, xxx           ; Инициализация переменной "счетчика"
JMP conditional      ; Переход к проверке условия продолжения цикла
repeat:              ; Начало цикла
```



```

...           ; // ТЕЛО
...           ; //      ЦИКЛА
ADD A, xxx [SUB A, xxx] ; Модификация счетчика
conditional:   ; Проверка условия продолжения цикла
CMP A, xxx     ; ~
Jxx repeat     ; Переход в начало цикла, если условие истинно

```

Непосредственный прыжок вниз может быть результатом компиляции и цикла `for`, и оператора `GOTO`, но `GOTO` сейчас не в моде и используется крайне редко, а без него оператор условного перехода `IF-THEN` не может прыгнуть непосредственно в середину цикла `while`! Выходит, из всех «кандидатов» остается только цикл `for`.

Некоторые особо продвинутые компиляторы (Microsoft Visual C++, Borland C++, но не WATCOM C) поступают хитрее: анализируя код, они еще на стадии компиляции пытаются определить, выполняется ли данный цикл хотя бы один раз и, если видят, что он действительно выполняется, превращают `for` в типичный цикл с постусловием:

Листинг 183

```

MOV A, xxx           ; Инициализация переменной "счетчика"
repeat:              ; Начало цикла
...                  ; // ТЕЛО
...                  ; //      ЦИКЛА
ADD A, xxx [SUB A, xxx] ; Модификация счетчика
CMP A, xxx           ; Проверка условия продолжения цикла
Jxx repeat           ; Переход в начало цикла, если условие истинно

```

Наконец, самые совершенные компиляторы (из которых автор на вскидку может назвать один лишь Microsoft Visual C++ 6.0) могут даже заменять циклы с приращением на циклы с убыванием при условии, что параметр цикла не используется операторами цикла, а лишь прокручивает цикл определенное число раз. Зачем это компилятору? Оказывается, циклы с убыванием гораздо короче — одnobайтовая инструкция `DEC` не только уменьшает операнд, но и выставляет `Zero`-флаг при достижении нуля. В результате, в команде `CMP A, xxx` отпадает всякая необходимость.

Листинг 184

```

MOV A, xxx           ; Инициализация переменной "счетчика"
repeat:              ; Начало цикла
...                  ; // ТЕЛО
...                  ; //      ЦИКЛА
DEC A                ; Декремент счетчика
JNZ repeat           ; Повтор, пока A != 0

```

Таким образом, в зависимости от настроек и характера компилятора, циклы `for` могут транслироваться и в циклы с предусловием, и в циклы с постусловием, начинающими свое выполнение с проверки условия продолжения цикла. Причем условие продолжения может инвертироваться в условие завершения, а возрастающий цикл может волшебным образом превращаться в убывающий.

Такая неоднозначность затрудняет идентификацию циклов `for`, надежно отождествляются лишь циклы, начинающиеся с проверки постусловия (так как они не могут быть отображены на `do` без использования `GOTO`). Во всех остальных случаях никаких строгих рекомендаций по распознаванию `for` дать невозможно.

Скажем так: если логика исследуемого цикла синтаксически удобнее выражается через оператор `for`, то и выражайте ее через `for`! В противном случае используйте `while` или `do (repeat \until)` для циклов с пред- и постусловием соответственно.

И в заключение пара слов о «кастрированных» циклах — язык Си позволяет опустить инициализацию переменной цикла, условие выхода из цикла, оператор приращения переменной или все это вместе. При этом `for` вырождается во `while` и становится практически неотличимым от него.

Циклы с условием в середине. Популярными языками высокого уровня непосредственно не поддерживают циклы с условием в середине, хотя необходимость в них возникает достаточно часто. Поэтому программисты их реализуют на основе уже имеющихся циклов `while` (`while \do`) и оператора выхода из цикла `break`. Например:

Листинг 185

<pre>while(1) { ... if (условие) break; ... }</pre>	<pre>repeat: ... CMP xxx Jxx end ... JMP repeat end:</pre>
---	--

Компилятор (если он не совсем осел — Ии в смысле) разворачивает бесконечный цикл в безусловный переход `JMP`, направленный, естественно, назад (ослы генерируют код `like` — `MOV EAX, 1 \CMP EAX, 1 \JZ repeat`). Безусловный переход, направленный назад, весьма характерен — за исключением бесконечного цикла, его может порождать один лишь оператор `GOTO`, но `GOTO`, как не раз говорилось, уже давно не в моде. А раз у нас есть бесконечный цикл, то условие его завершения может находиться лишь в середине этого цикла (сложные случаи многопоточных защит, модифицирующих из соседнего потока безусловный переход в `NOP`, мы пока не рассматриваем). Остается прочесть тело цикла и найти это самое условие.

Сделать это будет нетрудно — оператор `break` транслируется в переход на первую команду, следующую на `JMP repeat`, а сам `break` получает управление от ветки `IF (условие) — THEN — [ELSE]`. Условие ее срабатывания и будет искомым условием завершения цикла. Вот, собственно, и все.

Циклы с множественными условиями выхода. Оператор `break` позволяет организовать выход из цикла в любом удобном для программиста месте, поэтому любой цикл может иметь множество условий выхода беспорядочно разбросанных по его телу. Это ощутимо усложняет анализ дизассемблируемой программы, так

как возникает риск «прозевать» одно из условий завершения цикла, что приведет к неправильному пониманию логики программы.

Идентифицировать же условия выхода из цикла очень просто — они всегда направлены «вниз», т. е. в область старших адресов, и указывают на команду, непосредственно следующую за инструкцией условного (безусловного) перехода, направленного «вверх», в область младших адресов (см. также «Циклы с условием в середине»).

Циклы с несколькими счетчиками. Оператор «запятая» языка Си позволяет осуществлять множественную инициализацию и модификацию счетчиков цикла `for`. Например: `for (a=0, b=10; a != b; a++, b—)`. А как насчет нескольких условий завершения? И «ветхий» и «новый» заветы (первое и второе издание K&R соответственно), и стандарт ANSI C, и руководства по C, прилагаемые к компиляторам Microsoft Visual C, Borland C, WATCOM C, на этот счет хранят партизанское гробовое молчание.

Если попробовать скомпилировать следующий код: `for (a=0, b=10; a >0, b <10; a++, b—)`, он будет благополучно «проглочен» практически всеми компиляторами без малейших ругательств с их стороны, но ни один из них не откомпилирует данный пример *правильно*. Логическое условие ($a_1, a_2, a_3, \dots, a_n$) лишено смысла, и компиляторы без малейших колебаний и зазрений совести отбросят все, кроме самого правого выражения a_n . Оно-то и будет единолично определять условие продолжения цикла. Один лишь WATCOM вяло ворчит по этому поводу: **Warning! W111: Meaningless use of an expression: the line contains an expression that does nothing useful. In the example "i = (1,5);", the expression "1," is meaningless. This message is also generated for a comparison that is useless.**

Если условие продолжения цикла зависит от нескольких переменных, то их сравнения следует объединить в одно выражение посредством логических операций OR, AND и др. Например: `for (a=0, b=10; (a >0 && b <10); a++, b—)` — цикл прерывается сразу же, как только одно из двух условий станет ложно; `for (a=0, b=10; (a >0 || b <10); a++, b—)` — цикл продолжается до тех пор, пока истинно хотя бы одно условие из двух.

В остальном же циклы с несколькими счетчиками транслируются аналогично циклам с одним счетчиком, за исключением того, что инициализируется и модифицируется не одна, а сразу несколько переменных.

Идентификация continue. Оператор `continue` приводит к непосредственной передаче управления на код проверки условия продолжения (завершения) цикла. В общем случае он транслируется в безусловный `jump`, в циклах с предусловием, направленным вверх, а в циклах с постусловием — вниз. Код, следующий за `continue`, уже не получает управления, поэтому `continue` практически всегда используется в условных конструкциях.

Например, `while (a++ < 10) if (a == 2) continue;...` компилируется приблизительно так:

Листинг 186

```
repeat:          ; Начало цикла while
```


Два конца и два начала вполне напоминают два цикла, из которых один вложен в другой. Правда, начала обоих циклов совмещены, но ведь может же такое быть, если в цикл с постусловием вложен цикл с предусловием? На первый взгляд да, но если подумать, то... ай-ай-ай! А ведь *условие1* выхода из цикла прыгает аж за второй конец! Если это предусловие вложенного цикла, то оно прыгало бы за первый конец. А если *условие1* — предусловие материнского цикла, то конец вложенного цикла не смог бы передать на него управление. Выходит, это не два цикла, а один. А первый «конец» — результат трансляции оператора `continue`.

С разбором сложных условий продолжения цикла с постусловием дела обстоят еще лучше. Рассмотрим такой пример:

```
do
{
    ...
} while(условие1 || условие2);
```

Результат его трансляции в общем случае будет выглядеть так:

```
...
условие продолжения1
условие продолжения2
```

Ну, чем не:

```
do
{
    do
    {
        ...
    }while(условие1)
}while(условие2)
```

Строго говоря, предложенный вариант является логически верным, но синтаксически некрасивым. Материнский цикл крутит в своем теле один лишь вложенный цикл и не содержит никаких других операторов. Так зачем он тогда, спрашивается, нужен? Объединить его с вложенным циклом в один!

Дизассемблерные листинги примеров. Давайте для закрепления сказанного рассмотрим несколько живых примеров.

Начнем с самого простого — с циклов `while\do`:

Листинг 187. Демонстрация идентификации циклов `while\do`

```
#include <stdio.h>

main()
{
    int a=0;
    while(a++<10) printf("Оператор цикла while\n");

    do {
        printf("Оператор цикла do\n");
    } while(--a >0);
```

```
}

```

Результат компиляции этого примера компилятором Microsoft Visual C++ 6.0 с настройками по умолчанию должен выглядеть так:

Листинг 188

```
main          proc near          ; CODE XREF: start+AFp
var_a         = dword ptr -4

    push     ebp
    mov      ebp, esp
    ; Открываем кадр стека

    push     ecx
    ; Резервируем память для одной локальной переменной

    mov      [ebp+var_a], 0
    ; Заносим в переменную var_a значение 0x0

loc_40100B:    ; CODE XREF: main_401000+29j
    ; ~~~~~
    ; Перекрестная ссылка, направленная вниз,
    ; говорит о том, что это начало цикла.
    ; Естественно, раз перекрестная ссылка направлена вниз, то переход,
    ; ссылающийся на этот адрес, будет направлен вверх!

    mov      eax, [ebp+var_a]
    ; Загружаем в EAX значение переменной var_a

    mov      ecx, [ebp+var_a]
    ; Загружаем в ECX значение переменной var_a
    ; (недальновидность компилятора - можно было бы поступить и короче MOV ECX, EAX)

    add      ecx, 1
    ; Увеличиваем ECX на единицу

    mov      [ebp+var_a], ecx
    ; Обновляем var_a

    cmp      eax, 0Ah
    ; Сравниваем старое (до обновления) значение переменной var_a с числом 0xA

    jge      short loc_40102B
    ; Если var_a >= 0xA - прыжок "вперед", непосредственно за инструкцию
    ; безусловного перехода, направленного "назад".
    ; Раз "назад", значит, это цикл, а поскольку условие выхода из цикла
    ; проверяется в его начале, то это цикл с предусловием.
    ; Для его отображения на цикл while необходимо инвертировать условие выхода
    ; из цикла на условие продолжения цикла (т. е. заменить >= на <).
    ; Сделав это, мы получаем:
    ; while (var_a++ < 0xA)...
    ;

// Начало тела цикла
    push     offset aOperatorCiklaW ; "Оператор цикла while\n"
    call     _printf

```

```

    add     esp, 4
    ; printf("Оператор цикла while\n")

    jmp     short loc_40100B
    ; Безусловный переход, направленный назад, на метку loc_40100B.
    ; Между loc_40100B и jmp short loc_40100B
    ; есть только одно условие выхода из цикла -
    ; jge short loc_40102B,
    ; значит, исходный цикл выглядел так:
    ; while (var_a++ < 0xA) printf("Оператор цикла while\n")

loc_40102B:                                ; CODE XREF: main_401000+1A7j
                                           ; main_401000+457j
                                           ; 0000000000000000
    ; // Это начало цикла с постусловием
    ; // Однако на данном этапе мы этого еще не знаем,
    ; // хотя и можем догадываться
    ; // благодаря наличию перекрестной ссылки, направленной вниз.

    ; Ага, никакого условия в начале цикла не присутствует, значит, это цикл
    ; с условием в конце или середине
    push    offset a0operatorCiklaD ; "Оператор цикла do\n"
    call    _printf
    add     esp, 4
    ; printf("Оператор цикла do\n")
    ; // Тело цикла

    mov     edx, [ebp+var_a]
    ; Загружаем в EDX значение переменной var_a

    sub     edx, 1
    ; Уменьшаем EDX на единицу

    mov     [ebp+var_a], edx
    ; Обновляем переменную var_a

    cmp     [ebp+var_a], 0
    ; Сравниваем переменную var_a с нулем

    jg      short loc_40102B
    ; Если var_a > 0, то переход в начало цикла.
    ; Поскольку условие расположено в конце тела цикла, этот цикл - do:
    ; do printf("Оператор цикла do\n"); while (--a > 0)
    ;
    ; // Для повышения читабельности дизассемблерного текста рекомендуется
    ; // заменить префиксы loc_ в начале цикла на while и do (repeat) в циклах
    ; // с пред- и постусловием соответственно

    mov     esp, ebp
    pop     ebp
    ; Закрываем кадр стека
    retn

main     endp

```

Совсем другой результат получится, если включить оптимизацию.

Откомпилируем тот же самый пример с ключом /Ox (максимальная оптимизация) и посмотрим на результат, выданный компилятором:

Листинг 189

```
main          proc near          ; CODE XREF: start+AFp
    push     esi
    push     edi
    ; Сохраняем регистры в стеке

    mov     esi, 1
    ; Присваиваем ESI значение 0x1
    ; Внимание! Взгляните на исходный код – ни одна из переменных не имела
    ; такого значения!

    mov     edi, 0Ah
    ; Присваиваем EDI значение 0xA. Ага, это константа для проверки условия
    ; выхода из цикла

loc_40100C:    ; CODE XREF: main+1Dj
    ; ; ~~~~~
    ; Судя по перекрестной ссылке, направленной вниз, это – цикл!

    push     offset a0operatorCiklaW ; "Оператор цикла while\n"
    call     _printf
    add     esp, 4
    ; printf("Оператор цикла while\n")
    ; ...тело цикла while? (растерянно так).
    ; Постой, постой! А где же предусловие?!

    dec     edi
    ; Уменьшаем EDI на один

    inc     esi
    ; Увеличиваем ESI на один

    test     edi, edi
    ; Проверяем EDI на равенство нулю

    ja      short loc_40100C
    ; Переход в начало цикла, пока EDI != 0
    ; Так... (задумчиво). Компилятор в порыве оптимизации превратил неэффективный
    ; цикл с предусловием в более компактный и быстрый цикл с постусловием.
    ; Имел ли он на это право? А почему нет?! Проанализировав код, компилятор понял,
    ; что данный цикл выполняется, по крайней мере, один раз, следовательно,
    ; скорректировав условие продолжения, его проверку можно вынести в конец цикла.
    ; Поэтому-то начальное значение переменной цикла равно единице, а не нулю!
    ; То есть while ((int a = 0) < 10) компилятор заменил на
    ; do ... while (((int a = 0)+1) < 10) == do ... while ((int a=1) < 10)
    ;
    ; Причем, что интересно, он не сравнивал переменную цикла с константой,
    ; а поместил константу в регистр и уменьшал его до тех пор, пока тот не стал
    ; равен нулю! Зачем? А затем, что так короче, да и работает быстрее.
    ; Что ж, это все хорошо, но как нам декомпилировать этот цикл?
    ; Непосредственное отображение на язык Си дает следующую конструкцию:
    ; var_ ESI = 1; var _EDI = 0xA;
```



```

; do {
;;printf("Оператор цикла while\n"); var_EDI--; var_ESI++;
; } while(var_EDI > 0)
;
; Правда, коряво и запутанно? Что ж, тогда попытаемся избавиться от одной
; из двух переменных. Это действительно возможно, так как они модифицируются
; синхронно, и var_EDI = 0xB - var_ESI
; ОК, выполняем подстановку:
; var_ ESI = 1; var _EDI = 0xB - var_ESI ; (== 0xA;)
; do {
;;printf("Оператор цикла while\n"); var_EDI--; var_ESI++;
; } while((0xB - var_ESI) > 0); (== var_ESI > 0xB)
;
; Это мы вообще сокращаем, так как var_EDI уже выражена через var_ESI
;
; Что ж, уже получается нечто осмысленное:
;
; var_ ESI = 1; var _EDI == 0xA;
; do {
;; printf("Оператор цикла while\n"); var_ESI++;
; } while(var_ESI > 0xB)
; На этом можно и остановиться, а можно и пойти дальше, преобразовав цикл
; с постусловием в более наглядный цикл с предусловием
;
; var_ ESI = 1; var _EDI == 0xA; ← var_EDI не используется, можно сократить
; while (var_ESI <= 0xA) {
;   printf("Оператор цикла while\n"); var_ESI++;
; }
; Но и это не предел выразительности: во-первых, var_ESI <= 0xA эквивалентно
; var_EDI < 0xB, а во-вторых, поскольку переменная var_ESI используется лишь
; как счетчик, ее начальное значение можно безбоязненно привести к нулевому
; значению, а операцию инкремента внести в сам цикл:
;
; var_ ESI = 0;
; while (var_ESI++ < 0xA) ← вычитаем единицу из левой и правой половины
;   printf("Оператор цикла while\n");
;
; Ну разве не красота?! Сравните этот вариант с первоначальным,
; насколько он стал яснее и понятнее

```

```

loc_40101F:                                ; CODE XREF: main+2Fj
                                           ; .....

```

; Перекрестная ссылка, направленная вниз, говорит о том, что это начало цикла.

; // Предусловия нет, - значит, это цикл do

```

push    offset aOperatorCiklaD ; "Оператор цикла do\n"
call    _printf
add     esp, 4
; printf("Оператор цикла do\n");

```

```

dec     esi
; Уменьшаем var_ESI

```

```

test    esi, esi

```

```

; Проверка ESI на равенство нулю

jg      short loc_40101F
; Продолжать цикл, пока var_ESI > 0
;
; ОК. Этот цикл легко и непринужденно отображается на язык Си:
; do printf("Оператор цикла do\n"); while (--var_ESI > 0 )

pop     edi
pop     esi
; Восстанавливаем сохраненные регистры

ret     retn

main    endp

```

Несколько иначе оптимизирует циклы компилятор Borland C++ 5.x. Смотрите:

Листинг 190

```

_main    proc near                ; DATA XREF: DATA:00407044o

push     ebp
mov      ebp, esp
; Открываем кадр стека

push     ebx
; Сохраняем EBP в стеке

xor      ebx, ebx
; Присваиваем регистровой переменной EBX значение ноль.
; Как легко догадаться, EBX и есть "a"

jmp      short loc_40108F
; Безусловный прыжок вниз. Очень похоже на цикл for...

loc_401084:                        ; CODE XREF: _main+19j
;                                     ; 0000000000000000
; Перекрестная ссылка, направленная вниз, - значит, это начало какого-то цикла

push     offset a0operatorCiklaW ; "Оператор цикла while\n"
call     _printf
pop      ecx
; printf("Оператор цикла while\n")

loc_40108F:                        ; CODE XREF: _main+6j
; А вот сюда был направлен самый первый jump
; Посмотрим, что же это такое.

mov      eax, ebx
; Копирование EBX в EAX

inc      ebx
; Увеличение EBX

cmp      eax, 0Ah
; Сравнение EAX со значением 0xA

jle      short loc_401084
; Переход в начало цикла, если EAX < 0xA.

```

```

; Вот так-то Borland оптимизировал код! Он расположил условие в конце цикла,
; но, чтобы не транслировать цикл с предусловием в цикл с постусловием,
; просто начал выполнение цикла с этого самого условия!
;
; Отображение этого цикла на язык Си дает:

; for (int a=0; a < 10; a++) printf("Оператор цикла while\n")
;
; и, хотя подлинный цикл выглядел совсем не так, наш вариант ничем не хуже!
; (а может, даже и лучше – нагляднее)

loc_401097:                                ; CODE XREF: _main+29j
                                           ; 000000000000000000000000

; Начало цикла!

; Условия нет, – значит, это цикл с постусловием

push    offset aOperatorCiklaD ; "Оператор цикла do\n"
call    _printf
pop     ecx
; printf("Оператор цикла do\n")

dec     ebx
; -var_EBX

test    ebx, ebx
jg      short loc_401097
; Продолжать цикл, пока var_EBX > 0
; do printf("Оператор цикла do\n"); while (--var_EBX > 0)

xor     eax, eax
; return 0

pop     ebx
pop     ebp
; Восстанавливаем сохраненные регистры

ret     0

_main    endp

```

Остальные компиляторы генерируют аналогичный или даже еще более примитивный и очевидный код, поэтому не будем подробно их разбирать, а лишь кратко опишем используемые ими схемы трансляции.

Компилятор Free Pascal 1.x ведет себя аналогично компилятору Borland C++ 5.0, всегда помещая условие в конец цикла и начиная с него выполнение while-циклов.

Компилятор WATCOM C не умеет преобразовывать циклы с предусловием в циклы с постусловием, вследствие чего располагает условие выхода из цикла в начале while-циклов, а в их конец вставляет безусловный jump. (Классика!)

Компилятор GCC вообще не оптимизирует циклы с предусловием, генерируя самый неоптимальный код. Смотрите:

Листинг 191

```
mov     [ebp+var_a], 0
```

```

; Присвоение переменной a значения 0
mov     esi, esi
; Э... на редкость умный код! При его виде трудно не упасть со стула!
loc_401250:                                ; CODE XREF: sub_40123C+34j
;                                             ; ~~~~~
; Начало цикла

mov     eax, [ebp+var_a]
; Загрузка в EAX значения переменной var_a

inc     [ebp+var_a]
; Увеличение var_a на единицу

cmp     eax, 9
; Сравнение EAX со значением 0x9

jle     short loc_401260
; Переход, если EAX <= 0x9 (EAX < 0xA)

jmp     short loc_401272
; Безусловный переход в конец цикла.
; Стало быть, предыдущий условный переход - переход на его продолжение.
; Какой неоптимальный код! Зато нет инверсии условия продолжения цикла,
; что упрощает дизассемблирование

align 4
; Выравнивание перехода по адресам, кратным четырем, ускоряет код, но заметно
; увеличивает его размер (особенно если переходов очень много)

loc_401260:                                ; CODE XREF: sub_40123C+1Dj
add     esp, 0FFFFFFF4h
; Вычитание из ESP значения 12 (0xC)

push    offset aOperatorCiklaW ; "Оператор цикла while\n"
call    printf
add     esp, 10h
; Восстанавливаем стек (0xC + 0x4 ) == 0x10

jmp     short loc_401250
; Переход в начало цикла

loc_401272:
; Конец цикла.

```

Разобравшись с while\do, перейдем к циклам for. Рассмотрим следующий пример:

Листинг 192. Демонстрация идентификации циклов for

```

#include <stdio.h>

main()
{
    int a;
    for (a=0;a<10;a++)    printf("Оператор цикла for\n");
}

```

Результат компиляции Microsoft Visual C++ 6.0 с настройками по умолчанию будет выглядеть так:

Листинг 193

```
main          proc near          ; CODE XREF: start+AFp
var_a         = dword ptr -4

    push      ebp
    mov       ebp, esp
    ; Открываем кадр стека

    push      ecx
    ; Резервируем память для локальной переменной

    mov       [ebp+var_a], 0
    ; Присваиваем локальной переменной var_a значение 0

    jmp       short loc_401016
    ; Непосредственный переход на код проверки условия продолжения цикла -
    ; характерный признак for

loc_40100D:    ; CODE XREF: main+29j
               ; 0000000000000000
    ; Перекрестная ссылка, направленная вниз, говорит о том, что это начало цикла

    mov       eax, [ebp+var_a]
    ; Загрузка в EAX значения переменной var_a

    add       eax, 1
    ; Увеличение EAX на единицу

    mov       [ebp+var_a], eax
    ; Обновление EAX. Следовательно, исходный код выглядел так:
    ; ++a

loc_401016:    ; CODE XREF: main+Bj
    cmp       [ebp+var_a], 0Ah
    ; Сравниваем var_a со значением 0xA

    jge       short loc_40102B
    ; Выход из цикла, если var_a >= 0xA

    push      offset a0operatorCiklaF ; "Оператор цикла for\n"
    call      _printf
    add       esp, 4
    ; printf("Оператор цикла for\n")

    jmp       short loc_40100D
    ; Безусловный переход в начало цикла
    ;
    ; Итак, что мы имеем?
    ; инициализация переменной var_a
    ; переход на проверку условия выхода из цикла
    ; инкремент переменной var_a
    ; проверка условия относительно var_a
    ; прыжок на выход из цикла, если условие истинно
```

```

; вызов printf
; переход в начало цикла
; конец цикла
;
; Проверка на завершения, расположенная в начале цикла, говорит о том, что
; это цикл с предусловием, но непосредственно выразить его через while
; не удастся - мешает безусловный переход в середину цикла, минуя код
; инкремента переменной var_a.
; Однако этот цикл с легкостью отображается на оператор for, смотрите:
; for (a = 0; a < 0xA; a++) printf("Оператор цикла for\n")
;
; Действительно, цикл for сначала инициализирует переменную - счетчик,
; затем проверяет условие продолжения цикла
; (оптимизируемое компилятором в условие завершение), далее выполняет
; оператор цикла, модифицирует счетчик, вновь проверяет условие и т. д.
;
loc_40102B:                                ; CODE XREF: main+1A j
mov     esp, ebp
pop     ebp
; Закрываем кадр стека

ret     ret

main                                         endp

```

А теперь задействуем оптимизацию и посмотрим, как видоизменится наш цикл:

Листинг 194

```

main                                         proc near                                ; CODE XREF: start+AF p
push    esi
mov     esi, 0Ah
; Инициализируем переменную - счетчик.
; Внимание! В исходном коде начальное значение счетчика равнялось нулю!

loc_401006:                                ; CODE XREF: main+14 j
push    offset a0operatorCiklaF ; "Оператор цикла for\n"
call    _printf
add     esp, 4
; printf("Оператор цикла for\n")
; Выполняем оператор цикла! Причем безо всяких проверок!
; Хитрый компилятор проанализировал код и понял, что цикл выполняется
; по крайней мере один раз!

dec     esi
; Уменьшаем счетчик, хотя в исходном коде программы мы его увеличивали!
; Ну правильно - dec \ jnz намного короче INC\ CMP reg, const\ jnz xxx
; Ой и мудрит компилятор! Кто же ему давал право так изменять цикл?!
; А очень просто - он понял, что параметр цикла в самом цикле используется
; только как счетчик и нет никакой разницы - увеличивается он
; с каждой итерацией или уменьшается!

jnz     short loc_401006
; Переход в начало цикла, если ESI > 0.

```

```

;
; М-да, по внешнему виду это типичный
; a = 0xa; do printf("Оператор цикла for\n"); while (-a);
;
; Если вас устраивает читабельность такой формы записи – оставляйте ее,
; а если нет – for (a = 0; a < 10; a++) Оператор цикла for\n");
;
; Постой, постой! На каком основании автор выполнил такое преобразование?!
; А на том же самом, что и компилятор: раз параметр цикла используется только
; как счетчик, законна любая запись, выполняющая цикл ровно десять раз,
; остается выбрать ту, которая удобнее (с эстетической точки зрения)
; Никто же не будет утверждать, что
; for (a = 10; a > 0; a-) более привычно, чем for (a = 0; a < 10; a++)?

pop     esi
ret     ret
main    endp

```

А что скажет нам товарищ Borland C++ 5.0? Компилируем и смотрим:

Листинг 195

```

_main proc near                                ; DATA XREF: DATA:00407044o
    push     ebp
    mov      ebp, esp
    ; Открываем кадр стека

    push     ebx
    ; Сохраняем EBX в стеке

    xor      ebx, ebx
    ; Присваиваем регистровой переменной EBX значение 0

loc_401082:                                    ; CODE XREF: _main+15j
                                                ; 00000000000000000000000000000000
    ; Начало цикла

    push     offset a0operatorCiklaF ; format
    call     _printf
    pop      ecx
    ; Начинаем цикл с выполнения его тела.
    ; OK, Borland понял, что цикл выполняется по крайней мере раз

    inc      ebx
    ; Увеличиваем параметр цикла

    cmp      ebx, 0Ah
    ; Сравниваем EBX со значением 0xA

    jl       short loc_401082
    ; Переход в начало цикла, пока EBX < 0xA

    xor      eax, eax
    pop      ebx
    pop      ebp
    ret

```

```
_main      endp
```

Видно, что Borland C++ 5.0 не дотягивает до Microsoft Visual C++ 6.0, понять, что цикл выполняется один раз, он понял, а вот реверс счетчика ума уже не хватило. Аналогичным образом поступает и большинство других компиляторов, в частности WATCOM C.

Теперь настала очередь циклов с условием в середине или циклов, завершаемых вручную оператором break. Рассмотрим следующий пример:

Листинг 196. Демонстрация идентификации break

```
#include <stdio.h>

main()
{
    int a=0;
    while(1)
    {
        printf("1й оператор\n");
        if (++a>10) break;
        printf("2й оператор\n");
    }

    do
    {
        printf("1й оператор\n");
        if (-a<0) break;
        printf("2й оператор\n");
    }while(1);
}
```

Результат компиляции Microsoft Visual C++ 6.0 с настройками по умолчанию должен выглядеть так:

Листинг 197

```
main      proc near      ; CODE XREF: start+AFp
var_a     = dword ptr -4

    push    ebp
    mov     ebp, esp
    ; Открываем кадр стека

    push    ecx
    ; Резервируем место для локальной переменной

    mov     [ebp+var_a], 0
    ; Присваиваем переменной var_a значение 0x0

loc_40100B:                ; CODE XREF: main+3Fj
                           ; 0000000000000000
    ; Перекрестная ссылка, направленная вниз, - цикл

    mov     eax, 1
    test    eax, eax
```



```

jz      short loc_401041
; Смотрите! Когда optimize disabled, - компилятор транслирует безусловный
; цикл "слишком буквально", так как присваивает EAX значение 1 (TRUE)
; и затем педантично проверяет ее на равенство нулю.
; Если в кои веки TRUE будет равно FALSE - произойдет выход из цикла.
; Словом, все эти три инструкции - глупый и бесполезный код цикла
; while (1)

push    offset a1iOperator ; "1-й оператор\n"
call    _printf
add     esp, 4
; printf("1-й оператор\n")

mov     ecx, [ebp+var_a]
; Загружаем в ECX значение переменной var_a

add     ecx, 1
; Увеличиваем ECX на единицу

mov     [ebp+var_a], ecx
; Обновляем var_a

cmp     [ebp+var_a], 0Ah
; Сравниваем var_a со значением 0xA

jle     short loc_401032
; Переход, если var_a <= 0xA.
; Но куда этот переход? Во-первых, переход направлен вниз, т. е. это уже
; не переход к началу цикла, следовательно, и условие не условие цикла, а
; результат компиляции конструкции IF - THEN.
; Второе - переход прыгает на первую команду, следующую за безусловным
; jump loc_401041, передающим управление инструкции, следующей
; за командной jmp short loc_401075 - безусловного перехода, направленного
; вверх - в начало цикла
; Следовательно, jmp short loc_401041 осуществляет выход из цикла, а
; jle short loc_401032 продолжает его выполнение

jmp     short loc_401041
; ОК, это переход на завершение цикла. А кто у нас завершает цикл?
; Ну конечно же break! Следовательно, окончательная декомпиляция выглядит так:
; if (++var_a > 0xA) break
; Мы инвертировали <= в >, так как JLE передает управление на код продолжения
; цикла, а ветка THEN в нашем случае - на break

loc_401032:                                ; CODE XREF: main+2Ej
                                           ; 0000000000000000
; Перекрестная ссылка направлена вверх, следовательно, это не начало цикла

push    offset a2iOperator ; "2-й оператор\n"
call    _printf
add     esp, 4
; printf("2-й оператор\n")

jmp     short loc_40100B
; Прыжок в начало цикла. Вот мы и добрались до конца цикла.
; Восстанавливаем исходный код:
; while(1)

```

```

; {
; printf("1-й оператор\n");
; if (++var_a > 0xA) break;
; printf("2-й оператор\n");
; }
;

loc_401041:                                ; CODE XREF: main+12j main+30j ...
; .....
; Перекрестная ссылка, направленная вниз, говорит, что это начало цикла

push    offset a1i0perator_0 ; "1-й оператор\n"
call    _printf
add     esp, 4
; printf("1-й оператор\n")

mov     edx, [ebp+var_a]
sub     edx, 1
mov     [ebp+var_a], edx
; -var_a

cmp     [ebp+var_a], 0
; Сравниваем var_a со значением 0x0

jge     short loc_40105F
; Переход вниз, если var_a >= 0.
; Смотрите, оператор break цикла do ничем не отличается от break цикла while!
; Поэтому не будем разглагольствовать, а сразу его декомпилируем!
; if (var_a < 0) ...

jmp     short loc_401075
; ...break

loc_40105F:                                ; CODE XREF: main+5Bj
push    offset a2i0perator_0 ; "2-й оператор\n"
call    _printf
add     esp, 4
; printf("2й оператор\n")

mov     eax, 1
test    eax, eax
jnz     short loc_401041
; А это - проверка продолжения цикла

loc_401075:                                ; CODE XREF: main+5Dj
mov     esp, ebp
pop     ebp
; Закрываем кадр стека

retn

main                                         endp

```

Что ж, оператор break в обоих циклах выглядит одинаково и элементарно распознается (правда, не с первого взгляда, но при отслеживании нескольких переходов — да). А вот с бесконечными циклами не оптимизирующий компилятор

подкачал, транслировав их в код, проверяющий условие, истинность (не истинность) которого очевидна. А как поведет себя оптимизирующий компилятор?

Давайте откомпилируем тот же самый пример компилятором Microsoft Visual C++ 6.0 с ключом /Ox и посмотрим:

Листинг 198

```
main          proc near          ; CODE XREF: start+AFp
    push      esi
    ; Сохраняем ESI в стеке

    xor       esi, esi
    ; Присваиваем ESI значение 0
    ; var_ESI = 0;

loc_401003:    ; CODE XREF: main+23j
    ;
    ; Перекрестная ссылка, направленная вперед.
    ; Это начало цикла

    push      offset a1i0perator ; "1-й оператор\n"
    call      _printf
    add       esp, 4
    ; printf("1-й оператор\n")
    ;
    ; Ага! Проверки на дорогах нет, значит, это цикл с постусловием
    ; (или условием в середине)

    inc       esi
    ; ++var_ESI

    cmp       esi, 0Ah
    ; Сравниваем var_ESI со значением 0xA

    jg        short loc_401025
    ; Выход из цикла, если var_ESI > 0xA.
    ; Поскольку данная команда не последняя в теле цикла,
    ; это цикл с условием в середине
    ; if (var_ESI > 0xA) break

    push      offset a2i0perator ; "2-й оператор\n"
    call      _printf
    add       esp, 4
    ; printf("2-й оператор\n")

    jmp       short loc_401003
    ; Безусловный переход в начало цикла.
    ; Как видно, оптимизирующий компилятор выкинул никому не нужную проверку
    ; условия, упростив код и облегчив его понимание:
    ; Итак:
    ; var_ESI = 0
    ; for (;;) ← вырожденный for представляет собой бесконечный цикл
    ; {
    ; printf("1-й оператор\n");
    ; ++var_ESI;
    ; if (var_ESI > 0xA) break;
```

```

        ; printf("2-й оператор\n");
        ; }

loc_401025:                                ; CODE XREF: main+14j
                                           ; 00000000000000000000000000000000

        ; Это не начало цикла!

        push    offset a1i0perator_0 ; "1-й оператор\n"
        call    _printf
        add     esp, 4
        ; printf("1-й оператор\n")
        ; Хм, как же это не начало цикла?! Очень похоже!

        dec     esi
        ; -var_ESI

        js      short loc_401050
        ; Выход из цикла, если var_ESI < 0

        inc     esi
        ; Увеличиваем var_ESI на единицу.
        ; М-м-м... (задумчиво)...

loc_401036:                                ; CODE XREF: main+4Ej
                                           ; 00000000000000000000000000000000

        ; А вот это начало цикла!

        push    offset a2i0perator_0 ; "2-й оператор\n"
        call    _printf
        ; printf("2-й оператор\n")
        ; Только странно, что начало цикла начинается с его, с позволения сказать,
        ; середины...

        push    offset a1i0perator_0 ; "1-й оператор\n"
        call    _printf
        add     esp, 8
        ; printf("1-й оператор\n")
        ;
        ; ???!!! Что за чудеса творятся? Во-первых, вызов первого оператора второго
        ; цикла уже встречался ранее, во-вторых, не может же следом за серединой цикла
        ; следовать его начало?!

        dec     esi
        ; -var_ESI

        jnz     short loc_401036
        ; Продолжение цикла, пока var_ESI != 0

loc_401050:                                ; CODE XREF: main+33j
        ; Конец цикла.
        ; Да... тут есть над чем подумать!
        ; Компилятор нормально "перевалил" первую строку цикла
        ; printf("1-й оператор\n"),
        ; а затем "напоролся" на ветвление:
        ; if (-a<0) break.
        ; Хитрые парни из Microsoft знают, что для суперконвейерных процессоров
        ; (коими и являются чипы Pentium) ветвления все равно, что чертополох для

```

; Тиггеров. Кстати, Си-компиляторы под процессоры серии CONVEX вообще
 ; отказываются компилировать циклы с ветвлениями, истощенно понося
 ; умственные способности программистов. А вы еще IBM PC ругаете ;-)
 ; Вот и приходится компилятору исправлять ляпы программиста, что он делать
 ; в принципе не обязан, но за что ему большое человеческое спасибо!
 ; Компилятор как бы "прокручивает" цикл, "слепляя" вызовы функций printf
 ; и вынося ветвления в конец
 ; Образно исполняемый код можно представить трассой, а процессор - гонщиком.
 ; Чем длиннее участок дороги без поворотов, тем быстрее его проскочит гонщик!
 ; Выносить условие из середины цикла в его конец компилятор вполне правомерен,
 ; ведь переменная, относительно которой выполняется ветвление,
 ; не модифицируется ни функцией printf, ни какой другой
 ; Поэтому не все ли равно, где ее проверять? Конечно же не все равно!!!
 ; К моменту, когда условие ($-a < 10$) становится истинно, успевает выполниться
 ; первый printf, а вот второй уже не получает управления
 ; Вот для этого-то компилятор и поместил код проверки условия следом за
 ; первым вызовом первой функции printf, а затем изменил порядок вызова
 ; printf в теле цикла. Это привело к тому, что на момент выхода из цикла
 ; по условию первый printf выполняется на один раз больше, чем второй
 ; (так как он встречается дважды).
 ; Остается разобраться с увеличением var_ESI. Что бы это значило?
 ; Давайте рассуждать от противного: что произойдет, если выкинуть
 ; команду INC ESI? Поскольку счетчик цикла при первой итерации цикла
 ; декрементируется дважды, возникнет недостача и цикл выполнится на раз
 ; короче. Чтобы этого не произошло, var_ESI искусственно увеличивается
 ; на единицу.
 ; Ой, и не просто во всей этой головоломке разобраться, а представить,
 ; насколько сложно реализовать компилятор, умеющий проделывать такие фокусы!
 ; А еще кто-то ругает автоматическую оптимизацию. Да уж! Конечно, руками-то
 ; можно и круче оптимизировать (особенно понимая смысл кода), но ведь эдак
 ; и мозги можно будет вывихнуть! А компилятор, даже будучи стиснут со всех
 ; сторон кривым кодом программиста, за доли секунды успевает его довольно
 ; прилично окультурить

```

pop     esi
retn
main    endp

```

Компиляторы Borland C++ и WATCOM при трансляции бесконечных циклов заменяют код проверки условия продолжения цикла на безусловный переход, но вот, увы, оптимизировать ветвления, вынося их в конец цикла так, как это делает Microsoft Visual C++ 6.0, они не умеют...

Теперь, после break, рассмотрим, как компиляторы транслируют его «астральный антипод», — оператор continue. Рассмотрим следующий пример:

Листинг 199. Демонстрация идентификации continue

```

#include <stdio.h>

main()
{
    int a=0;
    while (a++<10)
    {

```

```

        if (a == 2) continue;
        printf("%x\n",a);
    }

do
{
    if (a == 2) continue;
    printf("%x\n",a);
} while (-a>0);
}

```

Результат его компиляции компилятором Microsoft Visual C++ 6.0 с настройками по умолчанию будет выглядеть так:

Листинг 200

```

main          proc near          ; CODE XREF: start+AFp
var_a          = dword ptr -4

    push      ebp
    mov       ebp, esp
    ; Открываем кадр стека

    push      ecx
    ; Резервируем место для локальной переменной

    mov       [ebp+var_a], 0
    ; Присваиваем локальной переменной var_a значение 0

loc_40100B:    ; CODE XREF: main+22j main+35j
               ; ~~~~~
    ; Две перекрестные ссылки, направленные вперед, говорят о том, что это либо
    ; начало двух циклов (один из которых вложенный), либо переход в начало
    ; цикла оператором continue

    mov       eax, [ebp+var_a]
    ; Загружаем в EAX значение var_a

    mov       ecx, [ebp+var_a]
    ; Загружаем в ECX значение var_a

    add       ecx, 1
    ; Увеличиваем ECX на единицу

    mov       [ebp+var_a], ecx
    ; Обновляем переменную var_a

    cmp       eax, 0Ah
    ; Сравниваем значение переменной var_a до увеличения с числом 0xA

    jge       short loc_401037
    ; Выход из цикла (переход на команду, следующую за инструкцией, направленной
    ; вверх - в начало цикла), если var_a >= 0xA

    cmp       [ebp+var_a], 2
    ; Сравниваем var_a со значением 0x2

    jnz       short loc_401024

```

; Если `var_a != 2`, то прыжок на команду, следующую за инструкцией
 ; безусловного перехода, направленной вверх - в начало цикла.
 ; Очень похоже на условие выхода из цикла, но не будем спешить с выводами!
 ; Вспомним, в начале цикла нам встретились две перекрестные ссылки.
 ; Безусловный переход `jmp short loc_40100B` как раз образует одну из них.
 ; А кто "отвечает" за другую?
 ; Чтобы ответить на этот вопрос, необходимо проанализировать остальной код цикла

```
jmp     short loc_40100B
```

; Безусловный переход, направленный в начало цикла, это либо конец цикла,
 ; либо `continue`.
 ; Предположим, что это конец цикла. Тогда что же представляет собой
 ; `jge short loc_401037`? Предусловие выхода из цикла? Не похоже, в таком
 ; случае оно прыгало бы гораздо "ближе" - на метку `loc_401024`.
 ; А может, `jge short loc_401037` предусловие одного цикла, а
 ; `jnz short loc_401024` - постусловие другого, вложенного в него?
 ; Вполне возможно, но маловероятно - в этом случае постусловие представляло бы
 ; собой условие продолжения, а не завершения цикла.
 ; Поэтому с некоторой долей неуверенности мы можем принять конструкцию
 ; `CMP var_a, 2 \ JNZ loc_401024 \ JMP loc_40100B` за `if (a==2) continue`

```
loc_401024:                                ; CODE XREF: main+20j
```

```
mov     edx, [ebp+var_a]
push    edx
push    offset asc_406030 ; "%x\n"
call    _printf
add     esp, 8
; printf("%x\n", var_a)
```

```
jmp     short loc_40100B
```

; А вот это явно конец цикла, так как `jmp short loc_40100B` - самая
 ; последняя ссылка на начало цикла.
 ; Итак, подытожим, что мы имеем.
 ; Условие, расположенное в начале цикла, крутит этот цикл до тех пор, пока
 ; `var_a < 0xA`, причем инкремент параметра цикла происходит до его сравнения.
 ; Затем следует еще одно условие, возвращающее управление в начало цикла, если
 ; `var_a == 2`. Строй замыкает оператор цикла `printf` и безусловный переход в его
 ; начало. То есть

```
;
; Начало цикла:
; Инкремент переменной var_a
; условие "далекого" выхода
; условие "ближнего" продолжения
; тело цикла
; безусловный переход в начало
; конец цикла
;
; Условие "ближнего" продолжения не может быть концом цикла, так как тогда условию
; "далекого" выхода пришлось выйти аж из надлежащего цикла, на что ни break,
; ни другие операторы не способны. Таким образом, условие "ближнего" продолжения
; может быть только оператором continue и на языке Си всю эту конструкцию
; будет выглядеть так:
; while(a++<10)           // <- инкремент var_a и условие далекого выхода
; {
```

```

        ; if (a == 2) continue; // <- условие "ближнего" продолжения
        ; printf("%x\n", var_a); // <- тело цикла
        ; } // <- безусловный переход на начало цикла

loc_401037:                                ; CODE XREF: main+1Aj main+5Dj
                                           ; ~~~~~~

        ; Начало цикла

        cmp     [ebp+var_a], 2
        ; Сравняем переменную var_a со значением 0x2

        jnz     short loc_40103F
        ; Если var_a != 2, то продолжение цикла

        jmp     short loc_401050.
        ; Переход к коду проверки условия продолжения цикла.
        ; Это бесспорно continue, и вся конструкция выглядит так:
        ; if (a==2) continue;

loc_40103F:                                ; CODE XREF: main+3Bj
        mov     eax, [ebp+var_a]
        push    eax
        push    offset asc_406034 ; "%x\n"
        call    _printf
        add     esp, 8
        ; printf("%x\n", var_a);

loc_401050:                                ; CODE XREF: main+3Dj
        mov     ecx, [ebp+var_a]
        sub     ecx, 1
        mov     [ebp+var_a], ecx
        ; -var_a;

        cmp     [ebp+var_a], 0
        ; Сравнение var_a с нулем

        jg      short loc_401037
        ; Пока var_a > 0, продолжать цикл.
        ; Похоже на постусловие, верно? Тогда
        ; do
        ; {
        ; if (a==2) continue;
        ; printf("%x\n", var_a);
        ; } while (-var_a > 0);
        ;
        mov     esp, ebp
        pop     ebp
        retn

main     endp

```

А теперь посмотрим, как повлияла оптимизация (/Ox) на вид циклов:

Листинг 201

```

main     proc near                                ; CODE XREF: start+AFp
        push    esi

```



```

        mov     esi, 1
loc_401006:                                ; CODE XREF: main+1Fj
                                           ; 000000000000000000000000
        ; Начало цикла

        cmp     esi, 2
        jz      short loc_401019
        ; Переход на loc_401019, если ESI == 2

        push    esi
        push    offset asc_406030 ; "%x\n"
        call    _printf
        add     esp, 8
        ; printf("%x\n", ESI)
        ; Примечание: эта ветка выполняется, только если ESI != 2.
        ; Следовательно, ее можно изобразить так:
        ; if (ESI != 2) printf("%x\n", ESI)

loc_401019:                                ; CODE XREF: main+9j
        mov     eax, esi
        inc     esi
        ; ESI++;

        cmp     eax, 0Ah
        jl      short loc_401006
        ; Продолжение цикла пока (ESI++ < 0xA)
        ; Итого
        ; do
        ; {
        ; if (ESI != 2) printf("%x\n", ESI);
        ; } while (ESI++ < 0xA)
        ;
        ; А что, выглядит вполне читабельно, не правда ли? Ничуть не хуже, чем
        ; if (ESI == 2) continue
        ;

loc_401021:                                ; CODE XREF: main+37j
                                           ; 00000000
        ; Начало цикла

        cmp     esi, 2
        jz      short loc_401034
        ; Переход на loc_401034, если ESI == 2

        push    esi
        push    offset asc_406034 ; "%x\n"
        call    _printf
        add     esp, 8
        ; printf("%x\n", ESI);
        ; Примечание: эта ветка выполняется, лишь когда ESI != 2.

loc_401034:                                ; CODE XREF: main+24j
        dec     esi
        ; -ESI

        test    esi, esi

```

```

    jg      short loc_401021
    ; Условие продолжение цикла - крутить пока ESI > 0
    ; Итого
    ; do
    ; {
    ; if (ESI != 2)
    ; {
    ; printf("%x\n", ESI);
    ; }
    ; } while (-ESI > 0)
    ;
    pop     esi
    retn

main      endp

```

Остальные компиляторы сгенерируют приблизительно такой же код. Общим для всех случаев будет то, что на циклах с предусловием оператор `continue` практически неотличим от вложенного цикла, а на циклах с постусловием `continue` эквивалентен элементарному ветвлению.

Наконец, настала очередь циклов `for`, вращающих несколько счетчиков одновременно. Рассмотрим следующий пример:

Листинг 202. Демонстрация идентификации циклов `for` с несколькими счетчиками

```

main()
{
    int a; int b;
    for (a = 1, b = 10; a < 10, b > 1; a++, b --)
        printf("%x %x\n", a, b);
}

```

Результат его компиляции компилятором Microsoft Visual C++ 6.0 должен выглядеть так:

Листинг 203

```

main      proc near                ; CODE XREF: start+AFp
var_b     = dword ptr -8
var_a     = dword ptr -4

    push    ebp
    mov     ebp, esp
    ; Открываем кадр стека

    sub     esp, 8
    ; Резервируем память для двух локальных переменных

    mov     [ebp+var_a], 1
    ; Присваиваем переменной var_a значение 0x1

    mov     [ebp+var_b], 0Ah
    ; Присваиваем переменной var_b значение 0xA

```

```

        jmp     short loc_401028
        ; Прыжок на код проверки условия выхода из цикла
        ; Это характерная черта не оптимизированных циклов for

loc_401016:                                ; CODE XREF: main+43j
                                           ; 00000000
        ; Перекрестная ссылка, направленная вниз, говорит о том, что это начало цикла.
        ; А выше мы уже выяснили, что тип цикла - for.

        mov     eax, [ebp+var_a]
        add     eax, 1
        mov     [ebp+var_a], eax
        ; var_a++

        mov     ecx, [ebp+var_b]
        sub     ecx, 1
        mov     [ebp+var_b], ecx
        ; var_b-

loc_401028:                                ; CODE XREF: main+14j
        cmp     [ebp+var_b], 1
        jle     short loc_401045
        ; Выход из цикла, если var_b <= 0x1.
        ; Обратите внимание, выполняется проверка лишь одного (второго слева) счетчика!
        ; Выражение (a1, a2, a3,...,an) компилятор считает бессмысленным и берет лишь an,
        ; молчаливо отбрасывая все остальное
        ; (из известных автору компиляторов на это ругается один WATCOM).
        ; В данном случае проверяется лишь условие (b > 1), а (a < 10) игнорируется!!!

        mov     edx, [ebp+var_b]
        push    edx
        mov     eax, [ebp+var_a]
        push    eax
        push    offset aXX ; "%x %x\n"
        call    _printf
        add     esp, 0Ch
        ; printf("%x %x\n", var_a, var_b)

        jmp     short loc_401016
        ; Конец цикла.
        ; Итак, данный цикл можно представить как:
        ; while(1)
        ; {
        ;     var_a++;
        ;     var_b--;
        ;     if (var_b <= 0x1) break;
        ;     printf("%x %x\n", var_a, var_b)
        ; }
        ;
        ; Но по соображениям удобочитаемости имеет смысл скомпоновать этот код в for
        ; for (var_a=1, var_b=0xA; var_b>1; var_a++, var_b-) printf("%x %x\n", var_a, var_b);
        ;

loc_401045:                                ; CODE XREF: main+2Cj
        mov     esp, ebp

```

```
    pop     ebp
    ; Закрываем кадр стека

    retn

main      endp
```

Оптимизированный вариант программы рассматривать не будем, так как это не покажет нам ничего нового. Какой бы компилятор мы ни выбрали, выражения инициализации и модификации счетчиков будут обрабатываться вполне корректно в порядке их объявления в тексте программы, а вот множественные выражения продолжения цикла не умеет правильно обрабатывать ни один компилятор!

Идентификация математических операторов

...если вы обессилены, то не удивительно, что вся ваша жизнь — не развлечение. У вас... так много вычислений, расчетов, которые необходимо сделать в вашей жизни, что она просто не может быть развлечением.

Ошо. Пустая лодка.

Беседы по высказываниям Чжуана Цзы

Идентификация оператора «+». В общем случае оператор «+» транслируется либо в машинную инструкцию ADD, «перемалывающую» целочисленные операнды, либо в инструкцию FADDx, обрабатывающую вещественные значения. Оптимизирующие компиляторы могут заменять ADD xxx, 1 более компактной командой INC xxx, а конструкцию $c = a + b + \text{const}$ транслировать в машинную инструкцию LEA c, [a + b + const]. Такой трюк позволяет одним махом складывать несколько переменных, возвратив полученную сумму в любом регистре общего назначения, — необязательно в левом слагаемом, как это требует мнемоника команды ADD. Однако LEA не может быть непосредственно декомпилирована в оператор «+», поскольку она используется не только для оптимизированного сложения (что, в общем-то, является только побочным продуктом ее деятельности), но и по своему непосредственному назначению — вычислению эффективно-го смещения. (подробнее об этом см. разделы «Идентификация констант и смещений», «Идентификация типов»).

Рассмотрим следующий пример:

Листинг 204. Демонстрация оператора «+»

```
main()
{
    int a, b, c;
    c = a + b;
    printf("%x\n", c);
    c=c+1;
    printf("%x\n", c);
}
```

```
}
```

Результат его компиляции компилятором Microsoft Visual C++ 6.0 с настройками по умолчанию должен выглядеть так:

Листинг 205

```
main          proc near          ; CODE XREF: start+AFp
var_c          = dword ptr -0Ch
var_b          = dword ptr -8
var_a          = dword ptr -4

    push      ebp
    mov      ebp, esp
    ; Открываем кадр стека

    sub      esp, 0Ch
    ; Резервируем память для локальных переменных

    mov      eax, [ebp+var_a]
    ; Загружаем в EAX значение переменной var_a

    add      eax, [ebp+var_b]
    ; Складываем EAX со значением переменной var_b и записываем результат в EAX

    mov      [ebp+var_c], eax
    ; Копируем сумму var_a и var_b в переменную var_c, следовательно,
    ; var_c = var_a + var_b

    mov      ecx, [ebp+var_c]
    push      ecx
    push      offset asc_406030 ; "%x\n"
    call     _printf
    add      esp, 8
    ; printf("%x\n", var_c)

    mov      edx, [ebp+var_c]
    ; Загружаем в EDX значение переменной var_c

    add      edx, 1
    ; Складываем EDX со значением 0x1, записывая результат в EDX

    mov      [ebp+var_c], edx
    ; Обновляем var_c
    ; var_c = var_c + 1

    mov      eax, [ebp+var_c]
    push      eax
    push      offset asc_406034 ; "%x\n"
    call     _printf
    add      esp, 8
    ; printf("%\n",var_c)

    mov      esp, ebp
    pop      ebp
    ; Закрываем кадр стека
```

```

    retn
main    endp

```

А теперь посмотрим, как будет выглядеть тот же самый пример, скомпилированный с ключом /Ox (максимальная оптимизация):

Листинг 206

```

main    proc near                ; CODE XREF: start+AFp
    push    ecx
    ; Резервируем место для одной локальной переменной
    ; (компилятор посчитал, что три переменные можно ужать в одну, и это действительно так)

    mov     eax, [esp+0]
    ; Загружаем в EAX значение переменной var_a

    mov     ecx, [esp+0]
    ; Загружаем в EAX значение переменной var_b
    ; (так как переменная не инициализирована, загружать можно откуда угодно)

    push    esi
    ; Сохраняем регистр ESI в стеке

    lea     esi, [ecx+eax]
    ; Используем LEA для быстрого сложения ECX и EAX с последующей записью суммы
    ; в регистр ESI.
    ; "Быстрое сложение" следует понимать не в смысле, что команда LEA выполняется
    ; быстрее, чем ADD, - количество тактов той и другой одинаково, - но LEA
    ; позволяет избавиться от создания временной переменной для сохранения
    ; промежуточного результата сложения, сразу направляя результат в ESI.
    ; Таким образом, эта команда декомпилируется как
    ; reg_ESI = var_a + var_b

    push    esi
    push    offset asc_406030 ; "%x\n"
    call    _printf
    ; printf("%x\n", reg_ESI)

    inc     esi
    ; Увеличиваем ESI на единицу
    ; reg_ESI = reg_ESI + 1

    push    esi
    push    offset asc_406034 ; "%x\n"
    call    _printf
    add     esp, 10h
    ; printf("%x\n", reg_ESI)

    pop     esi
    pop     ecx
    retn
main    endp

```

Остальные компиляторы (Borland C++, WATCOM C) генерируют приблизительно идентичный код, поэтому приводить результаты бессмысленно — никаких новых «изюминок» они в себе не несут.

Идентификация оператора «-». В общем случае оператор «-» транслируется либо в машинную инструкцию SUB (если операнды — целочисленные значения), либо в инструкцию FSUBx (если операнды — вещественные значения). Оптимизирующие компиляторы могут заменять SUB xxx, 1 более компактной командой DEC xxx, а конструкцию SUB a, const транслировать в ADD a, -const, которая ничуть не компактнее и нисколько не быстрее (и та и другая укладывается в один так), однако хозяин (компилятор) — барин. Покажем это на следующем примере:

Листинг 207. Демонстрация идентификации оператора «-»

```
main()
{
    int a,b,c;

    c = a - b;
    printf("%x\n",c);

    c = c - 10;
    printf("%x\n",c);
}
```

Не оптимизированный вариант будет выглядеть приблизительно так:

Листинг 208

```
main                proc near                ; CODE XREF: start+AFp
var_c                = dword ptr -0Ch
var_b                = dword ptr -8
var_a                = dword ptr -4

    push    ebp
    mov     ebp, esp
    ; Открываем кадр стека

    sub     esp, 0Ch
    ; Резервируем память под локальные переменные

    mov     eax, [ebp+var_a]
    ; Загружаем в EAX значение переменной var_a

    sub     eax, [ebp+var_b]
    ; Вычитаем из var_a значение переменной var_b, записывая результат в EAX.

    mov     [ebp+var_c], eax
    ; Записываем в var_c разность var_a и var_b
    ; var_c = var_a - var_b

    mov     ecx, [ebp+var_c]
    push    ecx
    push    offset asc_406030 ; "%x\n"
    call    _printf
    add     esp, 8
    ; printf("%x\n", var_c)

    mov     edx, [ebp+var_c]
    ; Загружаем в EDX значение переменной var_c
```

```

    sub     edx, 0Ah
    ; Выводим из var_c значение 0xA, записывая результат в EDX:

    mov     [ebp+var_c], edx
    ; Обновляем var_c
    ; var_c = var_c - 0xA

    mov     eax, [ebp+var_c]
    push    eax
    push    offset asc_406034 ; "%x\n"
    call    _printf
    add     esp, 8
    ; printf("%x\n", var_c)

    mov     esp, ebp
    pop     ebp
    ; Закрываем кадр стека
    retn

main      endp

```

А теперь рассмотрим оптимизированный вариант того же примера:

Листинг 209

```

main      proc near                ; CODE XREF: start+AFp
    push    ecx
    ; Резервируем место для локальной переменной var_a

    mov     eax, [esp+var_a]
    ; Загружаем в EAX значение локальной переменной var_a

    push    esi
    ; Резервируем место для локальной переменной var_b

    mov     esi, [esp+var_b]
    ; Загружаем в ESI значение переменной var_b

    sub     esi, eax
    ; Выводим из var_a значение var_b, записывая результат в ESI

    push    esi
    push    offset asc_406030 ; "%x\n"
    call    _printf
    ; printf("%x\n", var_a - var_b)

    add     esi, 0FFFFFFFh
    ; Добавляем к ESI (разности var_a и var_b) значение 0FFFFFFFh
    ; Поскольку 0FFFFFFFh == -0xA, данная строка кода выглядит так:
    ; ESI = (var_a - var_b) + (- 0xA) = (var_a - var_b) - 0xA

    push    esi
    push    offset asc_406034 ; "%x\n"
    call    _printf
    add     esp, 10h
    ; printf("%x\n", var_a - var_b - 0xA)

    pop     esi
    pop     ecx

```



```

; Закрываем кадр стека
    retn
main    endp

```

Остальные компиляторы (Borland, WATCOM) генерируют практически идентичный код, поэтому здесь не рассматриваются.

Идентификация оператора «/». В общем случае оператор «/» транслируется либо в машинную инструкцию DIV (беззнаковое целочисленное деление), либо в IDIV (целочисленное деление со знаком), либо в FDIVx (вещественное деление). Если делитель кратен степени двойки, то DIV заменяется на более быстройдействующую инструкцию битового сдвига вправо SHR a, N, где a — делимое, N — показатель степени с основанием два.

Несколько сложнее происходит быстрое деление знаковых чисел. Совершенно недостаточно выполнить арифметический сдвиг вправо (команда арифметического сдвига вправо SAR заполняет старшие биты с учетом знака числа), ведь если модуль делимого меньше модуля делителя, то арифметический сдвиг вправо сбросит все значащие биты в «битовую корзину», в результате чего получится 0xFFFFFFFF, т. е. -1, в то время как правильный ответ — ноль. Вообще же деление знаковых чисел арифметическим сдвигом вправо дает округление в *большую* сторону, что совсем не входит в наши планы. Для округления знаковых чисел в меньшую сторону необходимо перед выполнением сдвига добавить к делимому число $2^N - 1$, где N — количество битов, на которые сдвигается число при делении. Легко видеть, что это приводит к увеличению всех сдвигаемых битов на единицу и переносу в старший разряд, если хотя бы один из них не равен нулю.

Следует отметить: деление — очень медленная операция, гораздо более медленная, чем умножение (выполнение DIV может занять свыше 40 тактов, в то время как MUL обычно укладывается в 4), поэтому продвинутые оптимизирующие компиляторы заменяют деление умножением. Существует множество формул подобных преобразований, вот, например, она (самая популярная из них):

$$\frac{a}{b} = \frac{2^N}{b} + \frac{a}{2^N},$$

где N — разрядность числа. Выходит, грань между умножением и делением очень тонкая, а их идентификация довольно сложная. Рассмотрим следующий пример:

Листинг 210. Идентификация оператора «/»

```

main()
{
    int a;
    printf("%x %x\n", a / 32, a / 10);
}

```

Результат его компиляции компилятором Microsoft Visual C++ с настройками по умолчанию должен выглядеть так:

Листинг 211

```

main                proc near                ; CODE XREF: start+AFp

```

```

var_a      = dword ptr -4

    push    ebp
    mov     ebp, esp
    ; Открываем кадр стека

    push    ecx
    ; Резервируем память для локальной переменной

    mov     eax, [ebp+var_a]
    ; Копируем в EAX значение переменной var_a

    cdq
    ; Расширяем EAX до четверного слова EDX:EAX

    mov     ecx, 0Ah
    ; Заносим в ECX значение 0xA

    idiv    ecx
    ; Делим (учитывая знак) EDX:EAX на 0xA, занося частное в EAX
    ; EAX = var_a / 0xA

    push    eax
    ; Передаем результат вычислений функции printf

    mov     eax, [ebp+var_a]
    ; Загружаем в EAX значение var_a

    cdq
    ; Расширяем EAX до четверного слова EDX:EAX

    and     edx, 1Fh
    ; Выделяем пять младших бит EDX

    add     eax, edx
    ; Складываем знак числа для выполнения
    ; округления отрицательных значений
    ; в меньшую сторону

    sar     eax, 5
    ; Арифметический сдвиг вправо на 5 позиций
    ; эквивалентен делению числа на 25 = 32.
    ; Таким образом, последние четыре инструкции расшифровываются как:
    ; EAX = var_a / 32
    ; Обратите внимание,
    ; даже при выключенном режиме оптимизации компилятор
    ; оптимизировал деление

    push    eax
    push    offset aXX ; "%x %x\n"
    call    _printf
    add     esp, 0Ch
    ; printf("%x %x\n", var_a / 0xA, var_a / 32)

mov     esp, ebp
    pop     ebp
    ; Закрываем кадр стека

    retn

```

```
main                endp
```

А теперь, засучив рукава и глотнув пустырника (или валерьянки), рассмотрим оптимизированный вариант того же примера:

Листинг 212

```
main                proc near                ; CODE XREF: start+AFp
    push    ecx
    ; Резервируем память для локальной переменной var_a

    mov     ecx, [esp+var_a]
    ; Загружаем в ECX значение переменной var_a

    mov     eax, 66666667h
    ; Так, что это за зверское число?!
    ; В исходном коде ничего подобного и близко не было!

    imul    ecx
    ; Умножаем это зверское число на переменную var_a.
    ; Обратите внимание, именно умножаем, а не делим.
    ; Однако притворимся на время, что у нас нет исходного кода примера, потому
    ; ничего странного в операции умножения мы не видим

    sar     edx, 2
    ; Выполняем арифметический сдвиг всех битов EDX на две позиции вправо, что
    ; в первом приближении эквивалентно его делению на 4.
    ; Однако ведь в EDX находится старшее двойное слово результата умножения!
    ; Поэтому три предыдущих команды фактически расшифровываются так:
    ;  $EDX = (66666667h * var\_a) \gg (32 + 2) = (66666667h * var\_a) / 0x400000000$ 
    ;
    ; Понюхайте эту строчку, не пахнет ли паленым? Как так не пахнет?! Смотрите:
    ;  $(66666667h * var\_a) / 0x400000000 = var\_a * 66666667h / 0x400000000 =$ 
    ;  $= var\_a * 0,10000000003492459654808044433594$ 
    ; Заменяя по всем правилам математики умножение на деление и одновременно
    ; выполняя округление до меньшего целого, получаем:
    ;  $var\_a * 0,1000000000 = var\_a * (1/0,1000000000) = var\_a/10$ 
    ;
    ; Согласитесь, от такого преобразования код стал намного понятнее!
    ; Как можно распознать такую ситуацию в чужой программе, исходный текст которой
    ; неизвестен? Да очень просто: если встречается умножение, а следом за ним
    ; сдвиг вправо, обозначающий деление, то каждый нормальный математик сочтет
    ; своим долгом сократить такую конструкцию по методике, показанной выше!

    mov     eax, edx
    ; Копируем полученное частное в EAX

    shr     eax, 1Fh
    ; Сдвигаем на 31 позицию вправо

    add     edx, eax
    ; Складываем:  $EDX = EDX + (EDX \gg 31)$ 
    ; Что бы это значило? Нетрудно понять, что после сдвига EDX на 31 бит вправо
    ; в нем останется лишь знаковый бит числа.
    ; Тогда, если число отрицательно, мы добавляем к результату деления один,
    ; округляя его в меньшую сторону. Таким образом, весь этот хитрый код
```

```

; обозначает не что иное, как тривиальную операцию знакового деления:
; EDX = var_a / 10
; Не слишком ли много кода для одного лишь деления? Конечно, программа
; здорово "распухает", зато весь этот код выполняется всего лишь за 9 тактов,
; в то время как в не оптимизированном варианте аж за 28!
; /* Измерения проводились на процессоре CLERION с ядром P6, на других
; процессорах количество тактов может отличаться */
; То есть оптимизация дала более чем трехкратный выигрыш. Браво, Microsoft!

mov     eax, ecx
; Вспомним, что находится в ECX? 0x, уж эта наша дырявая память, более дырявая,
; чем дуршлаг без дна... Прокручиваем экран дизассемблера вверх. Ага, в ECX
; последний раз разгружалось значение переменной var_a

push     edx
; Передаем функции printf результат деления var_a на 10

cdq
; Расширяем EAX (var_a) до четверного слова EDX:EAX

and      edx, 1Fh
; Выбираем младшие 5 битов регистра EDX, содержащие знак var_a

add      eax, edx
; Округляем до меньшего

sar      eax, 5
; Арифметический сдвиг на 5 эквивалентен делению var_a на 32

push     eax
push     offset aXX ; "%x %x\n"
call     _printf
add      esp, 10h
; printf("%x %x\n", var_a / 10, var_a / 32)

retn
main     endp

```

Ну а другие компиляторы, насколько они продвинуты в плане оптимизации? Увы, ни Borland, ни WATCOM не умеют заменять деление более быстрым умножением для чисел, отличных от степени двойки. В подтверждение тому рассмотрим результат компиляции того же примера компилятором Borland C++:

Листинг 213

```

_main      proc near          ; DATA XREF: DATA:00407044o

    push    ebp
    mov     ebp, esp
    ; Открываем кадр стека

    push    ebx
    ; Сохраняем EBX

    mov     eax, ecx
    ; Копируем в EAX содержимое неинициализированной регистровой переменной ECX

    mov     ebx, 0Ah

```

```

; Заносим в EBX значение 0xA

cdq
; Расширяем EAX до четверного слова EDX:EAX

idiv    ebx
; Делим ECX на 0xA (долго делим - тактов 20, а то и больше)

push    eax
; Передаем полученное значение функции printf

test    ecx, ecx
jns     short loc_401092
; Если делимое не отрицательно, то переход на loc_401092

add     ecx, 1Fh
; Если делимое положительно, то добавляем к нему 0x1F для округления

loc_401092:                                ; CODE XREF: _main+11j
sar     ecx, 5
; Сдвигом на 5 позиций вправо делим число на 32

push    ecx
push    offset aXX ; "%x %x\n"
call    _printf
add     esp, 0Ch
; printf("%x %x\n", var_a / 10, var_a / 32)

xor     eax, eax
; Возвращаем ноль

pop     ebx
pop     ebp
; Закрываем кадр стека

ret     0
_main   endp

```

Идентификация оператора «%». Специальной инструкции для вычисления остатка в наборе команд микропроцессоров серии 80x86 нет, вместо этого остаток вместе с частным возвращается инструкциями деления DIV, IDIV и FDIVx (см. «Идентификация оператора «/»).

Если делитель представляет собой степень двойки ($2^N = b$), а делимое — беззнаковое число, то остаток будет равен N младшим битам делимого числа. Если же делимое — знаковое, необходимо установить все биты, кроме первых N , равными знаковому биту для сохранения знака числа. Причем, если N первых битов равно нулю, все биты результата должны быть сброшены независимо от значения знакового бита.

Таким образом, если делимое — беззнаковое число, то выражение $a \% 2^N$ транслируется в конструкцию AND a, N , в противном случае трансляция становится неоднозначна — компилятор может вставлять явную проверку на равенство нулю с ветвлением, а может использовать хитрые математические алгоритмы, самый популярный из которых выглядит так: DEC $x \setminus$ OR $x, -N \setminus$ INC x . Весь фокус в том, что если первые N битов числа x равны нулю, то все биты результата,

кроме старшего, знакового бита, будут гарантированно равны одному, а OR x , $-N$ принудительно установит в единицу и старший бит, т. е. получится значение, равное -1 . А INC -1 даст ноль! Напротив, если хотя бы один из N младших битов равен одному, заема из старших битов не происходит и INC x возвращает значению первоначальный результат.

Продвинутые оптимизирующие компиляторы могут путем сложных преобразований заменять деление на ряд других, более быстродействующих операций. К сожалению, алгоритмов для быстрого вычисления остатка для всех делителей не существует и делитель должен быть кратен $k \cdot 2^t$, где k и t — некоторые целые числа. Тогда остаток можно вычислить по следующей формуле:

$$a \% b = a \% k \cdot 2^t = a - \left(\frac{2^N}{k} + \frac{a}{2^N} \& - 2^k \right) \cdot k.$$

Да, эта формула очень сложна и идентификация оптимизированного оператора «%» может быть весьма и весьма непростой, особенно учитывая патологическую любовь оптимизаторов к изменению порядка команд.

Рассмотрим следующий пример:

Листинг 214. Идентификация оператора «%»

```
main()
{
    int a;
    printf("%x %x\n", a % 16, a % 10);
}
```

Результат его компиляции компилятором Microsoft Visual C++ с настройками по умолчанию должен выглядеть так:

Листинг 215

```
main                proc near                ; CODE XREF: start+AFp
var_4                = dword ptr -4
    push    ebp
    mov     ebp, esp
    ; Открываем кадр стека

    push    ecx
    ; Резервируем память для локальной переменной

    mov     eax, [ebp+var_a]
    ; Заносим в EAX значение переменной var_a

    cdq
    ; Расширяем EAX до четверного слова EDX:EAX

    mov     ecx, 0Ah
    ; Заносим в ECX значение 0xA

    idiv    ecx
    ; Делим EDX:EAX (var_a) на ECX (0xA)

    push    edx
    ; Передаем остаток от деления var_a на 0xA функции printf
```

```

mov     edx, [ebp+var_a]
; Заносим в EDX значение переменной var_a

and     edx, 8000000Fh
; "Вырезаем" знаковый бит и четыре младших бита числа,
; в четырех младших битах содержится остаток от деления EDX на 16

jns     short loc_401020
; Если число не отрицательно, то прыгаем на loc_401020

dec     edx
or      edx, 0FFFFFF0h
inc     edx
; Последовательность сия, как говорилось выше, характерна для быстрого
; расчета отставка знакового числа
; Следовательно, последние шесть инструкций расшифровываются как:
; EDX = var_a % 16

loc_401020:                                ; CODE XREF: main+19j
push    edx
push    offset aXX ; "%x %x\n"
call    _printf
add     esp, 0Ch
; printf("%x %x\n",var_a % 0xA, var_a % 16)

mov     esp, ebp
pop     ebp
; Закрываем кадр стека
retn

main                                         endp

```

Любопытно, что оптимизация не влияет на алгоритм вычисления остатка. Увы, ни Microsoft Visual C++, ни остальные известные мне компиляторы не умеют вычислять остаток умножением.

Идентификация оператора «*». В общем случае оператор «*» транслируется либо в машинную инструкцию MUL (беззнаковое целочисленное умножение), либо в IMUL (целочисленное умножение со знаком), либо в FMULx (вещественное умножение). Если один из множителей кратен степени двойки, то MUL (IMUL) обычно заменяется командой битового сдвига влево SHL или инструкцией LEA, способной умножать содержимое регистров на 2, 4 и 8. Обе последних команды выполняются за один такт, в то время как MUL требует в зависимости от модели процессора от двух до девяти тактов. К тому же LEA за тот же такт успевает сложить результат умножения с содержимым регистра общего назначения и/или константой в придачу. Это позволяет умножать на 3, 5 и 9, просто добавляя к умножаемому регистру его значение. Ну разве это не сказка? Правда, у LEA есть один недочет — она может вызывать остановку AGI, в конечном счете «съедающую» весь выигрыш в быстродействии на нет.

Рассмотрим следующий пример:

Листинг 216. Идентификация оператора «*»

```
main()
```

```
{
    int a;
    printf("%x %x %x\n", a * 16, a * 4 + 5, a * 13);
}
```

Результат его компиляции компилятором Microsoft Visual C++ с настройками по умолчанию должен выглядеть так:

Листинг 217

```
main          proc near          ; CODE XREF: start+AFp
var_a         = dword ptr -4

    push      ebp
    mov       ebp, esp
    ; Открываем кадр стека

    push      ecx
    ; Резервируем место для локальной переменной var_a

    mov       eax, [ebp+var_a]
    ; Загружаем в EAX значение переменной var_a

    imul      eax, 0Dh
    ; Умножаем var_a на 0xD, записывая результат в EAX

    push      eax
    ; Передаем функции printf произведение var_a * 0xD

    mov       ecx, [ebp+var_a]
    ; Загружаем в ECX значение var_a

    lea       edx, ds:5[ecx*4]
    ; Умножаем ECX на 4 и добавляем к полученному результату 5, записывая его в EDX.
    ; И все это выполняется за один такт!

    push      edx
    ; Передаем функции printf результат var_a * 4 + 5

    mov       eax, [ebp+var_a]
    ; Загружаем в EAX значение переменной var_a

    shl       eax, 4
    ; Умножаем var_a на 16.

    push      eax
    ; Передаем функции printf произведение var_a * 16

    push      offset aXXX ; "%x %x %x\n"
    call      _printf
    add       esp, 10h
    ; printf("%x %x %x\n", var_a * 16, var_a * 4 + 5, var_a * 0xD)

    mov       esp, ebp
    pop       ebp
    ; Закрываем кадр стека
    retn

main          endp
```


За вычетом вызова функции `printf` и загрузки переменной `var_a` из памяти на все про все требуется лишь **три** такта процессора. А что будет, если скомпилировать этот пример с ключиком `/Ox`? А будет вот что:

Листинг 218

```
main          proc near          ; CODE XREF: start+AFp
    push     ecx
    ; Выделяем память для локальной переменной var_a

    mov     eax, [esp+var_a]
    ; Загружаем в EAX значение переменной var_a

    lea     ecx, [eax+eax*2]
    ; ECX = var_a * 2 + var_a = var_a * 3

    lea     edx, [eax+ecx*4]
    ; EDX = (var_a * 3)* 4 + var_a = var_a * 13!
    ; Вот так компилятор ухитрился умножить var_a на 13,
    ; причем всего за один (!) такт. Да, обе инструкции LEA прекрасно спариваются
    ; на Pentium MMX и Pentium Pro!

    lea     ecx, ds:5[edx*4]
    ; ECX = EAX*4 + 5

    push     edx
    push     ecx
    ; Передаем функции printf var_a * 13 и var_a * 4 +5

    shl     eax, 4
    ; Умножаем var_a на 16

    push     eax
    push     offset aXXX ; "%x %x %x\n"
    call    _printf
    add     esp, 14h
    ; printf("%x %x %x\n", var_a * 16, var_a * 4 + 5, var_a * 13)
    retn

main          endp
```

Этот код, правда, все же не быстрее предыдущего, не оптимизированного, и укладывается в те же три такта, но в других случаях выигрыш может оказаться вполне ощутимым.

Другие компиляторы также используют LEA для быстрого умножения чисел. Вот, к примеру, Borland поступает так:

Листинг 219

```
_main        proc near          ; DATA XREF: DATA:00407044o
    lea     edx, [eax+eax*2]
    ; EDX = var_a*3

    mov     ecx, eax
    ; Загружаем в ECX неинициализированную регистровую переменную var_a

    shl     ecx, 2
```

```

; ECX = var_a * 4

push    ebp
; Сохраняем EBP

add     ecx, 5
; Добавляем к var_a * 4 значение 5.
; Borland не использует LEA для сложения. А жаль...

lea     edx, [eax+edx*4]
; EDX = var_a + (var_a *3) *4 = var_a * 13
; А вот в этом Borland и MS единодушны :-)

mov     ebp, esp
; Открываем кадр стека
; Да, да..., вот так посреди функции и открываем...
; Выше, кстати, "потерянная" команда push EBP

push    edx
; Передаем printf произведение var_a * 13

shl     eax, 4
; Умножаем ((var_a *4) + 5) на 16
; Что такое?! Да, это глюк компилятора, посчитавшего, раз переменная var_a
; неинициализирована, то ее можно и не загружать...

push    ecx
push    eax
push    offset aXXX ; "%x %x %x\n"
call    printf
add     esp, 10h
xor     eax, eax
pop     ebp
retn

_main   endp

```

Хотя визуально Borland генерирует более тупой код, его выполнение укладывается в те же три такта процессора. Другое дело WATCOM, показывающий удручающе отсталый результат на фоне двух предыдущих компиляторов:

Листинг 220

```

main    proc near
    push    ebx
    ; Сохраняем EBX в стеке

    mov     eax, ebx
    ; Загружаем в EAX значение неинициализированной регистровой переменной var_a

    shl     eax, 2
    ; EAX = var_a * 4

    sub     eax, ebx
    ; EAX = var_a * 4 - var_a = var_a * 3
    ; Вот он какой, WATCOM! Сначала умножает "с запасом", а потом лишнее отнимает!

    shl     eax, 2

```

```

; EAX = var_a * 3 * 4 = var_a * 12
add    eax, ebx
; EAX = var_a * 12 + var_a = var_a * 13
; Вот так, да? Четыре инструкции, в то время как "ненавистный" многим
; Microsoft Visual C++ вполне обходится и двумя!

push    eax
; Передаем printf значение var_a * 13

mov     eax, ebx
; Загружаем в EAX значение неинициализированной регистровой переменной var_a

shl     eax, 2
; EAX = var_a * 4

add     eax, 5
; EAX = var_a * 4 + 5
; Ага! Пользоваться LEA WATCOM тоже не умеет!

push    eax
; Передаем printf значение var_a * 4 + 5

shl     ebx, 4
; EBX = var_a * 16

push    ebx
; Передаем printf значение var_a * 16

push    offset aXXX ; "%x %x %x\n"
call    printf_
add     esp, 10h
; printf("%x %x %x\n", var_a * 16, var_a * 4 + 5, var_a*13)

pop     ebx

retn

main_    endp

```

В результате код, сгенерированный компилятором WATCOM, требует шести тактов, т. е. вдвое больше, чем у конкурентов.

Комплексные операторы. Язык Си\Си++ выгодно отличается от большинства своих конкурентов поддержкой комплексных операторов: $x=$ (где x — любой элементарный оператор), $++$ и $--$.

Комплексные операторы семейства $a x= b$ транслируются в $a = a x b$, и они идентифицируются так же, как и элементарные операторы (см. «*Элементарные операторы*»).

Операторы « $++$ » и « $--$ »: в префиксной форме они выражаются в тривиальные конструкции $a = a + 1$ и $a = a - 1$, не представляющие для нас никакого интереса, но вот постфиксная форма — дело другое.

Способы затруднения анализа программ

Устоявшиеся образцы могут быть путеводными, а могут привести в ловушку. Надо помнить, что даже узоры созвездий меняются.

Ф. Херберт. Дети Дюны

Три основных этапа взлома защитных механизмов — это **локализации кода защиты** в сотнях килобайтов (мегабайтов) кода приложения и **анализ алгоритма** ее работы. Последняя стадия — собственно сам **взлом**. Все этапы одинаково важны, если, например, не будет пройден второй из них — за взлом нечего и браться.

Можно классифицировать защиты по типу «этапа преткновения». Например, шифры и криптозащиты опираются на третий этап — алгоритм их работы обычно общедоступен, хорошо документирован и в общем случае известен хакеру, — но это не сильно облегчает взлом (разве что упрощает написание лобового переборщика). Механизмы регистрационных номеров, напротив, делают упор на засекречивании алгоритма генерации и затруднении его поиска и анализа в коде программы (еще бы, зная алгоритм, можно легко написать кейген).

Однако даже если защита построена с применением криптографических методов, скажем, шифрует тело критическиважный функций криптостойким методом по непомерно длинному ключу, она может быть «отвязана» от ключа, например, копированием дампа программы после расшифровки. Еще проще — распространять программу вместе с ключом (обычная тактика пиратов). Один из способов воспрепятствования такому беспределу — заложить в ключ зашифрованную привязку к компьютеру или проверять «чистоту» копии через Интернет (можно даже и втихомолку — скрыто от пользователя, хотя это считается дурным тоном). Но что помешает хакеру, владеющему лицензионной копией программы, расшифровать ее своим ключом и выкусить все-все проверки чего бы там ни было?

Таким образом, любой защите желательно уметь эффективно препятствовать своему обнаружению, анализу, попутно отравляя жизнь дизассемблеру и отладчику — основным инструментам взломщика. Без этого защита — не защита.

В эпоху царствования MS-DOS землей безраздельно владели программы реального режима, монопольно распоряжающиеся процессором, памятью и аппаратурой, в любой момент беспрепятственно переходящие в защищенный режим и возвращающиеся обратно. Отладчики в то время (еще хлипкие, немощные, нежизнеспособные) легко обманывались (срубались, завешивались) тривиальными

приемами программирования, активно используемыми защитами. Дизассемблеры тогда были очень глупыми и впадали в ступор от одного только вида зашифрованного или самомодифицирующегося кода. Словом, настоящий рай для разработчиков защит.

Сегодня все изменилось. Прежде всего, прикладной программе под Windows особо выпендриваться никто и не даст. С защищенным режимом теперь особо не разгонишься — используй прозаические непривилегированные инструкции, а о разных «тигростях» и не помышляй. Та же небольшая часть защитных приемов, что может функционировать даже в такой «юзеризированной» среде, наталкивается на сильно поумневшие отладчики и дизассемблеры.

Аппаратная поддержка отладки в процессорах 386+ в совокупности с виртуальным режимом работы, привилегированными инструкциями и виртуальной памятью позволяет создавать отладчики, которые практически не могут быть обнаружены прикладной программой, и уж тем более для нее невозможно получить над ними контроль.

Существуют и отладчики-эмуляторы, фактически настоящие виртуальные машины, самостоятельно исполняющие код, вместо того чтобы пустить его на «живой» процессор. При этом эмулятор всегда исполняется в режиме супервизора даже по отношению к отлаживаемому коду нулевого кольца. У защиты очень мало шансов обнаружить отладчик или помешать его работе (да и то, если эмулятор реализован с ошибками).

Появились и интерактивные дизассемблеры (та же IDA), которые в силу тесного взаимодействия с пользователем (в смысле хакером) могут обходить любые мыслимые и немыслимые ловушки, оставленные разработчиком.

Даже если приложение и установит свой vxd (vxd выполняется в нулевом кольце и может вытворять что угодно), это только *облегчит* задачу взломщика, так как взаимодействовать с vxd защита сможет только через специальный API, что упрощает изучение алгоритма защиты и эмуляцию работу vxd для «отвязки» приложения от электронного ключа или ключевой дискеты.

Но даже на уровне нулевого кольца в Windows очень трудно что-либо скрыть, для обеспечения совместимости со всем парком Windows-подобных операционных систем приходится использовать только документированные возможности. Строить в «окнах» защиту — все равно что пытаться заблудиться в парке. Будь там хоть миллион деревьев — все они геометрически правильно расположены и обильно увешаны табличками «выход — там».

Таким образом, надежно противостоять изучению программы очень трудно, если вообще возможно. Однако многие приемы против отладчиков и дизассемблеров просто интересны сами по себе и достойны того, чтобы их рассмотреть в этой книге.

Приемы против отладчиков

Немного истории

Историческая система взаимных грабежей и вымогательств остановится здесь, на Арраки. Нельзя с годами преодолеть расхищение того, в чем нуждаешься, не принимая во внимание интересы тех, кто придет после тебя.

Ф. Херберт. Дюна

Раньше всех появился debug.com — пародия, отдаленно напоминающая отладчик, зато входящая в штатную поставку MS-DOS. Сегодня этот инструмент годится разве что для забавы и изучения ассемблера. Впрочем, и тогда от него мало кто был в восторге, и новые отладчики росли как грибы после дождя. Правда, большинство из них недалеко ушло от своего прототипа, отличаясь от оригинала разве что интерфейсом.

Это было золотое время разработчиков защит. Стоило лишь «запереть» клавиатуру, запретить прерывания, сбросить флаг трассировки, и отладка программы становилась невозможной.

Первые мало-мальски пригодные для взлома отладчики появились только после оснащения компьютеров 80286 процессором. В памяти хакеров навсегда останутся **AFD PRO**, написанный в 1987 году AdTec GmbH, знаменитый **Turbo Debugger**, созданный годом позже двумя братьями Крисом и Ричем Вильямсами, первый эмулирующий отладчик Сергея Пачковки, написанный, правда, с большим опозданием: в 1991 году. Разработчики защит крикнули, но выдержали — эти отладчики по-прежнему позволяли отлаживаемой программе захватить над собой контроль и очень плохо переносили «извращения» со стеком, экраном, клавиатурой...

Ситуация изменилась с выходом 80386 процессора — резкое усложнение программного обеспечения (и как следствие — огромные сложности с его отладкой) диктовало необходимость наличия развитых отладочных средств в самом процессоре. И в 386 они появились! С этого момента разработчикам защит стали наступать на пятки.

Масла в огонь подлила NuMega, выпустившая в конце восьмидесятых годов свой замечательный Soft-Ice, пользовавшийся у хакеров огромной популярностью, а ныне портированный на Windows 9x и Window NT/2000 и до сих пор остающийся бесспорным фаворитом (хотя не без конкуренции). Впрочем, неверно было бы считать, что NuMega — криминальная фирма, а Soft-Ice исключительно хакерский продукт. Этот отладчик предназначен в первую очередь для разработчиков драйверов и легальных исследователей операционной системы (не разбираясь во внутренностях ОС, с драйверами особо не разгонишься).

Но так или иначе, Soft-Ice задал копоти всем защитам и их разработчикам. Пускай он не был (да и сегодня не стал) полностью Stealth-отладчиком, невидимым для отлаживаемых программ, имел и имеет ряд ошибок, позволяющих обна-

ружить отладчик, завесить его и/или вырваться защите из-под контроля, но... в умелых руках отладчик справлялся со всеми этими ограничениями и обходил заботливо расставленные «капканы». И с каждой версией Айса противостоять ему становилось все труднее и труднее (старые ошибки устранялись быстрее, чем вносились новые).

Постепенно мода на антиотладочные приемы сошла на нет и уж совсем заглохла под победное шествие Windows. Распространилось совершенно нелепое убеждение, что под Windows на прикладном уровне дернуть хвост человеку с отладчиком невозможно. Это вызывает ухмылку профессионалов, эпизодически встраивающих разные ловушки в свои программы — так, больше для разминки (дабы мозги жиром не заплыли), чем для серьезной борьбы с хакерами.

Бороться с хакерами при современном уровне средств анализа приложений несколько наивно — те и от Тигра хвост оторвут, но сегодня, кроме хакеров, серьезную угрозу представляют и вчерашние желторотые пользователи, начитавшиеся различных faq, «как ломать программы» (благо сейчас они доступны всем, кому ни попадя), и теперь только и ищущие, на чем испытать свою богатырскую силу.

Как работает отладчик

...древним с их мыслящими машинами было куда легче.

Ф. Херберт. Дюна

Бороться с отладчиком, не представляя себе, как он работает, было бы по меньшей мере некультурно, поэтому ниже будут рассмотрены базовые принципы, лежащие в его основе. Это изложение не является всеобъемлющим, но, тем не менее, позволяет читателю составить общее представление о вопросе. Технические подробности исчерпывающе изложены в главе «Debugging and Performance Monitoring» технического руководства «Intel Architecture Software Developer's Manual Volume 3: System Programming Guide», бесплатно распространяемого фирмой Intel.

Все существующие отладчики можно разделить на две категории — первые используют отладочные средства процессора, а вторые самостоятельно эмулируют процессор, полностью контролируя выполнение «подопытной» программы.

Качественный эмулирующий отладчик отлаживаемому коду ни обнаружить, ни обойти невозможно, но полноценных эмуляторов Pentium-процессоров на сегодняшний день нет и вряд ли они появятся в обозримом будущем.

Да и есть ли смысл их создавать? Микропроцессоры Pentium предоставляют в распоряжение разработчика богатейшие отладочные возможности, позволяющие контролировать даже привилегированный код! Они поддерживают **пошаговое исполнение** программы, отслеживают **выполнения инструкции по заданному адресу**, контролируют **обращения к заданным ячейкам памяти** (или портам ввода-вывода), сигнализируют о **переключениях задач** и т. д.

Если бит трассировки регистра флагов установлен, то после выполнения каждой машинной инструкции автоматически генерируется отладочное исключе-

ние INT 1, передавая управление отладчику. Отлаживаемый код может обнаружить трассировку анализом регистра флагов, поэтому для обеспечения собственной невидимости отладчик должен распознавать команды чтения регистра флагов и эмулировать их выполнение, возвращая нулевое значение флага трассировки.

Следует обратить внимание на одно важное обстоятельство: после выполнения команды, модифицирующей значение регистра SS, отладочное исключение *не генерируется!* Отладчик должен уметь распознавать такую ситуацию и самостоятельно устанавливать точку останова на следующую инструкцию. В противном случае войти в процедуру, предваренную инструкцией POP SS (например, так: PUSH SS; POP SS; CALL MySecretProc), автоматический трассировщик не сможет. Не все современные отладчики учитывают эту тонкость, и такой прием, несмотря на свою архаичность, может оказаться далеко не бесполезным.

Четыре отладочных регистра DR0-DR3 хранят линейные адреса четырех контрольных точек, а управляющий регистр DR7 содержит для каждой из них условие, при выполнении которого процессор генерирует исключение INT 0x1, передавая управление отладчику. Всего существует четыре различных условия: прерывание при **выполнении команды**, прерывание при **модификации ячейки памяти**, прерывание при **чтении** или **модификации**, но **не исполнении ячейки памяти** и прерывание при **обращении к порту ввода-вывода**.

Установкой специального бита можно добиться генерации отладочного исключения при всяком обращении к отладочным регистрам, которое возникает даже в том случае, если их пытается прочесть (модифицировать) привилегированный код. Грамотно спроектированный отладчик может скрыть факт своего присутствия, не позволяя отлаживаемому коду себя обнаружить, какие бы ни были у него привилегии (правда, если «подопытный» код отлаживает сам себя, задействовав все четыре контрольные точки, отладчик не сможет работать).

Если бит T в TSS отлаживаемой задачи установлен, то при каждом **переключении** на нее будет генерироваться отладочное исключение **до** выполнения первой команды задачи. Чтобы предотвратить собственное обнаружение, отладчик может отслеживать всякие обращения к TSS и возвращать программе подложные данные. Необходимо заметить: Windows NT по соображениям производительности не использует TSS (точнее, использует, но всего один) и эта отладочная возможность для нее совершенно бесполезна.

Программная точка останова — единственное, что нельзя замаскировать, не прибегая к написанию полноценного эмулятора процессора. Она представляет собой однобайтовый код 0xCC, который, будучи помещенным в начало инструкции, вызывает исключение INT 0x3 при попытке ее выполнения. Отлаживаемой программе достаточно подсчитать свою контрольную сумму, чтобы выяснить, была ли установлена хотя одна точка останова или нет. Для достижения этой цели она может воспользоваться командами MOV, MOVS, LODS, POP, CMP, CMPS или любыми другими, никакому отладчику невозможно их всех отследить и проэмулировать.

Настоятельно рекомендуется использовать программные точки останова в тех и только в тех случаях, когда аппаратных уже не хватает. Однако практически все современные отладчики (в том числе и Soft-Ice) всегда устанавливают

программные, а не аппаратные точки останова. Это обстоятельство может быть с успехом использовано в защитных механизмах, примеры реализаций которых приведены в разделе «Как противостоять трассировке».

Обработка исключений в реальном и защищенном режимах

Когда возникает отладочное исключение (как, впрочем, и любое другое исключение вообще), процессор заносит в стек регистр флагов, адрес следующей (или текущей — в зависимости от рода исключения) выполняемой инструкции и лишь затем передает управление отладчику.

В реальном режиме флаги с адресом возврата заносятся *в стек отлаживаемой программы*, поэтому факт отладки обнаружить очень просто — достаточно контролировать целостность содержимого, лежащего выше указателя стека. Или, как вариант, установить указатель на его вершину, тогда добавление новых данных в стек окажется невозможным и отладчик не сможет функционировать.

Иная ситуация складывается при работе в защищенном режиме — обработчик исключения может находиться в своем собственном адресном пространстве и не использовать никаких ресурсов отлаживаемого приложения, в том числе и стека. Грамотно спроектированный отладчик защищенного режима ни обнаружить, ни заблокировать принципиально невозможно, даже привилегированному коду, исполняющемуся в нулевом кольце.

Сказанное справедливо для Windows NT, но неприменимо к Windows 9x — эта операционная система не использует должным образом всех преимуществ защищенного режима и всегда «замусоривает» стек отлаживаемой задачи, независимо от того, находится ли она под отладкой или нет.

Как хакеры ломают программы

Вскрыть защитный механизм взломщику в общем случае не проблема. Куда сложнее найти его во многих мегабайтах кода ломаемого приложения. Сегодня мало кто использует для этой цели автоматическую трассировку — на смену ей пришли аппаратные контрольные точки.

Например, пусть некая защита запрашивает пароль и затем каким-то образом удостоверяется в его подлинности (например, сравнивает с оригиналом) и в зависимости от результатов проверки передает управление соответствующей ветке программы. Вскрыть такую защиту взломщик может, даже не вникая в алгоритм аутентификации! Он просто введет первый пришедший ему на ум пароль (не обязательно совпадающий с правильным), найдет его в памяти, установит контрольную точку на первый символ строки своего пароля, дождется «всплытия» отладчика, отследившего обращение к паролю, выйдет из сравнивающей процедуры и «подправит» условие перехода так, чтобы управление всегда получала нужная ветвь программы.

Время снятия подобных защит измеряется *секундами* (!), и обычно такие программы ломаются раньше, чем успевают дойти до легального потребителя. К счастью, этому можно противостоять!

Как защитить свои программы

Откуда бы ни бралась ключевая информация — из реестра, файла или клавиатуры, — взломщик может практически мгновенно локализовать ее местоположение в памяти и установить на него контрольную точку. Помешать этому нельзя, но не составит труда подложить хакеру неожиданный сюрприз — пусть ключевая информация анализируется не сразу же после получения, а передается в качестве аргумента множеству функций, которые что-то с ней делают и затем передают другим функциям, а те, в свою очередь, следующим.

Защитный механизм может быть встроен во что угодно, хоть в процедуру открытия файла или расчета зарплаты. Не стоит делать явных проверок, пусть лучше в случае вызова функции с неверной ключевой информацией она возвратит неправильный результат, но не сигнализирует об ошибке. Взломанная программа на первый взгляд будет исправно работать, и далеко не сразу выяснится, что работает она неправильно (например, выводит на экран одни числа, а на принтер — совсем другие). А чтобы обезопасить легального пользователя от ошибочного ввода пароля, достаточно в одном месте явно проверить его контрольную сумму, которая не дает взломщику никакой информации об истинном значении пароля.

Таким образом, защита как бы «размазывается» по всей программе, буфера с ключевыми данными многократно дублируются, и на отслеживание обращений у взломщика не хватит ни контрольных точек, ни терпения для анализа огромного объема манипулирующими с ними кода. Будет еще лучше, если после выполнения проверки ключевой информации эти же самые буфера использовать для хранения служебных данных, обращение к которым происходит по возможности максимального часто. Это не позволит взломщику быстро отделить защитный механизм от прочего прикладного кода.

Попутно: поскольку большинство взломщиков ставит контрольную точку на начало контрольного буфера, имеет смысл поместить в первые четыре байта ключа «заглушку», обращение к которой либо не происходит вовсе, либо с ней манипулирует имитатор защиты, направляя хакера по ложному пути.

В такой ситуации взломщику ничего не останется, кроме того, как, затравившись пивом, плотно засесть за кропотливое изучение *всего* кода программы, прямо или косвенно манипулирующего с ключевой информацией (а это многие мегабайты дизассемблерного листинга!). Если критическая часть кода зашифрована, причем ни в какой момент работы программы не расшифровывается полностью (при выходе в каждую функцию она расшифровывается, а при выходе зашифровывается вновь), хакер не сможет получить готовый к дизассемблированию дамп и будет вынужден прибегнуть к трассировке. А вот тут его будет ждать второй сюрприз!

Как противостоять трассировке

Принципиальная возможность создания подлинно «невидимых» отладчиков большей частью просто возможностью и остается — большинство из них позволяет обнаружить себя даже непривилегированному коду.

Наибольшие нарекания вызывает использование однобайтового кода 0xCC для создания точки останова вместо поручения той же задачи специально для этого предназначенным отладочным регистром. Так поступают Soft-Ice, Turbo Debugger, Code Viewer и отладчик, интегрированный в Microsoft Visual Studio. Причем последний неявно использует точки останова при пошаговом прогоне программы, помещая в начало следующей инструкции этот пресловутый байт 0xCC.

Тривиальная проверка собственной целостности позволяет обнаружить факт установки точек останова, свидетельствующий об отладке. Не стоит использовать конструкции наподобие *if (CalculateMyCRC()!=MyValidCRC) {printf("Hello, Hacker!\n");return;}*, их слишком легко обнаружить и нейтрализовать, подправив условный переход так, чтобы он всегда передавал управление нужной ветке программы. Лучше расшифровывать полученным значением контрольной суммы критические данные или некоторый код.

Простейшая защита может выглядеть, например, так:

Листинг 221

```
int main(int argc, char* argv[])
{
    // зашифрованная строка Hello, Free World!
    char s0[]="\x0C\x21\x28\x28\x2B\x68\x64\x02\x36\x
\x21\x21\x64\x13\x2B\x36\x28\x20\x65\x49\x4E";
    __asm
    {
        BeginCode:                ; //начало контролируемого кода
            pusha                  ; //сохранение всех регистров общего назначения
            lea    ebx,s0          ; // ebx=&s0[0]
        GetNextChar:              ; // do
            XOR    eax,eax         ; // eax = 0;
            LEA    esi,BeginCode   ; // esi = &BeginCode
            LEA    ecx,EndCode     ; // вычисление длины...
            SUB    ecx,esi         ; // ...контролируемого кода
        HarvestCRC:              ; // do
            LODSB                  ; // загрузка очередного байта в al
            ADD    eax,eax         ; // вычисление контрольной суммы
        LOOP HarvestCRC          ; // until(-cx>0)
            xor    [ebx],ah        ; // расшифровка очередного символа s0
            inc    ebx             ; // указатель на след. символ
            cmp    [ebx],0         ; // until (пока не конец строки)
            jnz    GetNextChar     ; // продолжить расшифровку
            popa                   ; // восстановить все регистры
        EndCode:                 ; // конец контролируемого кода
            NOP                   ; // Safe BreakPoint here
    }
}
```

```

printf(s0)                ; // вывод строки на экран
return 0;
}

```

При нормальном запуске на экране должна появиться строка «*Hello, Free World!*», но при прогоне под отладчиком при наличии хотя бы одной точки останова, установленной в пределах от BeginCode до EndCode, на экране появится бессмысленный мусор наподобие: «*Ignnm."Dpgg"Umpnf#0*».

Значительно усилить защиту можно, поместив процедуру подсчета контрольной суммы в отдельный поток, занимающийся (для сокрытия своей деятельности) еще чем-нибудь полезным, так чтобы защитный механизм по возможности не бросался в глаза.

Потоки вообще великая вещь, требующая к себе особого подхода. Человеку очень трудно смириться с тем, что программа может исполняться во множестве мест одновременно. Распространенные отладчики грешат тем, что отлаживают каждый поток по отдельности, но никогда два и более сразу. Приведенный ниже пример показывает, как это можно использовать для защиты.

Листинг 222

```

// Эта функция будет выполняться в отдельном потоке
// ее назначение незаметно изменять регистр символов в строке,
// содержащей имя пользователя
void My(void *arg)
{
    int p=1;           // Указатель на шифруемый байт
                       // обратите внимание, шифровка выполняется
                       // не с первого байта, это позволяет обойти
                       // контрольную точку, установленную на начало буфера
    // выполнять до тех пор, пока не встретится перенос строки
    while ( ((char *) arg)[p]!='\n')
    {
        // ожидать, пока очередной символ не будет инициализирован
        while( ((char *) arg)[p]<0x20 );

        // инвертировать пятый бит
        // это приводит к изменению регистра латинских
        // символов на противоположный
        ((char *) arg)[p] ^=0x20;

        // указатель на следующий обрабатываемый байт
        p++;
    }
}

int main(int argc, char* argv[])
{
    char name[100];      // буфер, содержащий имя пользователя
    char buff[100];      // буфер, содержащий пароль

    // забивка буфера имени пользователя нулями
    // некоторые компиляторы это делают за нас, но не все!
    memset(&name[0],0,100);
}

```

```
// выполнять процедуру My в отдельном потоке
_beginthread(&My, NULL, (void *) &name[0]);

// запрос имени пользователя
printf("Enter name:"); fgets(&name[0], 66, stdin);

// запрос пароля
// Важно: пока пользователь вводит пароль, второй поток
// получает достаточно квантов времени, чтобы изменить
// регистр всех символов имени пользователя
// Это обстоятельство не так очевидно и не вытекает из
// беглого анализа программы, особенно при ее исследовании
// под отладчиком, слабо показывающим влияние
// отдельных компонентов программы друг на друга
printf("Enter password:"); fgets(&buff[0], 66, stdin);

// сравнение имени и пароля с эталонными значениями
if (!strcmp(&buff[0], "password\n"))
// Важно: поскольку введенное пользователем имя было
// преобразовано, фактически происходит сравнение не
// strcmp(&name[0], "KPNC\n") а strcmp(&name[0], "Kpnc\n"),
// что далеко не очевидно на первый взгляд
    || strcmp(&name[0], "KPNC\n"))
// правильные имя и пароль
    printf("USER OK\n");
else
// ошибка в вводе имени или пароля
    printf("Wrong user or password!\n");
return 0;
}
```

На первый взгляд программа ожидает «услышать» «KPNC:password». Но так ли это на самом деле? А вот и нет! Верный ответ — «Kpnc:password». В то время пока пользователь вводит свой пароль, второй поток обрабатывает буфер, содержащий его имя, меняет регистр всех символов, кроме первого, на противоположный. Весь фокус в том, что при пошаговой трассировке одного потока все остальные потоки выполняются независимо от него и могут произвольным образом вклиниваться в работу отлаживаемого потока, например, модифицировать его код.

Взять потоки под контроль можно введением в каждый из них точки останова, но если потоков окажется больше четырех (а что мешает разработчику защиты их создать?), отладочных регистров на всех не хватит и придется прибегать к использованию опкода 0xCC, который защитному механизму ничего не стоит обнаружить!

Ситуация усугубляется тем, что большинство отладчиков, в том числе и хваленый Soft-Ice, очень плохо переносят программы со **структурной обработкой исключений (SEH)**. Инструкция, вызывающая обрабатываемое исключение, либо «срывает» отладчик, выходя из-под его контроля, либо передает управление на библиотечный фильтр исключений, который, прежде чем передать управление прикладной обработке, вызывает множество своих служебных функций, в которых взломщику немудрено и «утонуть».

Впрочем, по сравнению с ранними версиями SoftIce даже это большой прогресс, так как раньше он жестко держал некоторые прерывания, не позволяя программе самостоятельно обрабатывать, скажем, деление на ноль.

Если попытаться прогнать приведенный пример под SoftIce вплоть до версии 4.05 включительно (остальные не проверял ввиду их отсутствия, но, скорее всего, они будут вести себя точно так же), он, достигнув строки `int c=c/(a-b)`, внезапно «слетит», теряя контроль над отлаживаемым приложением. Теоретически исправить ситуацию можно заблаговременной установкой точки останова на первую команду блока `__except`, но попробуй-ка вычислить, где расположен этот блок, не заглядывая в исходный текст, которого у хакера заведомо нет!

Листинг 223

```
// Пример защиты, построенный на обработке структурных исключений
int main(int argc, char* argv[])
{
    // Защищенный блок
    __try{
        int a=1;           // Попытка деления на ноль
        int b=1;           // многословность объясняется тем,
        int c=c/(a-b);      // что большинство компиляторов
                           // выдают ошибку, встретив конструкцию
                           // наподобие int a=a/0;
        // при выполнении следующей инструкции отладчик SoftIce
        // теряет контроль над отлаживаемой программой и "слетает"
        // некий код, который никогда не получит управления,
        // но может быть вставлен для "отвода глаз". Если значение
        // переменным a и b присваивается не непосредственно, а
        // из результата, возвращенного некими функциями, то при
        // дизассемблировании программы их равенство будет не так
        // очевидно. В результате взломщик может потратить много
        // времени на анализ совершенно бесполезного кода
    }

    __except(EXCEPTION_EXECUTE_HANDLER)
    {
        // этот код получит управление при возникновении
        // исключения "деление на ноль",
        // но отладчик Soft-Ice не распознает такой ситуации
        // и требует ручной установки точки останова на первую
        // инструкцию блока __except,
        // а чтобы определить адрес блока __except, требуется
        // разобраться, каким именно образом реализована поддержка
        // SEH в конкретном компиляторе.
    }
}
```

Прежде чем справиться с такой защитой, взломщику придется основательно изучить реализацию механизма обработки структурных исключений как на уровне операционной системы, так и на уровне конкретного компилятора. В по-

давляющем большинстве существующей литературы этот вопрос обходится стороной. И неспроста — реализация SEH действительно очень сложна, громоздка, многословна. Все это приводит к тому, что большинство программистов и технических писателей совершенно не представляют, что находится у нее «под капотом».

Поскольку SEH по-разному реализована в каждом компиляторе, нет ничего удивительно, что Soft-Ice отказывается ее поддерживать. Поэтому предложенный вариант защиты очень стоек к взлому и в то же время крайне прост в реализации. А самое важное — он одинаково хорошо работает во всех операционных системах семейства Windows от 95 до 2000.

Как противостоять контрольным точкам останова

Контрольные точки, установленные на важнейшие системные функции, — мощное оружие в руках взломщика. Пусть, к примеру, защита пытается открыть ключевой файл. Под Windows существует только один документированный способ это сделать — вызвать функцию `CreateFile` (точнее, `CreateFileA` или `CreateFileW` для ASCII и UNICODE-имени файла соответственно). Все остальные функции, наподобие `OpenFile`, доставшиеся в наследство от ранних версий Windows, на самом деле представляют собой переходники к `CreateFile`.

Зная об этом, взломщик может заблаговременно установить точку останова на адрес начала этой функции (благо он ему известен) и мгновенно локализовать защитный код, вызывающий эту функцию, ну а остальное, как говорится, дело техники.

Но не всякий взломщик знает, что открыть файл можно и другим путем — вызвать функцию `ZwCreateFile` (равно как и `NtCreateFile`), экспортируемую `NTDLL.DLL`, или обратиться напрямую к ядру вызовом прерывания **INT 0x2Eh**. Сказанное справедливо не только для `CreateFile`, но и для всех остальных функций ядра. Причем для этого не нужны никакие привилегии и такой вызов можно осуществить даже из прикладного кода!

Опытного взломщика такой трюк надолго не остановит, но почему бы ему не приготовить один маленький сюрприз, поместив вызов `INT 0x2E` в блок `__try`. Это приведет к тому, что управление получит не ядро системы, а обработчик данного исключения, находящийся за блоком `_try`. Взломщик же, не имеющий исходных текстов, не сможет быстро определить, относится ли данный вызов к блоку `_try` или нет. Отсюда он может быть легко введен в заблуждение — достаточно имитировать открытие файла, не выполняя его на самом деле! Кроме того, ничего не мешает использовать прерывание `INT 0x2E` для взаимодействия компонентов своей программы — взломщику будет очень не просто отличить, какой вызов пользовательский, а какой системный.

Хорошо, с ядром все понятно, а как же быть с функциями модулей `USER` и `GDI`, например `GetWindowsText`, использующейся для считывания введенной пользователем ключевой информации (как правило, серийного номера или пароля)?

На помощь приходит то обстоятельство, что практически все эти функции начинаются с инструкций `PUSH EBP\MOV EBP, ESP`, которые прикладной код может выполнить и самостоятельно, передав управление **не на начало функции, а на три байта ниже**. (Поскольку `PUSH EBP` изменяет стек, приходится прибегать к передаче управления посредством `JMP` вместо `CALL`.) Контрольная точка, установленная взломщиком на начало функции, не вызовет никакого действия! Такой трюк может сбить с толку даже опытного хакера, хотя рано или поздно он все равно раскусит обман, но...

Если есть желание окончательно отравить взломщику жизнь, следует скопировать системную функцию в свой собственный стек и передать на него управление — контрольные точки взломщика «отдыхают»! Основная сложность заключается в необходимости распознавания всех инструкций с относительными адресными аргументами и их соответствующей коррекции. Например, двойное слово, стоящее после инструкции `CALL`, представляет собой не адрес перехода, а разность целевого адреса и адреса следующей за `CALL` инструкции. Перенос инструкции `CALL` на новое место потребует коррекции ее аргумента. Впрочем, эта задача не так сложна, как может показаться на первый взгляд (глаза страшатся, а руки делают), и результат оправдывает средства. Во-первых, при каждом запуске функции можно произвольным образом менять ее адрес, во-вторых, проверкой целостности кода легко обнаружить программные точки останова — а аппаратных точек на все вызовы просто не хватит!

Разве ж не заслуживают награды за свою целеустремленность те единицы, которые такую защиту взламывают?! (Под наградой здесь подразумевается отнюдь не сама взломанная программа, а глубокое чувство удовлетворения от того, что «я это сделал!».)

Еще легче противостоять аппаратным точкам останова на память, поскольку их всего четыре и каждая может контролировать не более двойного слова, взломщик может одновременно контролировать не более 16 байтов памяти. Если же обращения к буферам, содержащим ключевую информацию, будут происходить не последовательно байт за байтом от начала до конца, а произвольно и количество самих буферов окажется больше четырех, отследить все операции чтения-записи в них станет невозможно.

Некоторые отладчики поддерживают возможность установки точки останова на диапазон памяти, но ее функциональность вызывает большие сомнения — единственный способ контролировать целый регион — трассировать исследуемую программу, проверяя, не обращается ли очередная команда к охраняемому диапазону, и если да — генерировать исключение.

Во-первых, команд, манипулирующих с памятью, очень много, и можно придумать самые неожиданные комбинации, например, установить указатель стека на требуемую ячейку памяти и вызвать `RET` для чтения содержащегося в ней значения. Во-вторых, возникшее при этом исключение может служить хорошим средством избавления от трассировщика (см. разд. «Как противостоять трассировке»).

Таким образом, справиться с контрольными точками защитному механизму совсем нетрудно!

Точка останова представляет собой однобайтовую команду `0xCC`, генерирующую исключение `0x3` при попытке ее выполнения (в просторечии «дергающие отладочным прерыванием»). Обработчик `INT 0x3` получает управление и может делать с программой абсолютно все, что ему заблагорассудится, но прежде — до вызова прерывания — в стек заносятся текущие *регистр флагов*, указатель кодового сегмента (*регистр CS*), указатель команд (*регистр IP*), запрещаются прерывания (*очищается флаг IF*) и *сбрасывается флаг трассировки*, — словом, вызов отладочного прерывания не отличается от вызова любого прерывания вообще (см. рис. 37).

Чтобы узнать, в какой точке программы произошел останов, отладчик извлекает из стека сохраненное значение регистров, не забывая о том: `CS:IP` указывают на *следующую выполняемую команду*.

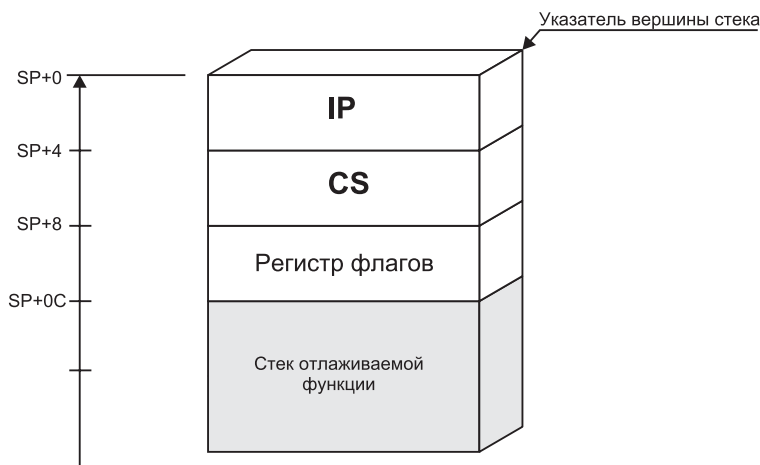


Рис. 37. Состояние стека на момент входа в обработчик прерывания

Условно точки останова (называемые также контрольными точками) можно разделить на две категории: *точки останова, жестко прописанные в программе самим разработчиком*, и *точки динамические, устанавливаемые самим отладчиком*. Ну, с первыми все ясно: хочешь остановить программу и передать управление отладчику в таком-то месте — пишешь `__asm{ int 0x3 }` — и надевай, тигра, Шляпу!

Несколько сложнее установить точку в произвольное место программы: сначала отладчик должен сохранить текущее значение ячейки памяти по указанному адресу, затем записать сюда код `0xCC`, а перед выходом из отладочного прерывания вернуть все на место и модифицировать сохраненный в стеке `IP` для перемещения его на начало восстановленной команды (иначе он будет указывать на ее середину).

Какими недостатками обладает механизм точек останова 8086 процессора? Первое, и самое неприятное, состоит в том, что отладчик, устанавливая точку останова, вынужден непосредственно модифицировать код.

Причем Soft-Ice неявно помещает точку останова в начало каждой следующей команды при трассировке программы по *Step Over* (клавиша <F10>)! Разумеется, это искажает контрольную сумму, чем и пользуется защита.

Самое простое решение проблемы — положить кирпич на клавишу <F8> (покомандная трассировка) и идти пить чай, пока программа будет расшифровываться. Шутка, конечно. А если говорить серьезно, то необходимо вспомнить, в каком веке мы живем, и, отбросив каменные топоры, установить аппаратную точку останова (см. раздел «*Приемы против отладчиков защищенного режима*»).

Наши же предки (хакеры восьмидесятых) в этой ситуации обычно «вручную» расшифровывали программу, а затем затирали процедуру расшифровки NOP'ми, после чего отладка программы уже не представляла проблемы (естественно, если в защите не было других нычек). До появления IDA расшифровщику приходилось писать на Си (Паскале, Базике) в виде самостоятельной программы, теперь же эта задача упростилась и заниматься расшифровкой стало возможно непосредственно в самом дизассемблере.

Техника расшифровки сводится к воспроизведению расшифровщика на языке IDA-Си — в данном случае сначала необходимо вычислить контрольную сумму от BeginCode до EndCode, подсчитывая сумму байтов, используя при этом младший байт контрольной суммы для загрузки следующего символа, а затем полученным значением «поксорить» строку s0. Все это можно сделать следующим скриптом (предполагается, что в дизассемблированном тексте соответствующие метки уже расставлены):

Листинг 225

```
auto a; auto p; auto crc; auto ch;
for (p=LocByName("s0");Byte(p)!=0;p++)
{
    crc=0;

    for(a=LocByName("BeginCode");a<(LocByName("EndCode"));a++)
    {
        ch=Byte(a);
        // Поскольку IDA не поддерживает типов byte и word
        // (а напрасно), приходится заниматься битовыми
        // выкрутасами - сначала очищать младший байт crc,
        // а затем копировать в него считанное значение ch
        crc = crc & 0xFFFFF00;
        crc = crc | ch;
        crc=crc+crc;
    }
    // Берем старший байт от crc
    crc=crc & 0xFFFF;
    crc=crc / 0x100;

    // Расшифровываем очередной байт строки
    PatchByte(p,Byte(p) ^ crc);
}
```

Если под рукой нет IDA, эту же операцию можно осуществить и в HIEW-e:

```

NoTrace.exe  iW      PE 00001040 a32 <Editor>      28672 ? Hiew 6.04 (c)SEN
00401003: 83EC18          sub     esp,018 ;"i"
00401006: 53              push    ebx
00401007: 56              push    esi
00401008: 57              push    edi
00401009: B905000000      000005 ;" ♣"
0040100E: BE30604000      406030 ;" @`0"
00401013: 8D7DE8          p)[-0018]
00401016: F3A5
гнать          00401018: A4
отсюда-> 00401019: 6660
0040101B: 8D9DE8FFFF
00401021: 33C0
.0040101B: 8D9DE8FFFF
.00401021: 33C0          xor     eax,eax
.00401023: 8D3519104000    lea     esi,[000401019] ; < BeginCode
.00401029: 8D0D40104000    lea     ecx,[000401040] ; < EndCode
.0040102F: 2BCE          sub     ecx,esi
.00401031: AC          lodsb
.00401032: 03C0          add     eax,eax
00401034: E2FB          loop   000001031
00401036: 3023          xor     [ebx],ah
00401038: 43          inc     ebx
00401039: 803B00        cmp     b,[ebx],000 ;" "
0040103C: 75E3          jne     000001021
0040103E: 6661          popa
досюда-> 00401040: 90          nop
00401041: 8D45E8          lea     eax,[ebp][-0018]
00401044: 50          push    eax
00401045: E80C000000      call   000001056
0040104A: 83C404          add     esp,004 ;"♦"
1Help  2Size  3Direct 4Clear  5ClrReg 6      7Exit  8      9Store 10Load

```

[Byte/Forward]	
1>mov	bl,al
2	add ebx,ebx
3	
4	
5	
6	

AX=0061
BX=44C2
CX=0000
DX=0000
SI=0000
DI=0000

На первой стадии производится подсчет контрольной суммы. Загрузив файл в HIEW, находим нужный фрагмент (клавиша **<ENTER>**, **<ENTER>** для перехода в режим ассемблера и клавиши **<F8>**, **<F5>** — для прыжка в точку входа, далее находим в стартовом коде процедуру main), нажимаем клавишу **<F3>** для разрешения правки файла, вызываем редактор скрипта-расшифровщика (сочетание клавиш **<CTRL-F7>**, впрочем, эта комбинация варьируется от версии к версии) и вводим следующий код:

```

mov bl, al
add ebx, ebx

```

Вместо EBX можно использовать и другой регистр, но не EAX: HIEW, считывая очередной байт, обнуляет EAX целиком. Теперь установим курсор на строку 0x401019 и, нажимая клавишу **<F7>**, погоним расшифровщик до строки 0x401040, **не** включая последнюю. Если все сделано правильно, в старшем байте BX должно находиться значение 0x44, это и есть контрольная сумма.

На второй стадии находим зашифрованную строку (ее смещение грузится в ESI и равно .406030) и ксори́м ее по 0x44. (Нажимаем клавишу **<F3>** для перехода в режим правки, комбинацию клавиш **<CTRL-F8>** — для задания ключа шифрования — 0x44, а затем ведем расшифровщик по строке, нажимая клавишу **<F8>**.)

```

NoTrace.exe  iW      PE 00006040      <Editor>      28672 ? Hiew 6.04 (c)SEN
00006030: 48 65 6C 6C-6F 2C 20 46-72 65 65 20-57 6F 72 6C  Hello, Free Worl
00006040: 20 65 49 4E-00 00 00 00-7A 1B 40 00-01 00 00 00  eIN      z-@ ©

```

Остается лишь забить XOR в строке 0x401036, командами NOP, иначе при запуске программы он испортит расшифрованный текст (зашифрует его вновь) и программа, естественно, работать не будет.

Теперь после снятия защиты ее можно безболезненно отлаживать сколько душе угодно — да, контрольная сумма по-прежнему считается, но теперь она не используется (если бы в защите была проверка на корректность CRC, пришлось бы нейтрализовать и ее, но в этом примере для упрощения понимания ничего подобного нет).

Как обнаружить отладку средствами Windows

В своей книге «Секреты системного программирования в Windows 95» Мэт Питтрек описал структуру *информационного блока цепочки (Thread Information Block)*, рассказав о назначении многих недокументированных полей. Особый интерес для данной статьи представляет двойное слово, лежащее по смещению 0x20 от начала структуры TIB, содержащее *контекст отладчика* (если данный процесс отлаживается) или ноль в противном случае. Информационный блок цепочки доступен через селектор, загруженный в регистр FS, и без проблем может читаться прикладным кодом.

Если двойное слово FS:[0x20] не равно нулю, процесс находится под отладкой. Это настолько заманчиво, что некоторые программисты включили такую проверку в свои защиты, не обратив внимания на ее «недокументированность». В результате их программы не смогли исполняться под Windows NT, поскольку она хранит в этом поле не контекст отладчика, а идентификатор процесса, который никогда не бывает равным нулю, отчего защита ошибочно полагает, что находится под отладкой.

Это обстоятельство было подробно описано самим же Мэтом Питтреком в майском номере журнала «Microsoft Systems Journal» за 1996 год, где в статье «Under The Hood» он привел следующую структуру:

Листинг 226

```
union                                // 1Ch (NT/Win95 differences)
{
    struct // Win95 fields
    {
        WORD    TIBFlags;                // 1Ch
        WORD    Win16MutexCount;        // 1Eh
        DWORD    DebugContext;           // 20h
        DWORD    pCurrentPriority;        // 24h
        DWORD    pvQueue;                // 28h Message Queue selector
    } WIN95;

    struct // WinNT fields
    {
        DWORD    unknown1;               // 1Ch
        DWORD    processID;              // 20h
    }
}
```

```
        DWORD threadID;           // 24h
        DWORD unknown2;           // 28h
    } WINNT;
} TIB_UNION2;
```

Этот случай в очередной раз подтвердил: не стоит без особой необходимости использовать недокументированные особенности, как правило, они приносят больше проблем, чем пользы.

Приемы против дизассемблеров

Самомодифицирующийся код в современных операционных системах

— Это мы говорим, будто мы выдумываем. На самом деле все давным-давно выдумано.

Стругацкие. Трудно быть богом

Лет десять—двадцать тому назад, в эпоху рассвета MS-DOS, программистами широко использовался самомодифицирующийся код, без которого не обходилась практически ни одна мало-мальски серьезная защита. Да и не только защита, он встречался в компиляторах, компилирующих код в память, распаковщиках исполняемых файлов, полиморфных генераторах и т. д. и т. п.

В середине девяностых годов началась массовая миграция пользователей с MS-DOS на Windows 95/Windows NT, и разработчикам пришлось задуматься о переносе накопленного опыта и приемов программирования на новую платформу, от бесконтрольного доступа к «железу», памяти, компонентам операционной системы и связанным с ними хитроумными трюками программирования пришлось отвыкать. В частности, стала невозможна непосредственная модификация исполняемого кода приложений, поскольку Windows защищает его от непреднамеренных изменений. Это привело к рождению нелепого убеждения, дескать, под Windows создание самомодифицирующегося кода вообще невозможно, по крайней мере, без использования VxD и недокументированных возможностей операционной системы.

На самом деле существует по крайней мере два документированных способа изменения кода приложений, одинаково хорошо работающих как под управлением Windows 95/Windows 98/Windows Me, так и под Windows NT/Windows 2000 и вполне удовлетворяющих привилегиями гостевого пользователя.

Во-первых, *kernel32.dll* экспортирует функцию *WriteProcessMemory*, предназначенную, как и следует из ее названия, для модификации памяти процесса. Во-вторых, практически все операционные системы, включая Windows и LINUX, разрешают выполнение и модификацию кода, размещенного **в стеке**.

В принципе задача создания самомодифицирующегося кода может быть решена исключительно средствами языков высокого уровня, таких, например, как Си, Си++, Паскаль без применения ассемблера.

Материал, изложенный в настоящей главе, большей частью ориентирован на компилятор Microsoft Visual C++ и 32-разрядный исполняемый код. Под Windows 3.x приведенные примеры работать не будут. Но это вряд ли представляет существенную проблему — доля машин с Windows 3.x на рынке очень невелика, поэтому ими можно полностью пренебречь.

Архитектура памяти Windows

Создание самомодифицирующегося кода требует знания некоторых тонкостей архитектуры Windows, не очень-то хорошо освещенных в документации, точнее, совсем не освещенных, но от этого отнюдь не приобретающих статус «недокументированных особенностей», поскольку, во-первых, они одинаково реализованы на всех Windows-платформах, а во-вторых, их активно использует компилятор Visual C++ от Microsoft. Отсюда следует, что никаких изменений даже в отдаленном будущем компания не планирует; в противном случае код, сгенерированный этим компилятором, откажет в работе, а на это Microsoft не пойдет (вернее, **не должна** пойти, если верить здравому смыслу).

Для адресации четырех гигабайтов виртуальной памяти, выделенной в распоряжение процесса, Windows используют два селектора, один из которых загружается в сегментный регистр *CS*, а другой — в регистры *DS*, *ES* и *SS*. Оба селектора ссылаются на один и тот же базовый адрес памяти, равный нулю, и имеют идентичные лимиты, равные четырем гигабайтам. (**Замечание:** помимо перечисленных сегментных регистров, Windows еще использует и регистр *FS*, в который загружает селектор сегмента, содержащего информационный блок потока — TIB.)

Фактически существует всего **один** сегмент, вмещающий в себя и код, и данные, и стек процесса. Благодаря этому передача управления коду, расположенному в стеке, осуществляется близким (*near*) вызовом или переходом, и для доступа к содержимому стека использование префикса *SS* совершенно необязательно. Несмотря на то что значение регистра *CS* **не равно** значению регистров *DS*, *ES* и *SS*, команды `MOV dest, CS:[src]`; `MOV dest, DS:[src]` и `MOV dest, SS:[src]` в действительности обращаются к одной и той же ячейке памяти.

Отличия между регионами кода, стека и данных заключаются в атрибутах принадлежащих им страниц — страницы кода допускают *чтение* и *исполнение*, страницы данных — *чтение* и *запись*, а стека — *чтение*, *запись* и *исполнение* одновременно.

Помимо этого каждая страница имеет специальный флаг, определяющий уровень привилегий, необходимых для доступа к этой странице. Некоторые страницы, например те, что принадлежат операционной системе, требуют наличия прав супервизора, которыми обладает только код нулевого кольца. Прикладные про-

граммы, исполняющиеся в кольце 3, таких прав не имеют и при попытке обращения к защищенной странице порождают исключение.

Манипулировать атрибутами страниц, равно как и ассоциировать страницы с линейными адресами, может только операционная система или код, исполняющийся в нулевом кольце. В защите Windows 95/Windows 98 имеются люки, позволяющие прикладному коду повысить свои привилегии до супервизора, но выгода от их использования сомнительна, поскольку «привязывает» пользователя к этой операционной системе и не дает возможности проделать тот же трюк на Windows NT/Windows 2000.

***Замечание:** среди начинающих программистов ходит совершенно нелепая байка о том, что, дескать, если обратиться к коду программы командой, предваренной префиксом DS, Windows якобы беспрепятственно позволит его изменить. На самом деле это в корне неверно — обратиться-то она позволит, а вот изменить — нет, каким бы способом ни происходило обращение, так как защита работает на уровне физических страниц, а не логических адресов.*

Использование функции WriteProcessMemory

Если требуется изменить некоторое количество байтов своего (или чужого) процесса, самый простой способ сделать это — вызвать функцию *WriteProcessMemory*. Она позволяет модифицировать существующие страницы памяти, чей флаг супервизора не взведен, т. е. все страницы, доступные из кольца 3, в котором выполняются прикладные приложения. Совершенно бесполезно с помощью *WriteProcessMemory* пытаться изменить критические структуры данных операционной системы (например, *page directory* или *page table*) — они доступны лишь из нулевого кольца. Поэтому эта функция не представляет никакой угрозы для безопасности системы и успешно вызывается независимо от уровня привилегий пользователя (автору этих строк доводилось слышать утверждение, дескать, *WriteProcessMemory* требует прав отладки приложений, но это не так).

Процесс, в память которого происходит запись, должен быть предварительно открыт функцией *OpenProcess* с атрибутами доступа *PROCESS_VM_OPERATION* и *PROCESS_VM_WRITE*. Часто программисты, ленивые от природы, идут более коротким путем, устанавливая все атрибуты — *PROCESS_ALL_ACCESS*. И это вполне законно, хотя справедливо считается дурным стилем программирования.

Простейший пример использования функции *WriteProcessMemory* для создания самомодифицирующегося кода, приведен в листинге 227. Она заменяет инструкцию бесконечного цикла *JMP short \$-2* на условный переход *JZ \$-2*, который продолжает нормальное выполнение программы. Неплохой способ затруднить взломщику изучение программы, не правда ли? (Особенно если вызов *WriteMe* расположен не возле изменяемого кода, а помещен в отдельный поток; будет еще лучше, если модифицируемый код вполне естествен сам по себе и внешне не вызывает никаких подозрений, в этом случае хакер может долго блуждать в той ветке кода, которая при выполнении программы вообще не получает управления).

Листинг 227. Пример, иллюстрирующий использование функции WriteProcessMemory для создания самомодифицирующегося кода

```

int WriteMe(void *addr, int wb)
{
    HANDLE h=OpenProcess(PROCESS_VM_OPERATION|PROCESS_VM_WRITE,
                        true,GetCurrentProcessId());
    return WriteProcessMemory(h, addr,&wb,1,NULL);
}

int main(int argc, char* argv[])
{
    _asm {
        push 0x74 ; JMP -> > JZ
        push offset Here
        call WriteMe
        add esp,8
Here:    JMP short here
    }
    printf("#JMP SHORT $-2 was changed to JZ $-2\n");
    return 0;
}

```

Поскольку Windows для экономии оперативной памяти разделяет код между процессами, возникает вопрос: а что произойдет, если запустить вторую копию самомодифицирующейся программы? Создаст ли операционная система новые страницы или отошлет приложение к уже модифицируемому коду? В документации на Windows NT и Windows 2000 сказано, что они поддерживают копирование при записи (*copy on write*), т. е. автоматически дублируют страницы кода при попытке их модификации. Напротив, Windows 95 и Windows 98 **не поддерживают** такую возможность. Означает ли это то, что все копии самомодифицирующегося приложения будут вынуждены работать с **одними и теми же** страницами кода, что неизбежно приведет к конфликтам и сбоям?

Нет, и вот почему: несмотря на то что копирование при записи в Windows 95 и Windows 98 не реализовано, эту заботу берет на себя сама функция *WriteProcessMemory*, создавая копии всех модифицируемых страниц, распределенных между процессами. Благодаря этому самомодифицирующийся код одинаково хорошо работает как под Windows 95/Windows 98/Windows Me, так и под Windows NT/Windows 2000. Однако следует учитывать, что все копии приложения, модифицируемые **любым иным путем** (например, командой `mov` нулевого кольца), будучи запущенными под Windows 95/Windows 98, будут разделять одни и те же страницы кода со всеми вытекающими отсюда последствиями.

Теперь об ограничениях. Во-первых, использовать *WriteProcessMemory* разумно только в компиляторах, компилирующих в память или распаковщиках исполняемых файлов, а в защитах — несколько наивно. Мало-мальски опытный взломщик быстро обнаружит подвох, увидев эту функцию в таблице импорта. Затем он установит точку останова на вызов *WriteProcessMemory* и будет контролировать каждую операцию записи в память. А это никак не входит в планы работчика защиты!

Другое ограничение *WriteProcessMemory* заключается в невозможности создания новых страниц, ей доступны лишь уже существующие страницы. А как быть в том случае, если требуется выделить некоторое количество памяти, например, для кода, динамически генерируемого на лету? Вызов функций, управления кучей, таких, как *malloc*, не поможет, поскольку в куче выполнение кода запрещено. И вот тогда-то на помощь приходит возможность выполнения кода в стеке...

Выполнение кода в стеке

Разрешение на выполнение кода в стеке объясняется тем, что исполняемый стек необходим многим программам, в том числе и самой операционной системе, для выполнения некоторых системных функций. Благодаря этому упрощается генерация кода компиляторами и компилирующими интерпретаторами.

Однако вместе с этим увеличивается и потенциальная угроза атаки. Если выполнение кода в стеке разрешено и ошибки реализации при определенных обстоятельствах приводят к передаче управления на данные, введенные пользователем, злоумышленник получает возможность передать и выполнить на удаленной машине свой собственный зловредный код. Для операционных систем Solaris и Linux существуют «заплатки», установка которых приводит к запрету исполнения кода в стеке, но они не имеют большого распространения, поскольку делают невозможной работу множества программ, и большинству пользователей легче смириться с угрозой атаки, чем остаться без необходимых приложений.

Поэтому использование стека для выполнения самомодифицирующегося кода вполне законно и системно независимо, т. е. универсально. Помимо этого, такое решение устраняет оба недостатка функции *WriteProcessMemory*:

Во-первых, выявлять и отслеживать команды, модифицирующие заранее неизвестную ячейку памяти, чрезвычайно трудно, и взломщику придется провести кропотливый анализ кода защиты без надежды на скорый успех (при условии, что сам защитный механизм реализован без грубых ошибок, облегчающих задачу хакера).

Во-вторых, приложение в любой момент может выделить столько стековой памяти, сколько ему заблагорассудится, а затем, при исчезновении потребности, ее освободить. По умолчанию система резервирует один мегабайт стекового пространства, а если этого для решения поставленной задачи недостаточно, нужное количество можно указать при компоновке программы.

Замечательно, что для программ, выполняющихся в стеке, справедлив принцип фон Неймана — в один момент времени текст программы может рассматриваться как данные, а в другой — как исполняемый код. Именно это необходимо для нормальной работы всех распаковщиков и расшифровщиков исполняемого кода.

Однако программирование кода, выполняющегося в стеке, имеет ряд специфических особенностей, о которых и будет рассказано ниже.

«Подводные камни» перемещаемого кода

При разработке кода, выполняющегося в стеке, следует учитывать, что в операционных системах Windows 9x, Windows NT и Windows 2000 местоположение стека различно и, чтобы сохранить работоспособность при переходе от одной системы к другой, код должен быть безразличен к адресу, по которому он будет загружен. Такой код называют *перемещаемым*, и в его создании нет ничего сложного, достаточно следовать нескольким простым соглашениям, вот и все.

Замечательно, что у микропроцессоров серии Intel 80x86 все короткие переходы (*short jump*) и близкие вызовы (*near call*) *относительны*, т. е. содержат не линейный целевой адрес, а разницу целевого адреса и адреса следующей выполняемой инструкции. Это значительно упрощает создание перемещаемого кода, но вместе с этим накладывает на него некоторые ограничения.

Что произойдет, если следующую функцию `void Demo() { printf(«Demo\n»); }` скопировать в стек и передать ей управление? Поскольку инструкция *call*, вызывающая функцию *printf*, «переехала» на новое место, разница адресов вызываемой функции и следующей за *call* инструкции станет совсем иной и управление получит отнюдь не *printf*, а не имеющий к ней никакого отношения код! Вероятнее всего, им окажется «мусор», порождающий исключение с последующим аварийным закрытием приложения.

Программируя на ассемблере, такое ограничение можно легко обойти, используя регистровую адресацию. Перемещаемый вызов функции *printf* упрощенно может выглядеть, например, так: `lea eax, printf\ncall eax`. В регистр *eax* (или любой другой регистр общего назначения) заносится абсолютный линейный, а не относительный адрес, и независимо от положения инструкции *call* управление будет передано функции *printf*, а не чему-то еще.

Однако такой подход требует знания ассемблера, поддержки компилятором ассемблерных вставок и не очень-то нравится прикладным программистам, не интересующимся командами и устройством микропроцессора.

Для решения данной задачи исключительно средствами языка высокого уровня необходимо передать стековой функции указатели на вызываемые ее функции как аргументы. Это несколько неудобно, но более короткого пути, по-видимому, не существует. Простейшая программа, иллюстрирующая копирование и выполнение функций в стеке, приведена в листинге 228.

Листинг 228. Программа, иллюстрирующая копирование и выполнение функции в стеке

```
void Demo(int (*_printf) (const char *,...))
{
    _printf("Hello, Word!\n");
    return;
}

int main(int argc, char* argv[])
{
    char buff[1000];
```

```
int (*_printf) (const char *,...);
int (*_main) (int, char **);
void (*_Demo) (int (*) (const char *,...));
_printf=printf;

int func_len = (unsigned int) _main - (unsigned int) _Demo;
for (int a=0;a<func_len;a++)
    buff[a]= ((char *) _Demo)[a];
_Demo = (void (*) (int (*) (const char *,...))) &buff[0];

_Demo(_printf);
return 0;
}
```

Елей и деготь оптимизирующих компиляторов

Применяя языки высокого уровня для разработки выполняемого в стеке кода, следует учитывать особенности реализаций используемых компиляторов и, прежде чем останавливать свой выбор на каком-то одном из них, основательно изучить прилагаемую к ним документацию. В большинстве случаев код функции, скопированный в стек, с первой попытки запустить не получится, особенно если включены опции оптимизированной компиляции.

Так происходит потому, что на **чистом** языке высокого уровня, таком, как Си или Паскаль, скопировать код функции в стек (или куда-то еще) принципиально невозможно, поскольку стандарты языка не оговаривают, каким именно образом должна осуществляться компиляция. Программист может получить указатель на функцию, но стандарт не оговаривает, как следует ее интерпретировать, с точки зрения программиста, она представляет «магическое число», в назначение которого посвящен один лишь компилятор.

К счастью, логика кодогенерации большинства компиляторов более или менее одинакова, и это позволяет прикладной программе сделать некоторые предположения об организации откомпилированного кода.

В частности, программа, приведенная в листинге 228, молчаливо полагает, что указатель на функцию совпадает с точкой входа в эту функцию, а все тело функции расположено непосредственно за точкой входа. Именно такой код (наиболее очевидный с точки зрения здравого смысла) и генерирует подавляющее большинство компиляторов. Большинство, но не все! Тот же Microsoft Visual C++ в режиме отладки вместо функций вставляет «переходники», а сами функции размещает совсем в другом месте. В результате в стек копируется содержимое «переходника», но не само тело функции!

Заставить Microsoft Visual C++ генерировать «правильный» код можно сбросом флажка *Link incrementally*. У других компиляторов название этой опции может значительно отличаться, а в худшем случае — вообще отсутствовать. Если это так, придется отказаться либо от самомодифицирующегося кода, либо от данного компилятора.

Еще одна проблема: как достоверно определить длину тела функции? Язык Си не дает никакой возможности узнать значение этой величины, а оператор

sizeof возвращает размер указателя на функцию, но не размер самой функции. Одно из возможных решений опирается на тот факт, что компиляторы, **как правило**, располагают функции в памяти согласно порядку их объявления в исходной программе, следовательно, длина тела функции равна разности указателей на следующую за ней функцию и указателя на данную функцию. Поскольку Windows-компиляторы представляют указатели 32-разрядными целыми числами, их можно безболезненно преобразовывать в тип unsigned int и выполнять над ними различные математические операции. К сожалению, оптимизирующие компиляторы не всегда располагают функции в таком простом порядке, а в некоторых случаях даже «разворачивают» их, подставляя содержимое функции на место вызова. Поэтому соответствующие опции оптимизации (если они есть) придется отключить.

Другое коварство оптимизирующих компиляторов заключается в выкидывании ими всех, не используемых (с их точки зрения) переменных. Например, в программе, приведенной в листинге 228, в буфер buff что-то пишется, но ничто оттуда не читается! А передачу управления на буфер большинство компиляторов (в том числе и Microsoft Visual C++) распознать не в силах, вот они и опускают копирующий код, отчего происходит передача управления на неинициализированный буфер с очевидными последствиями. Если возникнут подобные проблемы, попробуйте сбросить флажок Global optimization, а лучше отключите оптимизацию вообще (плохо, конечно, но надо).

Откомпилированная программа по-прежнему не работает? Вероятнее всего, причина в том, что компилятор вставляет в конец каждой функции вызов процедуры, контролирующей состояние стека. Именно так ведет себя Microsoft Visual C++, помещая в отладочные проекты вызов функции `__chkesp` (не ищите ее описания в документации — его там нет). А вызов этот, как нетрудно догадаться, относительный! К сожалению, никакого документированного способа это запретить, по-видимому, не существует, но в финальных (release) проектах Microsoft Visual C++ не контролирует состояние стека при выходе из функции, и все работает нормально.

Самомодифицирующийся код как средство защиты приложений

И вот после стольких мытарств и ухищрений злополучный пример запущен и победно выводит на экран «Hello, World!». Резонный вопрос: а зачем, собственно, все это нужно? Какая выгода от того, что функция будет исполнена в стеке? Ответ: код функции, исполняющееся в стеке, можно прямо на лету изменять, например, расшифровывать ее.

Шифрованный код чрезвычайно затрудняет дизассемблирование и усиливает стойкость защиты, а какой разработчик не хочет уберечь свою программу от хакеров? Разумеется, одна лишь шифровка кода не очень-то серьезное препятствие для взломщика, снабженного отладчиком или продвинутым дизассемблером, на-

подобие IDA Pro, но антиотладочные приемы (а они существуют, и притом в изобилии) — тема отдельного разговора, выходящего за рамки настоящей статьи.

Простейший алгоритм шифрования заключается в последовательной обработке каждого элемента исходного текста операцией «ИЛИ-исключающее-И» (XOR). Повторное применение XOR к шифротексту позволяет вновь получить исходный текст.

Следующий пример (см. листинг 229) читает содержимое функции Demo, зашифровывает его и записывает полученный результат в файл.

Листинг 229. Шифрование функции Demo

```
void _build()
{
    FILE *f;
    char buff[1000];
    void (*_Demo) (int *) (const char *,...);
    void (*_Build) ();
    _Demo=Demo;
    _Build=_build;

    int func_len = (unsigned int) _Build - (unsigned int) _Demo;
    f=fopen("Demo32.bin","wb");
    for (int a=0;a<func_len;a++)
        fputc(((int) buff[a]) ^ 0x77,f);
    fclose(f);
}
```

Теперь из исходного текста программы функцию *Demo* можно удалить, взамен этого разместив ее зашифрованное содержимое в строковой переменной (впрочем, не обязательно именно строковой). В нужный момент оно может быть расшифровано, скопировано в локальный буфер и вызвано для выполнения. Один из вариантов реализации приведен в листинге 230.

Обратите внимание, как функция *printf* в листинге 228 выводит приветствие на экран. На первый взгляд ничего необычного, но задумайтесь, *где* размещена строка «Hello, World!». Разумеется, не в сегменте кода — там ей не место (хотя некоторые компиляторы фирмы Borland помещают ее именно туда). Выходит, в сегменте данных, там, где ей и положено быть? Но если так, то одного лишь копирования тела функции окажется явно недостаточно, придется скопировать и саму строковую константу. А это утомительно. Но существует и другой способ — создать локальный буфер и инициализировать его по ходу выполнения программы, например, так: `...buff[666]; buff[0]='H'; buff[1]='e'; buff[2]='l'; buff[3]='l';buff[4]='o', ...` — не самый короткий, но, ввиду своей простоты, широко распространенный путь.

Листинг 230. Зашифрованная программа

```
int main(int argc, char* argv[])
{
    char buff[1000];
    int (*_printf) (const char *,...);
```

```

void (*_Demo) (int (*) (const char *,...));
char code[ ]=""\x22\xFC\x9B\xF4\x9B\x67\xB1\x32\x87\
\x3F\xB1\x32\x86\x12\xB1\x32\x85\x1B\xB1\
\x32\x84\x1B\xB1\x32\x83\x18\xB1\x32\x82\
\x5B\xB1\x32\x81\x57\xB1\x32\x80\x20\xB1\
\x32\x8F\x18\xB1\x32\x8E\x05\xB1\x32\x8D\
\x1B\xB1\x32\x8C\x13\xB1\x32\x8B\x56\xB1\
\x32\x8A\x7D\xB1\x32\x89\x77\xFA\x32\x87\
\x27\x88\x22\x7F\xF4\xB3\x73\xFC\x92\x2A\
\xB4";

_printf=printf;
int code_size=strlen(&code[0]);
strcpy(&buff[0],&code[0]);

for (int a=0;a<code_size;a++)
    buff[a] = buff[a] ^ 0x77;
_Demo = (void *) (int *) (const char *,...)) &buff[0];
_Demo(_printf);
return 0;
}

```

Теперь (см. листинг 230) даже при наличии исходных текстов алгоритм работы функции *Demo* будет представлять загадку! Этим обстоятельством можно воспользоваться для сокрытия некоторой критической информации, например процедуры генерации ключа или проверки серийного номера.

Проверку серийного номера желательно организовать так, чтобы даже после расшифровки кода ее алгоритм представлял бы головоломку для хакера. Один из примеров такого алгоритма предложен ниже.

Суть его заключается в том, что инструкция, отвечающая за преобразование бит, динамически изменяется в ходе выполнения программы, а вместе с ней соответственно изменяется и сам результат вычислений.

Поскольку при создании самомодифицирующегося кода требуется точно знать, в какой ячейке памяти какой байт расположен, приходится отказываться от языков высокого уровня и прибегать к ассемблеру.

С этим связана одна проблема: чтобы модифицировать такой-то байт, инструкции *mov* требуется передать его абсолютный линейный адрес, а он, как было показано выше, заранее неизвестен. Однако его можно узнать непосредственно в ходе выполнения программы. Наибольшую популярность получила конструкция *CALL \$+5\POP reg\mov [reg+relative_addres], xxM*, т. е. вызова следующей инструкцией *call* команды и извлечению из стека адреса возврата — абсолютного адреса этой команды, который в дальнейшем используется в качестве базы для адресации кода стековой функции. Вот, пожалуй, и все премудрости.

Листинг 231. Процедура генерации серийного номера, предназначенная для выполнения в стеке

MyFunc:

```

push    esi                ; сохранение регистра esi в стеке
mov     esi, [esp+8]       ; ESI = &username[0]
push    ebx                ; сохранение прочих регистров в стеке

```

```

        push    ecx
        push    edx
        xor     eax, eax          ; обнуление рабочих регистров
        xor     edx, edx

RepeatString:                                ; цикл обработки строки байт за байтом

        lodsb                     ; читаем очередной байт в AL
        test    al, al            ; ?достигнут конец строки
        jz      short Exit

; Значение счетчика для обработки одного байта строки.
; Значение счетчика следует выбирать так, чтобы с одной стороны все биты
; полностью перемешались, а с другой – была обеспечена четность (нечетность)
; преобразований операции xor
        mov     ecx, 21h

RepeatChar:
        xor     edx, eax          ; циклически меняется с xor на adc
        ror     eax, 3
        rol     edx, 5
        call    $+5                ; ebx = eip
        pop     ebx                ; /
        xor     byte ptr [ebx-0Dh], 26h ; Эта команда обеспечивает цикл.
                                           ; Изменение инструкции xor на adc

        loop    RepeatChar
        jmp     short RepeatString

Exit:
        xchg    eax, edx          ; результат работы (ser.num) в eax
        pop     edx                ; восстановление регистров
        pop     ecx
        pop     ebx
        pop     esi
        retn                       ; возврат из функции

```

Приведенный алгоритм интересен тем, что повторный вызов функции с передачей тех же самых аргументов может возвращать либо тот же самый, либо совершенно другой результат: если длина имени пользователя нечетна, то при выходе из функции XOR меняется на ADC с очевидными последствиями. Если же длина имени четна — ничего подобного не происходит.

Разумеется, стойкость предложенной защиты относительно невелика. Однако она может быть значительно усилена. На то существует масса хитрых приемов программирования — динамическая асинхронная расшифровка, подстановка результатов сравнения вместо коэффициентов в различных вычислениях, помещение критической части кода непосредственно в ключ и т. д.

Но назначение статьи состоит не в том, чтобы предложить готовую к употреблению защиту (да и зачем? Чтобы хакерам ее было бы легче изучать?), а доказать (и показать!) принципиальную возможность создания самомодифицирующегося кода под управлением Windows 95/Windows NT/Windows 2000. Как именно предоставленной возможностью можно воспользоваться — надлежит решать читателю.

Пара слов в заключение

Многие считают использование самомодифицирующегося кода «дурным» примером программирования, обвиняя его в отсутствии переносимости, плохой совместимости с различными операционными системами, необходимости обязательных обращений к ассемблеру и т. д. С появлением Windows 95/Windows NT этот список пополнился еще одним умозаключением, дескать, «самомодифицирующийся код — только для MS-DOS, в нормальных же операционных системах он невозможен (и поделом!)».

Как показывает настоящая глава, все эти притязания, мягко выражаясь, неверны. Другой вопрос: так ли необходим самомодифицирующийся код и можно ли без него обойтись? Низкая эффективность существующих защит (обычно программы ломаются быстрее, чем успевают дойти до легального потребителя) и огромное количество программистов, стремящихся «топтанием клавиш» заработать себе на хлеб, свидетельствует в пользу необходимости усиления защитных механизмов любыми доступными средствами, в том числе и рассмотренным выше самомодифицирующимся кодом.

Приглашение к дискуссии или новые приемы защиты

Многочисленные критические нападки — неизбежный удел всякой новой концепции.

Г. Селье. От мечты к открытию

В заключение книги автору хотелось бы поделиться собственным опытом создания защит, сломать которые принципиально невозможно. Точнее, их взлом потребовал бы многих тысяч, а то и миллионов лет на типичном бытовом компьютере (во всяком случае, очень хочется на это надеяться).

Гарантированно воспрепятствовать анализу кода позволяет только шифрование программы. Но сам процессор не может непосредственно исполнять зашифрованный код, поэтому перед передачей управления его необходимо расшифровать. Если ключ содержится внутри программы, стойкость такой защиты близка к нулю. Все, чего может добиться разработчик, — затруднить поиск и получение этого ключа, тем или иным способом препятствуя отладке и дизассемблированию программы.

Другое дело, если ключ содержится вне программы. Тогда стойкость защиты определяется стойкостью используемого криптоалгоритма (при условии, что ключ перехватить невозможно). В настоящее время опубликованы и детально описаны многие криптостойкие шифры, взлом которых заведомо недоступен рядовым злоумышленникам.

В общих чертах идея защиты заключается в описании алгоритма с помощью некой математической модели, одновременно с этим используемой для генерации

ключа. Разные ветви программы зашифрованы различными ключами, и, чтобы вычислить этот ключ, необходимо знать состояние модели на момент передачи управления соответствующей ветви программы. Код динамически расшифровывается в процессе его выполнения, а чтобы расшифровать его целиком, нужно последовательно перебрать все возможные состояния модели. Если их число будет очень велико (чего нетрудно добиться), восстановить весь код станет практически невозможно!

Для реализации этой идеи автором был создан специальный событийно-ориентированный язык программирования. События в нем представляют собой единственное средство вызова подпрограммы. Каждое событие имеет свой код и один (или несколько) аргументов. Событие может иметь какое угодно количество обработчиков, а может не иметь ни одного (в этом случае вызываемому коду возвращается ошибка).

На основе кода события и значения аргументов менеджер событий генерирует три ключа: первый только на основе кода события, второй только на основе аргументов и третий на основе кода и аргументов (см. пояснение 1). Затем он пытается полученными ключами последовательно расшифровать всех обработчиков событий. Если расшифровка происходит успешно, это означает, что данный обработчик готов обработать данное событие, и тогда ему передается управление.

Алгоритм шифрования должен быть выбран так, чтобы обратная операция была невозможна, при этом установить, какое событие данный обработчик обрабатывает, можно только полным перебором. Для блокирования возможности перебора в язык была введена контекстная зависимость — генерация дополнительной серии ключей, учитывающих некоторое количество предыдущих событий. Это позволило устанавливать обработчики на любые последовательности действий пользователя, например, на открытие файла с именем «Мой файл», запись в него строки «Моя строка» и переименование его в «Не мой файл».

Очевидно, что перебор комбинаций всех событий со всеми возможными аргументами займет бесконечное время и принципиально невозможен. Восстановить исходный код программы, защищенной таким образом, удастся не раньше, чем все ее ветви хотя бы однократно получат управление. Но частота вызова различных ветвей не одинакова, и у некоторых из них очень мала. Например, можно установить на слово «сосна», введенное в текстовом редакторе, свой обработчик, выполняющий некоторые дополнительные проверки на целостность кода программы или на лицензионную чистоту используемого ПО.

Взломщик не сможет быстро выяснить, до конца ли взломана программа, или нет. Ему придется провести тщательное и кропотливое тестирование, но даже после этого он не будет в этом уверен!

Таким же точно образом осуществляется ограничение срока службы демонстрационных версий. Разумеется, обращаться к часам реального времени бесполезно, их очень легко перевести назад, вводя защиту в заблуждение. Гораздо надежнее опираться на даты открываемых файлов — даже если часы переведены, созданные другими пользователями файлы в большинстве случаев имеют правильное время. Но взломщик не сможет узнать ни алгоритм определения даты, ни саму дату окончания использования продукта! Впрочем, дату в прин-

ципе можно найти и полным перебором, но что это дает? Модификации кода воспрепятствовать очень легко, достаточно, чтобы длина зашифрованного текста была чувствительна к любым изменениям исходного. В этом случае взломщик не сможет подправить «нужный» байт в защитном обработчике и вновь зашифровать его. Придется расшифровывать и вносить изменения во все остальные обработчики (при условии, что они контролируют смещение, по которому расположены), а это невозможно, так как соответствующие им ключи заранее неизвестны.

Существенными недостатками предлагаемого решения являются низкая производительность и высокая сложность реализации. Если со сложностью реализации можно смириться, то производительность налагает серьезные ограничения на сферу его применения. Впрочем, можно значительно оптимизировать алгоритм или оставить все критичные к быстродействию модули незашифрованными (или расшифровывать каждый обработчик только один раз), словом, дорогу осилит идущий! Интересно другое — действительно ли эта технология позволяет создавать принципиально неизучаемые приложения или в приведенные рассуждения вкралась ошибка? Было бы очень интересно выслушать мнения коллег, специализирующихся на защите информации.

Пояснение 1

Три ключа были необходимы для отказа от явной проверки значения аргументов, которую легко обнаружить анализирующему лицу. Например, пусть событие KEY (key_code) генерируется при каждом нажатии на клавиатуру. Тогда обработчик, считывающий входную информацию, должен привязываться только к коду события (KEY) и получать введенный символ в виде аргумента.

Если одна из клавиш (или комбинаций клавиш) зарезервирована для специальной цели (например, задействует некоторые дополнительные функции в программе), то ее обработчик может привязываться одновременно к коду события (KEY) и коду клавиши (key_code), не опасаясь за свое раскрытие, так как правильный ключ дает лишь единственная комбинация KEY и key_code, а явная проверка на соответствие нажатого символа секретному коду отсутствует.

Привязка к аргументам позволяет отлавливать искомые последовательности в потоке данных независимо от того, каким образом они получены. Например, процедура аутентификации, ожидающая пароля MyGoodPassword, не интересуется, введен ли он с клавиатуры, получен ли с удаленного терминала, загружен ли из файла и т. д.

Такой подход значительно упрощает программирование и уменьшает зависимость одних модулей от других. Программа представляет собой совокупность обработчиков, автоматически коммутируемых возникающими событиями. Никакого детерминизма! Это чем-то напоминает взаимодействие биологической клетки с окружающей средой и в скором будущем может стать довольно перспективным направлением.

Приложение

Ошибки Джеффри Рихтера

И на солнце есть пятна.
Народная китайская мудрость

Монография Джеффри Рихтера «*Windows для профессионалов*» — один из лучших (а может быть, и **самый лучший**) учебник по программированию, настольная книга многих Windows-разработчиков (в том числе и автора данной книги). Это самое полное, проработанное и систематизированное описание Win32 API, написанное живым, легко доступным языком, без излишнего занудства и воды.

Дифирамбы дифирамбами, но каким бы непререкаемым авторитетом Рихтер ни был, а ошибки есть и у него. Не то чтобы они сильно портили книгу, но все же... Словом, в этой главе речь пойдет именно о них. Автор не берется утверждать, что выловил все ошибки, но вот взгляните на то, что удалось обнаружить при вдумчивом чтении книги.

Номера страниц указаны по третьему изданию 1997 года — самому последнему, которое автору удалось приобрести. Возможно, даже наверняка, какая-то часть ошибок принадлежит не самому Рихтеру, а сотрудникам «Русской редакции», выполнивших ее перевод на русский язык.

К сожалению, автор не смог раздобыть оригинал и совершенно не представляет себе, как это сделать. Да и автор ли один? Ведь и подавляющее большинство читателей этого самого оригинала и в глаза не видело! К тому же цель не очернить Рихтера, а не дать ошибкам закрепиться в умах молодых программистов (запоминаются-то ошибки легко, а вот забываются куда труднее).

Итак...

Грубые ошибки автора

1) «Объекты ядра защищены, и процесс, прежде чем оперировать с ними, должен запрашивать разрешение на доступ к ним. Процесс — создатель объекта может предотвратить несанкционированный доступ к этому объекту со стороны другого процесса», с. 12.

Насчет защиты Рихтер немного загнул — она есть только под Windows NT, но даже там (за исключением серверных приложений) обычно не используется. Поэтому кто угодно может получить доступ к объектам ядра чужого процесса (за исключением системного), вызвав DuplicateHandle или обратившись к набору функций TOOLHELP32, — процесс и знать не будет, что дублирует его дескриптор!

И даже под NT, и даже с установленными атрибутами защиты в адресном пространстве процесса можно исполнить свой код, обращаясь к защищенному дескриптору от имени этого процесса. Делов-то!

Правильнее было бы говорить о защите от **непреднамеренного** доступа к дескрипторам чужого процесса.

2) «...согласно принципу неопределенности Гейзенберга, чем точнее определяется один квант, тем больше ошибка в измерении другого», с. 52.

Это не программистская, но все-таки грубая ошибка. Принцип Гейзенберга в пересказе автора данной книги звучит так: нельзя одновременно определить координаты и импульс **одной** частицы, поскольку любое измерение чего бы то ни было невозможно без взаимодействия, а любое такое взаимодействие искажает свойства объекта измерений.

3) «...потоки с более высоким приоритетом всегда вытесняют потоки с более низким приоритетом независимо от того, исполняются последние или нет», с. 65.

Нет, не исполняются. Во всяком случае, на однопроцессорной машине в каждый момент времени выполняется только один поток и, до тех пор пока не истечет отведенный ему квант времени, прервать ему некому.

Исключение составляют аппаратные прерывания, обрабатываемые системой, но это совсем другой разговор. А на многопроцессорных машинах за каждым процессором закрепляются «свои» потоки, и потоки одного процессора никогда не вытесняют потоки другого.

Правильно сказать так: а) потоки выполняются по очереди в согласии с приоритетом; б) при пробуждении потока он изменяет очередь исполнения, отбирая процессорное время у потоков с более низким приоритетом.

4) «Ни одна Win32-функция не возвращает уровень приоритета потока... Такая ситуация создана преднамеренно. Вспомните, что Microsoft может в любой момент изменить алгоритм распределения процессорного времени...», с. 71.

Неверно. Во-первых, явно пропущено слово «**абсолютный**», так как относительный приоритет автор сам только что получал функцией GetThreadPriority.

Во-вторых, абсолютный приоритет потока (далее по книге **базовый**) получается алгебраическим сложением с приоритетом процесса, возвращаемого функцией GetPriorityClass.

В-третьих, не надо путать незадокументированность «квантов» процессорного времени с классами приоритетов, значения которых задокументированы самой Microsoft.

5) «Резервируя регион в адресном пространстве, система обеспечивает еще четную кратность размера региона размеру **страницы**. Так называется единица объема памяти, используемая системой при управлении памятью», с. 86.

Бр-р... не понятно. Если проще — размер страницы всегда степень двойки, размер выделяемого региона всегда кратен размеру страниц, но **не обязательно**

должен быть четен количеству страниц. То есть запрос на выделение трех страниц выделит именно три страницы, а не четыре или две.

Небольшое уточнение — страничная организация памяти — прерогатива в первую очередь процессора, а не системы.

6) «*AllocationBase — идентифицирует базовый адрес региона, включающего в себя адрес, указанный в lpAddress*», с. 117.

Нет! В AllocationBase возвращается базовый адрес региона, ранее выделенного VirtualAlloc или 0, если регион был выделен как-то иначе или вообще не был выделен.

7) «*DLL-модулям куча по умолчанию не предоставляется, и поэтому при их компоновке нельзя применять параметр /HEAP*», с. 202.

По умолчанию DLL-модулям выделяется 1 Мб кучи, и его можно изменить ключом /HEAP. Соответствующее поле PE-заголовка послушно изменится, но... этой кучей динамической библиотеке воспользоваться так и не удастся, поскольку стандартный загрузчик ОС *всегда* игнорирует это поле при подключении DLL.

8) «*И последняя причина, по которой имеет смысл использовать в программе отдельные кучи, — локальный доступ... Обращаясь в основном к памяти, локализованной в небольшом диапазоне адресов, вы снизите вероятность перекачки страниц между оперативной памятью и страничным фреймом*», с. 204.

Это верно, но только по отношению к *физическим* адресам. Логически же удаленные друг от друга адреса могут ютиться и в смежных, и в далеко разнесенных страницах — это уж как ОС заблагорассудится их скомбинировать.

Если данные занимают размер, превышающий размер страницы (обычно 4 Кб), то за счет фрагментации виртуальной памяти они наверняка окажутся в несмежных страницах, а потому ожидаемое ускорение вылетит в трубу!

9) «*...поскольку в операционную систему встроена поддержка синхронизирующих объектов, **никогда** не применяйте этот метод (далее идет описание метода синхронизации с использованием переменной-флага, устанавливаемой в TRUE синхронизируемым потоком по завершению. — К. К)*», с. 217.

Во-первых, ввиду пропуска Рихтером ключевого слова **volatile**, предложенный им способ действительно *никогда* не следует использовать — работать он, скорее всего, не будет. Оптимизирующие компиляторы, увидев цикл `a la «while (!myvar)»`, подумают: раз переменная `myvar` явным образом не изменяется (во всяком случае, в рамках одного потока), так заменим ее константой и перепишем цикл как `while (1)`. Ключевое же слово **volatile** сообщает компилятору, что переменная может модифицироваться в любой момент времени внешним кодом и «оптимизировать» ее не надо. Между прочим, это камень преткновения очень многих начинающих программистов. Самое противное — прогон кода под отладчиком (отладочная версия обычно компилируется без оптимизации) работает на «ура», но финальная (оптимизированная) версия упорно не работает!

Во-вторых, не стоит совсем уж отказываться от ручной синхронизации потоков. Накручивать пустой цикл в ожидании результатов работы, конечно, глупо,

но вот если в это время заняться чем-нибудь другим, попутно периодически контролируя состояние переменной флага... А, собственно, почему это должен быть именно флаг?! Пусть один поток сообщает в этой переменной другому потоку процент выполненной им работы. Например загружая файл с дискеты, сети или другого медленного носителя, можно немедленно выводить скачанные данные на экран, если только один поток сообщит другому, какое именно количество на данный момент скачано.

10) *«Потом создавал буфер в адресном пространстве своего процесса и помещал в него машинный код, который выполнял такие операции... call LoadLibraryA... Все правильно, я сам брал машинные команды соответствующие каждой инструкции языка ассемблера, и заполнял ими буфер»*, с. 624.

Вот именно — **«команды, соответствующие каждой инструкции языка ассемблера»**, так как каждой инструкции ассемблера соответствует от одной до нескольких команд процессора. Не все они равнозначны, причем машинный код, сгенерированный всеми известными мне ассемблерами, **неперемещаем**, поскольку все вызовы в нем относительны, т. е. аргумент инструкции call представляет собой не смещение функции LoadLibraryA, а разницу ее смещения и смещения конца инструкции call. Поскольку адрес верхушки стека разнится от одной версии ОС к другой, созданный Рихтером код окажется работоспособен только в той ОС, для которой он предназначен, да и то лишь в том случае, если перед ассемблированием использовать директиву ORG xxx, где xxx — смещение начала буфера. (Рихтер об этом вообще ничего не говорит!)

Выходов два — либо формировать машинные команды «вручную», принудительно выбирая абсолютную адресацию (всякий ли знает, как это делать?), либо использовать регистровые вызовы, т. е. mov reg, offset LoadLibraryA.; call reg. Кстати, адрес LoadLibraryA у Рихтера константа, определяющаяся на этапе ассемблирования, но ведь она неодинакова в различных ОС!

11) *«...потом я изменил структуру CONTEXT... так, чтобы установить указатель стека на участок памяти перед моим машинным кодом, а указатель команд — на первый байт этого кода»*, с. 624.

Не совсем так — оба указателя должны быть установлены на начало машинного кода, так как стек растет в область меньших адресов и не может затереть код, лежащий после него.

12) *«Разрабатывая ThreadFunct, я должен постоянно помнить, что после копирования в удаленное адресное пространство функция будет находиться по виртуальному адресу, который почти наверняка не совпадет с ее адресом в локальном адресном пространстве. Это значит, надо написать функцию, не делающую внешних ссылок! Это очень трудно!»*, с. 632.

«Не делающую внешних ссылок» — не только литературно, но и технически некорректное выражение. Точнее:

а) все машинные команды этой функции для обращения к коду и переменным самой этой функции должны использовать только относительную адресацию;

б) для обращения к коду и переменным, не принадлежащим этой функции, — только абсолютную адресацию;

в) следует отказаться от статических или глобальных переменных, так как они размещаются компилятором в сегменте данных локального адресного пространства, но если это позарез необходимо вашей функции, поместите их в динамически выделяемую память (кучу).

Но и это еще не все! Многие компиляторы могут принудительно вставлять в код непереключаемые вызовы своих собственных функций. Например, Microsoft Visual C++ для контроля сбалансированности стека до и после вызова функции обращается к служебной процедуре `__chkesp`. Хорошо, если разработчики компилятора предусмотрели ключи, запрещающие подобную «самодеятельность», но так бывает не всегда.

Поэтому техника создания перемещаемой функции — тема не одного абзаца, а как минимум целой главы и рекомендаций Рихтера явно недостаточно для практического осуществления такого замысла.

13) «Флаг `FILE_FLAG_POSIX_SEMANTICS` сообщает, что при доступе к файлу следует применить правила `POSIX`. Файловые системы, использующие `POSIX`, чувствительны к регистру в именах файлов... В то же время `MS-DOS`, 16-разрядная `Windows` и `Win32` к регистру букв в именах файлов не чувствительны. Поэтому будьте крайне осторожны, используя `FILE_FLAG_POSIX_SEMANTICS`. Файл, при создании которого установлен этот флаг, может оказаться недоступным из приложений `MS-DOS`, 16-разрядной `Windows` и `Win32`», с. 472.

Во-первых, `Win32` тут явно «третий лишний» — если `Win32` поддерживает `POSIX` этим самым флагом, какие могут быть проблемы? Кстати, по поводу `POSIX` — его не поддерживает `FAT`, поэтому файл, созданный на `FAT`-диске, регистр игнорирует — создаваться-то с указанным регистром символов он создается, но вот возможности создания двух файлов с одинаковыми именами, но разными регистрами нет, помимо этого при открытии файла идентичность регистра не проверяется, даже если установлен `FILE_FLAG_POSIX_SEMANTICS`.

Другое отличие `POSIX` — обратный (ну, в смысле прямой) наклон черты разделителя, т. е. к файлу `TEST\test` доступ теперь осуществляется так: `TEST/test`.

Во-вторых, фраза «может оказаться недоступным» слишком витиевата, чтобы быть полезной. Почему бы не ответить, **когда именно** он оказывается недоступным? А вот когда. Если на `NTFS`-диске в одной директории содержится два и более файлов с одинаковыми именами, но разными регистрами, то из-под `Windows-16` и `MS-DOS` виден только первый (в порядке создания) из них. Во всех остальных случаях файл, созданный с флагом `FILE_FLAG_POSIX_SEMANTICS`, доступен отовсюду — можете не волноваться!

Неточности, недоговорки

1) «Почти все функции, создающие объекты ядра, принимают указатель на структуру `SECURITY_ATTRIBUTES` как аргумент... Большинство приложений вместо этого аргумента передают `NULL` и создают объект с защитой по умолчанию. Такая защита подразумевает, что администратор и

создатель объекта получают к нему полный доступ, а все прочие к объекту не допускаются», с. 9.

Гм-гм, выходит, если к объекту необходимо допускать всех остальных, как часто и бывает, придется явно инициализировать SECURITY_ATTRIBUTES? Конечно же нет! По умолчанию допускаются **все** пользователи со всеми полномочиями, будь то запись, чтение или еще что. Проверьте — создайте новый файл вызовом CreateFile, передав вместо атрибутов секретности NULL, и попытайтесь открыть его, войдя в систему под именем другого пользователя. Открывается? Вот и славненько!

2) *«...если вы создаете диалоговое окно, какой смысл формировать список одним потоком, а кнопку другим», с. 53.*

Смысл есть — пусть один (или несколько) поток, занятый, скажем, поиском файлов на диске, создает один (или несколько) элемент списка для вывода результатов своей работы, а кнопка «Стоп» их всех «срубает».

3) *«Как узнать, например, чьим объектом — User или ядра — является данный значок? ...проанализировать Win32-функцию, создающую объект. Практически у всех функций, создающих объекты ядра, есть параметр, позволяющий указать атрибуты защиты», с. 9.*

Не очень-то надежный способ! Вот у функции HINSTANCE LoadLibrary(LPCTSTR lpLibFileName) нет никаких атрибутов секретности, но описатель HINSTANCE принадлежит ядру. Почему? Да хотя бы уже потому, что ядро ее и экспортирует, о чем и рассказывается в SDK. Если под рукой нет SDK, на помощь приходит тот факт, что функция содержится в библиотеке kernel32.lib и, стало быть, «ядренная».

4) *«...по завершении процесса операционная система гарантированно освобождает все ресурсы, принадлежащие эту процессу», с. 12.*

...если только процесс не вызвал исключение, вызывающее его аварийное завершение. Именно поэтому приходится перегружать машину после очередного «зависания» того же, допустим, редактора Word, иначе при попытке открытия последнего редактируемого файла будет выдано сообщение: файл уже открыт другим процессом и работать с ним невозможно. Хороший программист должен предусмотреть такую ситуацию и принять адекватные меры по ее устранению.

5) *«Имейте в виду: **описатели** объектов наследуются, но **сами объекты** нет (курсив Рихтера)», с. 12.*

Ух ты! **Килограмм не длиннее литра**, да еще курсивом! Объекты ядра принадлежат ядру ОС, но не породившему их процессу, которому остается довольствоваться только описателями (дескрипторами) этих объектов. Поэтому о наследовании объектов ядра другими процессами говорить просто некорректно.

6) *«Первый и третий параметры функции DuplicateHandle представляют собой описатели объектов ядра, специфичные для вызывающего процесса», с. 18.*

Бр-р... ничего не понял! А вы, читатель? На самом деле эти параметры **описатели процессов** — процесса-источника и процесса-приемника (точнее, выражаясь терминологией самого же Рихтера — псевдоописатели).

7) *«Граница между двумя типами приложений (консольных и графических. — К. К) весьма условна. Можно, например, создать консольное приложение, способное отображать диалоговые окна...», с. 25.*

Тип приложения указывается в заголовке исполняемого файла и однозначно определяет механизм его загрузки и инициализации. Тот факт, что консольные приложения имеют доступ к GDI-функциям, а графические приложения могут создать консоли, не позволяет делать вывод об «условности» границ между обоими типами приложений.

8) *«Завершение потока», с. 60.*

К трем перечисленным Рихтером способам завершения потока (ExitThread; TerminateThread; завершение процесса, породившего поток) необходимо добавить и четвертый (кстати, самый популярный и простой из всех) — return. То есть возврат управления главной функции потока.

9) *«В Windows 95 все четыре описанные функции не предусмотрены. В ней не удастся даже загрузить программу, вызывающую любую из этих функций», с. 72.*

Да, но только если она загружает экспортирующую их DLL неявной компоновкой. Поэтому очень важно объяснить читателю, что API-функции, отсутствующие в Windows 95, настоятельно рекомендуется вызывать, явно загружая соответствующие им библиотеки и самостоятельно обрабатывая ситуации с отсутствием функций.

Вообще же в отношении функциональности Windows 95, Рихтер очень туманен и выражается то «приложение, использующее такие-то функции, не будет работать в Windows 95», то «приложение, использующее такие-то функции, вообще не удастся загрузить в Windows 95». Очень важно отличать отсутствие функций и отсутствие их реализаций. Первых — в Windows 95 вообще нет, вторые как будто-то есть, но при попытке вызова всегда возвращают ошибку. Рихтер, увы, не всегда различает эти два случая.

10) *«Любой поток может вызвать эту функцию (SuspendThread. — К. К) и приостановить выполнение другого потока. Хоть об этом нигде и не говорится (но я все равно скажу!), приостановить свое выполнение поток способен сам, а возобновить без посторонней помощи — нет... Поток допустимо задерживать не более чем М», с. 72.*

В этом маленьком абзаце сразу три ошибки. Первое — для приостановки другого потока его надо открыть с флагом THREAD_SUSPEND_RESUME, на что не у всех остальных потоков хватит прав, так как приостановить выполнение системных потоков очень проблематично (точнее, не прибегая к недокументированным секретам — невозможно).

Второе — приостановка осуществляется не только SuspendThread, но и массой функций, таких, как Sleep, WaitFor...

Третье — эти функции, в частности Sleep, позволяют потоку самостоятельно контролировать свое «засыпание» — «пробуждение». Ну, во всяком случае, без явного вызова ResumeThread другим потоком.

11) «Если система почему-либо не свяжет EXE-файл с необходимыми ему DLL-модулями, на экране появится соответствующее сообщение, а адресное пространство процесса и объект «процесс» освобождаются», с. 164.

Не могу удержаться, чтобы не заметить, что в Windows 2000 при запуске процесса из консольного приложения сообщение о неудачной загрузке DLL не появляется и процесс тихо «кончает», оставляя пользователя в недоумении, почему он не работает.

Поэтому теперь программисту недопустимо игнорировать результат успешности завершения CreateProcess и необходимо самостоятельно вызывать GetLastError для донесения до пользователя причины ошибки. Не стоит надеяться на операционную систему — отныне она это уже не делает.

12) «Семейство Interlocked функций», с. 312.

Описывая эти синхронизирующие функции, Рихтер упустил одно немаловажное обстоятельство — большинство компиляторов в большинстве случаев для приращения (уменьшения) значения переменной на единицу используют ассемблерные команды **inc [var]** и **dec[var]** соответственно. Они не могут быть прерванными на середине операции, и заботиться об их синхронизации незачем.

Не прерываются и операции сложения (вычитания) 32-разрядной переменной с 32-разрядной константой, а также все аналогичные битовые операции⁵.

Исключения:

а) 64-разрядные переменные;

б) известная «болезнь» ранних компиляторов от Borland — выполнение всех операций с переменными как минимум в три этапа: `mov reg,[var]\ ops reg, const\ mov [var], reg`;

в) сложные случаи адресации, разбиваемые компилятором на несколько стадий — вычисление эффективного адреса и приращение (уменьшение) переменной, расположенной по этому адресу.

13) «Если бы этот код выполнялся в Win32 приложении без блока *try-finally* и оно завершилось бы из-за неправильного доступа к памяти в *Funcinator*, семафор остался бы занят и не освободился — соответственно и ожидающие его потоки не получили бы процессорного времени», с. 522.

Постой, постой. Какие потоки? Если потоки самого процесса — так ведь они тихо скончались вместе с самим приложением, а если потоки других процессов — так ведь после завершения процесса семафор будет освобожден операционной системой. Так что принудительное освобождение семафора в этом случае — очевидное излишество.

14) «Это простейший способ внедрения DLL (добавления внедряемой DLL в ключ реестра `HKEY_LOCAL_MACHINE\Software\Microsoft\Windows NT\ CurrentVersion\ Windows\APPINIT_DLLS`. — К. К). Однако здесь

⁵ Зачастую сказанное остается верным и для 16- или 8- разрядных переменных, но некоторые компиляторы для увеличения быстродействия расширяют переменную до 32-разрядов, а потом обнуляют ее старшую часть. Естественно, содержимое переменной между этими операциями содержит мусор.

есть ряд недостатков... Ваша DLL будет спроецирована на адресное пространство лишь тех процессов, на которые отображен и USER32. А последнее делается только в GUI-приложениях, т. е. данный способ не подходит для программ консольного типа, — например, компиляторов или компоновщиков», с. 602.

В документации от Microsoft и в технических статьях сторонних авторов, содержащихся в том же MSDN, утверждается, что этот способ срабатывает для **всех** процессов системы. Сейчас проверил на Windows 2000 — действительно, внедряемая DLL послушно проецируется даже на консольные приложения.

Потом, неверно утверждение, что консольные приложения не используют USER32. Используют, да еще как! Чаще всего он им необходим для подачи сигналов вызовом функции MessageBeep, экспортируемой USER32.

Досадные описки и ляпы

1) «Флаг `HANDLE_FLAG_PROTECT_FROM_CLOSE` (передаваемый функции `SetHandleInformation`. — К. К) сообщает системе, что данный описать закрывать нельзя... Если какой-нибудь поток попытается закрыть защищенный описатель, `CloseHandle` приведет к исключению», с. 15.

Какое такое исключение? `CloseHandle` всего лишь вернет `NULL`, сигнализируя об ошибке...

2) «`DuplicateHandle(GetCurrentProcess, hObjProcessA, hObjProcessB, &hObjProcessB....)`», с. 21.

Опечатка — третий аргумент должен быть `hProcessB`.

3) «Кроме адресного пространства, процессу принадлежат такие ресурсы, как файлы...», с. 23.

Файл — объект ядра и принадлежит ядру, но не процессу.

4) «В Windows 95 функции `CreateFileMapping` можно передать флаг `PAGE_WRITECOPY...`», с. 170.

Пропущено слово «только», ибо других флагов Windows 95, увы, не поддерживает!

5) «`PAGE + WRITECOPY`», с. 174.

Досадная опечатка — конечно же должно быть `PAGE_WRITECOPY`.

6) «Обычно критические секции представляют собой набор глобальных переменных», с. 220.

Критические секции — не переменные! Структуры `CRITICAL_SECTION`, передаваемые им, действительно часто хранятся в глобальных переменных, но это дурной тон и гораздо лучше размещать их в куче или структуре данных, передаваемой синхронизируемым потокам через `lpvThreadParam`.

7) «Имена файлов и каталогов могут включать буквы разного регистра, но при поиске файлов и каталогов регистр букв не учитывается. Если файл с

именем *ReadMe.txt* уже существует, создание нового файла с именем *READ-ME.TXT* уже не допускается», с. 416.

Тут Рихтер противоречит сам себе — страницей назад от утверждал, что NTFS различает регистр символов, а главой вперед — чтобы заставить ее делать это, достаточно воспользоваться флагом `FILE_FLAG_POSIX_SEMANTICS`.

8) «...система создает для нового процесса виртуального адресное пространство размером 4 Гб и загружает в него код и данные как для исполняемого файла, так и для любых *DLL*», с. 36.

Не **загружает**, а **проецирует**. Разница принципиальна! Загрузка подразумевает считывание с диска и записи в память, но Windows поступает умнее — исполняемый файл и *DLL* трактуются как часть виртуальной памяти (о чем позднее сам же Рихтер и рассказывает в главе «Проецируемые в память файлы»).

Заключение

...изданий без ошибок не существует. Немало их и в оригинальной документации от самой Microsoft, равно как и от других компаний-производителей. Поэтому желательно выработать в себе железное правило — *не верить ничему*, каким бы логичным и естественным оно ни казалось. **Проверяйте!** Каждое слово, каждую запятую, каждую строчку примера — проверяйте!

Доверчивость читателей вполне простительна, но для авторов это — тяжкий грех. Если уж взялся за перо (тьфу ты, за клавиатуру), будь любезен, прежде чем печатать что-то, тщательно проверить, правильно ли это, или нет!

Содержание

ПРЕДИСЛОВИЕ РЕДАКТОРА.	3
ЧТО НОВОГО ВО ВТОРОМ ИЗДАНИИ.	4
О чем эта книга.	4
Кто такие хакеры	5
Чем мы будем заниматься	8
Что нам понадобится	9
ЗНАКОМСТВО С БАЗОВЫМИ ПРИЕМАМИ РАБОТЫ ХАКЕРА . . .	13
Введение	13
Классификация защит.	13
Философия стойкости.	16
Шаг первый. Разминочный	17
Шаг второй. Знакомство с дизассемблером	21
Шаг третий. Хирургический	26
Шаг четвертый. Знакомство с отладчиком	33
Способ 0. Бряк на оригинальный пароль	34
Способ 1. Прямой поиск введенного пароля в памяти	45
Способ 2. Бряк на функции ввода пароля.	53
Способ 3. Бряк на сообщения	55
Шаг пятый. На сцене появляется IDA	58
Шаг шестой. Дизассемблер & отладчик в связке	82
Шаг седьмой. Идентификация ключевых структур языков высокого уровня.	84
Идентификация функций	85
Идентификация стартовых функций	97
Идентификация виртуальных функций.	101
Идентификация конструктора и деструктора	124
Идентификация объектов, структур и массивов	133
Идентификация аргумента this	146
Идентификация операторов new и delete.	147
Идентификация библиотечных функций	150
Идентификация аргументов функций	156
Идентификация значения, возвращаемого функцией	216
Идентификация локальных стековых переменных.	252
Идентификация регистровых и временных переменных	266

Идентификация глобальных переменных	276
Идентификация констант и смещений	282
Идентификация литералов и строк	293
Идентификация условных операторов if-then-else	306
Идентификация операторов switch-case-break.	341
Идентификация циклов.	355
Идентификация математических операторов	386

СПОСОБЫ ЗАТРУДНЕНИЯ АНАЛИЗА ПРОГРАММ 402

Приемы против отладчиков	404
Немного истории	404
Как работает отладчик	405
Обработка исключений в реальном и защищенном режимах.	407
Как хакеры ломают программы	407
Как защитить свои программы	408
Как противостоять трассировке	409
Как противостоять контрольным точкам останова	413
Как обнаружить отладку средствами Windows	418
Приемы против дизассемблеров	419
Самомодифицирующийся код в современных операционных системах	419
Архитектура памяти Windows	420
Использование функции WriteProcessMemory	421
Выполнение кода в стеке.	423
«Подводные камни» перемещаемого кода	424
Елей и деготь оптимизирующих компиляторов	425
Самомодифицирующийся код как средство защиты приложений	426
Пара слов в заключение	430
Приглашение к дискуссии или новые приемы защиты.	430
Пояснение 1	431

ПРИЛОЖЕНИЕ 433

Ошибки Джеффри Рихтера	433
Грубые ошибки автора	433
Неточности, недоговорки.	437
Досадные описки и ляпы	441

ЗАКЛЮЧЕНИЕ 443

Крис Касперски

Фундаментальные основы хакерства

Техника дизассемблирования

Ответственный за выпуск

С. Иванов

Корректор

Н. Данилова

Макет и верстка

С. Тарасов

Обложка

Е. Жбанов

Издательство «СОЛОН-Р»

123242, г. Москва, а/я 20

Телефоны:

(095) 254-44-10, (095) 252-36-96, (095) 252-25-21

E-mail: Solon-R@coba.ru

ООО Издательство «СОЛОН-Р»

ЛР № 066584 от 14.05.99

Москва, ул. Тверская, д. 10, стр. 1, ком. 522

Формат 70×100/16. Объем 28 п. л. Тираж 2500

ООО «ПОЛИТЕХ-4»

Москва, Б. Переяславская, 46

Заказ №