Упаковщики исполняемых файлов в Linux/BSD

Крис Касперски

Большинство UNIX-программ распространяются в исходных текстах, но количество коммерческих продуктов с закрытым кодом неуклонно растет. Зачастую они распространяются в упакованном виде, что не только препятствует анализу, но снижает производительность и ухудшает совместимость с UNIX-клонами. Покажем на примере ELFCrypt, UPX, Burneye и Shiva, как можно освободиться от упаковщиков.

В Windows упаковщики исполняемых файлов получили очень большое распространение и сформировали обширную рыночную нишу, переваривающую огромные деньги, питающие целые фирмы и привлекающие высококвалифицированных специалистов, создающих нехилые защитные механизмы, борьба с которыми требует консолидации всего хакерского сообщества.

Под UNIX ситуация обстоит приблизительно так: потребность в упаковщиках уже есть (и многие коммерческие фирмы хотели бы выпустить закрытые порты своих продуктов под UNIX, основательно их защитив), но рынок протекторов еще не успел сформироваться, а потому разработкой упаковщиков занимается от силы десяток энтузиастов, повторяющих трюки времен ранней молодости MS-DOS, и только Shiva попытался предпринять качественный рывок вперед, вплотную приблизившись к протектору Software Passport (бывший Armadillo), однако, это его и погубило. На всех Linux/BSD-системах, до которых я только смог дотянуться, Shiva падает с сообщением «Segmentation fault». Какое же это счастье Windows-программистам иметь одну, ну пусть две (с учетом 9x) ОС, практически полностью совместимые между собой даже на уровне недокументированных возможностей!

Создание надежной защиты, запускающейся более чем на одной версии Linux, – практически безнадежное дело, а если вспомнить про BSD и экс-

Упаковщики и производительность

При запуске файла с дискеты или CD-ROM упаковка действительно ускоряет загрузку, поскольку физически передается существенно меньший объем данных, а скорость этих устройств несопоставима со скоростью процессора, поэтому временем распаковки можно полностью пренебречь и полученный выигрыш численно равен степени упаковки.

При запуске с жесткого диска все происходит с точностью до наоборот. Неупакованный elf проецируется непосредственно в оперативную память и в swap вытесняются только модифицированные страницы секции данных. При запуске нескольких экземпляров неупакованного elf-файла выделения новых страниц физической памяти практически не происходит, вместо этого операционная система просто отображает на адресное пространство процесса ранее загруженные страницы.

Если же файл упаковать, то при запуске он модифицирует всю свою проекцию целиком, а это значит, что при нехватке физической памяти операционная система уже не может «выкинуть» принадлежащие ему страницы, ведь возможности повторно загрузить их с диска уже нет и приходится заниматься вытеснением в swap. При однократном запуске программы это еще не так заметно, но многократный запуск упакованного файла приводит к существенному замедлению загрузки (ведь, вместо того, чтобы обратиться к уже загруженным страницам, операционной системе приходится заниматься распаковкой каждый раз). По той же причине растут потребности в оперативной памяти – несколько экземпляров упакованной программы не могут совместно использовать общие страницы физической памяти. В довершение ко всему большинство упаковщиков требуют дополнительной памяти для складывания промежуточных результатов распаковки. На утилитах, запускаемых огромное количество раз (например, make), разница между упакованным и неупакованным файлом просто колоссальна!

Отсюда вывод: с появлением жестких дисков и многозадачных операционных систем со страничной организацией памяти упаковка исполняемых файлов полностью утратила смысл и стала только вредить (она хорошо работала в эпоху господства MS-DOS, но те времена давно прошли). Теперь упаковывать (а точнее, зашифровывать) файл стоит только ради того, чтобы затруднить его анализ, да и то… стоит ли? Хакеры все равно взломают (ни один из существующих протекторов не избежал этой участи), а вот у легальных пользователей снижается производительность и появляются проблемы совместимости, тем более, что будущее все равно за открытым программным обеспечением. Как показывает практика, по мере взросления любая отрасль неизбежно приходит к открытым стандартам – взять хотя бы автомобилестроение или электронику. Лет тридцать назад японцы (в то время лидеры в этой области) закладывали в свои радиоприемники/магнитофоны ампулы с кислотой, чтобы при вскрытии корпуса все разъедало. Чуть позже для этой же цели стали применять эпоксидную смолу, а сейчас… принципиальные схемы раздаются всем сервисным центрам или отдаются под чисто формальное соглашение о неразглашении.

периментальные ядра типа Hurd, то к программированию можно даже не приступать. В то же время, слабость защитных механизмов компенсируется отсутствием достойного хакерского инструментария, поэтому даже простейшая защита представляет собой большую проблему, превращая распаковку программ в довольно нетривиальную задачу! Но мы ее решим! Начиная с самых простых упаковщиков и приобретая в сражении тактические навыки и необходимый инструментарий, в конечном счете, мы сможем сразиться с кем угодно!

ELF-Crypt

Простейший шифровщик (не упаковщик!) elf-файлов, созданный индийским студентом по прозвищу JunkCode (junkcode@yahoo.com) и распространяющийся в исходных текстах: http://www.infogreg.com/source-code/public-domain/elfcrypt-v1.0.html.

Он шифрует кодовую секцию (которой, как правило, является секция .text) и встраивает в elf-файл крохотный расшифровщик, возвращающий ее в исходный вид. Не содержит никаких антиотладочных приемов и замечательно распаковывается любым отладчиком, в том числе и gdb/ald (см. **листинг 1**).

Достаточно просто установить точку сразу же за концом расшифровщика (в данном случае она расположена по адресу 80495F9h), после чего в нашем распоряжении окажется расшифрованный elf, с которого можно снять дамп. В случае с gdb последовательность команд будет выглядеть приблизительно так (см. **листинг 2**).

Кстати говоря, последние версии IDA Pro, портированные под Linux, содержат интерактивный отладчик в стиле Turbo-Debugger, работающий через prtace() и позволяющий делать такие вещи прямо в дизассемблере! Но специально на этот случай JunkCode подложил хакерам большую свинью, сбивающую IDA Pro с толку.

Изменив точку входа в elf-файл (entrypoint) путем перенаправления ее на тело своего расшифровщика, он «забыл» скорректировать символьную метку _start, продолжающую

Листинг 1. Дизассемблерный листинг расшифровщика, внедряемого ELFCrypt в файл (как он выглядит в hiew)

```
:entrypoint
.080495DC: EB02          jmps     .0080495E0          ; переходим на расшифровщик
.080495DE: 06            push     es                  ; \ мусор, оставленный...
.080495DF: C6            ???                          ; / ...транслятором ассемблера
.080495E0: 60            pushad                       ; сохраняем все регистры в стеке
.080495E1: 9C            pushfd                       ; сохраняем флаги в стеке
.080495E2: BEC0820408    mov      esi, 0080482C0      ; начало расшифровываемого фрагмента
.080495E7: 8BFE          mov      edi, esi            ; EDI := EDI (расшифровка на месте)
.080495E9: B978000000    mov      ecx, 000000078      ; количество двойных слов для расшифровки
.080495EE: BBBD03CC09    mov      ebx, 009CC03BD      ; ключ расшифровки
.080495F3: AD            lodsd                        ; читаем очередной двойное слово <-----+
.080495F4: 33C3          xor      eax,ebx             ; расшифровываем через xor             |
.080495F6: AB            stosd                        ; записываем результат на место        |
.080495F7: E2FA          loop     .0080495F3          ; мотаем цикл ------------------------+
.080495F9: 9D            popfd                        ; восстанавливаем флаги из стека
.080495FA: 61            popad                        ; восстанавливаем все регистры
.080495FB: BDC0820408    mov      ebp, 0080482C0      ; адрес оригинальной точки входа (OEP)
.08049600: FFE5          jmp      ebp                 ; передаем управление расшифрован. коду
```

Листинг 2. Быстрая расшифровка elf-файла в отладчике gdb

```
; определяем точку входа в файл
root@5[elf_crypt]#objdump -f elfcrypt-demo

elfcrypt-demo:      формат файла elf32-i386
архитектура: i386, флаги 0x00000112:
EXEC_P, HAS_SYMS, D_PAGED
начальный адрес 0x080495dc

root@5[elf_crypt]# gdb elfcrypt-demo
; ставим точку останова на точку входа
(gdb) b *0x80495DC

Breakpoint 1 at 0x80495dc

; пускаем программу
(gdb) r

Starting program: /home/elf_crypt/elfcrypt-demo

; сработала точка останова

Breakpoint 1, 0x080495dc in ?? ()

; говорим отображать команды Ассемблера
(gdb) display/i $pc

1: x/i $pc  0x80495dc: jmp 0x80495e0

; начинаем трассировать программу
(gdb) si

0x080495e0 in ?? ()

; продолжаем трассировать

1: x/i $pc  0x80495e0:  pusha
...
0x080495f7 in ?? ()

; видим цикл

1: x/i $pc  0x80495f7:  loop   0x80495f3

; ставим точку останова за его концом
(gdb) b *0x80495F9

Breakpoint 2 at 0x80495f9

; запускаем программу «вживую»
(gdb) c

Continuing.

; точка останова достигнута

Breakpoint 2, 0x080495f9 in ?? ()

; программа расшифрована!

1: x/i $pc  0x80495f9:  popf
```

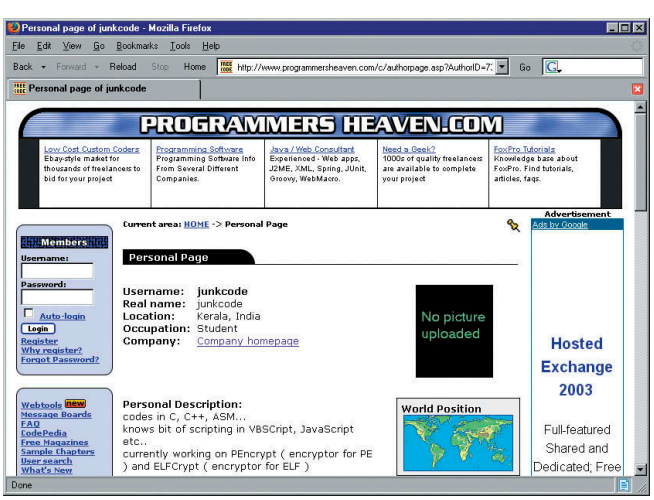

Рисунок 1. Домашняя страница создания JunkCode – создателя шифровщика ELFCrypt

указывать на оригинальную точку входа (в настоящий момент зашифрованную!), в результате чего IDA Pro показывает нечто совершенно бессмысленное:

Листинг 3. Нет, это не хитрый антиотладочный код, это просто оригинальная точка входа в программу, еще не расшифрованная расшифровщиком

```
.text:080482C0          _start   proc near
.text:080482C0 8C EE             mov      esi, gs
.text:080482C2 92                xchg     eax, edx
.text:080482C3 80 5C 80 28 F9    sbb      byte ptr ↵
     [eax+eax*4+28h], 0F9h
.text:080482C8 ED                in       eax, dx
.text:080482C9 57                push     edi
.text:080482CA 9E                sahf
.text:080482CB 61                popa
.text:080482CC AD                lodsd
.text:080482CD 87 C8             xchg     ecx, eax
.text:080482CF 01 D5             add      ebp, edx
.text:080482D1 B3 4F             mov      bl, 4Fh
.text:080482D3 0D B5 52 9A 61    or       eax, 619A52B5h
.text:080482D8 2D 80 C8 01 55    sub      eax, 5501C880h
; Trap to Debugger
.text:080482DD CC                int      3
.text:080482DE 33 F6             xor      esi, esi
.text:080482E0 42                inc      edx
.text:080482E1 F7 5C 99 E8       neg      dword ptr ↵
     [ecx+ebx*4-18h]
.text:080482E1 _start    endp
```

Кстати говоря, если установить точку останова на _start и дать отладчику немного поработать, мы попадем в самое начало расшифрованной программы, после чего ее будет можно анализировать в обычным режиме или снять дамп. Только это должна быть именно аппаратная (команда «hbreak _start» в gdb), а не программная («b _start») точка останова, иначе все рухнет (программная точка внедряет в расшифровываемую программу код CCh, который после расшифровки превращается совсем не в то, что было до нее).

Это очевидный просчет создателя шифратора. Вот если бы он перенаправил _start в какое-нибудь интересное место, вот тогда бы хакерам пришлось попыхтеть, а так… «защита» снимается в считанные секунды безо всякого труда, однако представляет интерес посмотреть, как выглядит код расшифровщика в IDA Pro, точка входа в который, как мы помним, равна 80495DCh:

Листинг 4. Дизассемблерный листинг расшифровщика, внедряемого ELFCrypt в файл (как он выглядит в IDA Pro)

```
extern:80495DC 7F 01 00 00    extrn puts@@GLIBC_2_0:near
extern:80495E0 FA 00 00 00    extrn ↵
      __libc_start_main@@GLIBC_2_0:near
; CODE XREF: .plt:_puts↑j
extern:80495E4 7F 01 00 00    extrn puts:near
; DATA XREF: .got:off_80495CC↑o
extern:80495E4
extern:80495E8 FA 00 00 00    extrn ↵
      __libc_start_main:near
; CODE XREF: ___libc_start_main↑j
extern:80495E8
; DATA XREF: .got:off_80495D0↑o
extern:80495E8
; weak
extern:80495EC 00             extrn _Jv_RegisterClasses
; weak
extern:80495F0 00             extrn __gmon_start__
; DATA XREF: .got:080495D4↑o
extern:80495F0
```

Что за чертовщина?! Каким образом расшифровщик может существовать в extern, когда здесь прямым текстом прописаны фактические адреса динамически загружаемых

функций! Но тот факт, что файл все-таки работает, убеждает нас, что да – может! Просто IDA Pro в попытке «эмуляции» загрузки elf-файла помещает в extern то, чего там на стадии загрузки файла еще нет.

Здесь мы подходим к одной из самых любопытных особенностей строения elf-файлов. В отличие от Windows, где заполнение extern происходит на стадии загрузки файла в память, в UNIX это делает стартовый код, причем делает он это очень хитрым способом. Ниже показан протокол трассировки программы под отладчиком с моими комментариями, отмеченными знаком «;» и содержимым дизассемблерного листинга IDA Pro, отмеченным знаком «#». Как говорится – сравните и почувствуйте разницу!

Для облегчения понимания возьмем незашифрованную программу, необработанную ELFCrypt:

```
Листинг 5. Протокол отладки, иллюстрирующий ход
динамической загрузки

; устанавливаем точку останова на начало стартового кода
(gdb) b _start

Breakpoint 1 at 0x80482c0: file ../sysdeps/i386/elf/start.S, line 47.

; запускаем программу на выполнение
(gdb) r

Breakpoint 1, _start () at ../sysdeps/i386/elf/start.S:47

; ок, мы в точке входа. Смотрим на extern
(gdb) x 0x80495DC

# extern:80495DC    7F 01 00 00          extern puts@@GLIBC_2_0:near
0x80495dc:          0x00000000

; IDA Pro нас уверяет, что extern содержит адрес 0000017Fh,
; но в действительности область extern на момент запуска
; файла девственно чиста и забита нулями

#.text:080482C0                          _start    proc near
#.text:080482C0 31 ED                    xor       ebp, ebp
1: x/i $pc  0x80482c0 <_start>:          xor       %ebp,%ebp

; незначащие машинные инструкции пропущены

#.text:080482D7 68 90 83 04 08           push      offset main
1: x/i $pc  0x80482d7 <_start+23>:       push      $0x8048390
#.text:080482DC E8 CF FF FF FF           call      __libc_start_main
1: x/i $pc  0x80482dc <_start+28>:       call      0x80482b0 <_init+56>

; но вот стартовый код вызывает библиотечную функцию
; __libc_start_main, поскольку компилятор еще не знает
; ее фактического адреса, он вставляет переходник к секции
; .plt, содержащей переходники к секции .got,
; заполняемой динамическим загрузчиком

#.plt:080482B0      __libc_start_main    proc near
#.plt:080482B0 FF 25 D0 95 04 08         jmp       ds:off_80495D0
1: x/i $pc  0x80482b0 <_init+56>:        jmp       *0x80495d0

; IDA Pro корректно отобразила plt-переходник, вызывающий
; функцию, указатель на которую расположен в двойном слове
; по адресу 80495D0h

#.got:080495D0 E8 95 04 08       off_80495D0 dd offset __libc_start_main
1: x/i $pc  0x80482b6 <_init+62>: push      $0x8
1: x/i $pc  0x80482bb <_init+67>: jmp       0x8048290 <_init+24>

; а вот тут уже начались расхождения...
; IDA Pro уверяет, что здесь расположено смещение функции
; __libc_start_main, в то время как отладчик показывает,
; что здесь находится специальный код push 08h/jmp 8048290h.
; Посмотрим, что покажет IDA Pro по адресу 8048290h

# .plt:08048290 ?? ?? ?? ?? ?? ??       dd 4 dup(?)
1: x/i $pc  0x8048290 <_init+24>:        pushl     0x80495c4
1: x/i $pc  0x8048296 <_init+30>:        jmp       *0x80495c8

; парад различий продолжается! IDA Pro вообще не показывает
; ничего! Отладчик же показывает код, засылающий в стек
; смещение первого (считая от нуля) элемента таблицы .got
; и передающего управление по адресу, записанному во втором
; элементе таблицы .got. Как следует из спецификации
; elf-формата, первые три элемента секции .got зарезервированы
; для служебных целей и вторая из них хранит адрес функции
; _dl_map_object_deps, которая, получив в качестве аргумента
; адрес начала .got, читает его содержимое
; (а содержатся там ссылки на библиотечные функции)
; и заполняет extern фактическими адресами

0x4000bbd0 in _dl_map_object_deps () from /lib/ld-linux.so.2
1: x/i $pc  0x4000bbd0 <_dl_map_object_deps+4384>:       push %eax

; ага! Вот эта функция, расположенная на моей машине
; по адресу 4000BBD0h, принадлежащему библиотеке libc.so.6
; (на других машинах этот адрес может быть иным)
; она-то и выполняет всю работу по инициализации extern,
; в котором находится наш расшифровщик, уже расшифровавший
; программу, а затем вызывает __libc_start_main,
; так что загрузка динамической библиотеки происходит
; совершенно прозрачно
```

Вот такая, оказывается, она IDA Pro! Чтобы скрыть код от глаз исследователя, достаточно разместить его в extern. Для вирусов, червей и прочего malware это очень даже актуально (особенно в свете того факта, что IDA Pro уже давно стала дизассемблером де-факто). На самом деле, IDA Pro (а точнее, elf-загрузчик) тут совсем не причем, «просто мы не умеем его готовить» . Чтобы все заработало правильно, необходимо при загрузке файла взвести флажок «Manual Load» и в появившемся диалоговом окне «Loading options» выбрать «Force using of PHT instead of SHT» (см. **рис. 2**).

Теперь и точка входа отображается нормально и файл можно расшифровать прямо встроенным в IDA Pro расшифровщиком (см. **рис. 3**), после чего продолжить дизассемблирование или снять готовый дамп.

Тот факт, что функция _dl_map_object_deps() вызывается из стартового кода, дает в наши руки универсальный способ распаковки elf-файлов, упакованных практически любым упаковщиком, за исключением тех случаев, когда файл слинкован со всеми используемыми библиотеками статическим образом (то есть действует по принципу «все свое всегда ношу с собой»), но такие файлы встречаются достаточно редко. Если только упаковщик не сопротивляется отладчику, достаточно всего лишь установить точку останова на _dl_map_object_deps() и... дождаться, когда она сработает. Тут же в стеке по адресу [ESP+08h] будет адрес возврата из CALL __libc_start_main, а по адресу

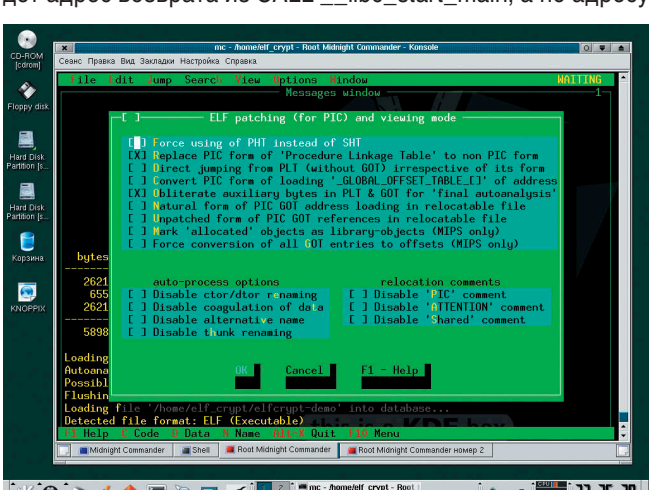

Рисунок 2. Выбор альтернативного метода загрузки elf-файлов в IDA Pro

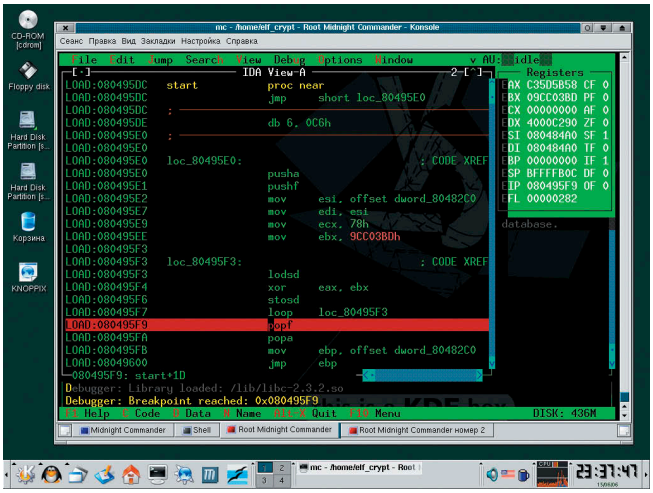

Рисунок 3. Расшифровка файла непосредственно в IDA Pro

[ESP+0Ch] указатель непосредственно на саму main. Если, конечно, нам повезет... Проблемы начинаются с того, что gdb с большой неохотой устанавливает точки останова на shared-функции и потому точка останова обязательно должна быть аппаратной, причем срабатывать она может несколько раз. Левые, срабатывая, распознаются легко. Если по [ESP+08h] и [ESP+0Ch] лежит совсем не то, что ожидалось (а это легко определить по диапазону адресов), пропускаем текущее срабатывание точки останова и продолжаем выполнение программы командой «c».

Примерный сеанс работы с отладчиком может выглядеть так:

```
Листинг 6. Распаковка программы путем установки точки
останова на _dl_map_object_deps

; загружаем программу в отладчик
root@5[elf_crypt]# gdb elfcrypt-demo
; ставим brkpnt на _dl_map_object_deps
(gdb) hbreak *0x4000BBD0

Hardware assisted breakpoint 1 at 0x4000bbd0

; запускаем программу
(gdb) r

Breakpoint 1, 0x4000bbd0 in _dl_map_object_deps () from /lib/ld-linux.so.2

; первое всплытие установленной точки останова
; сейчас будем проверять - «наше» ли оно или нет

; смотрим стек
(gdb) x $esp+8

0xbffffa6c:     0x40100498

; адрес указывает на libc.so.6
; это «левое» всплытие, идем дальше
(gdb) c

Continuing.

Breakpoint 1, 0x4000bbd0 in _dl_map_object_deps () from /lib/ld-linux.so.2

; второе всплытие установленной точки останова
; проверяем - «наше» ли оно или нет

; должен быть ret из call main
(gdb) x $esp+8

0xbffffafc:     0x080482e1

; судя по адресу, это возможно так и есть

; должен быть указатель на main
(gdb) x $esp+0xC

0xbffffb00:     0x08048390

; судя по адресу это так и есть

; проверяем наше предположение
(gdb) disassemble 0x80482e1

Dump of assembler code for function _start:
0x080482c0 <_start+0>:  xor    %ebp,%ebp
0x080482c2 <_start+2>:  pop    %esi
0x080482c3 <_start+3>:  mov    %esp,%ecx
0x080482c5 <_start+5>:  and    $0xfffffff0,%esp
0x080482c8 <_start+8>:  push   %eax
0x080482c9 <_start+9>:  push   %esp
0x080482ca <_start+10>: push   %edx
0x080482cb <_start+11>: push   $0x8048410
0x080482d0 <_start+16>: push   $0x80483b0
0x080482d5 <_start+21>: push   %ecx
0x080482d6 <_start+22>: push   %esi
0x080482d7 <_start+23>: push   $0x8048390
0x080482dc <_start+28>: call   0x80482b0 <_init+56>
0x080482e1 <_start+33>: hlt
End of assembler dump.

; дизассемблер показывает типичный стартовый код,
; значит, приложение уже распаковано!
```

Как вариант, изучив код распаковщика, можно написать скрипт для IDA Pro, выполняющий распаковку самостоятельно. Это хорошо работает с несложными расшифровщиками/распаковщиками, и в данном случае листинг укладывается всего в несколько строк:

```
Листинг 7. Скрипт для IDA Pro, расшифровывающий программу

auto a,x;
for(a=0x80482C0;a<0x8048338;)
{
      x=Dword(a);
      x = x ^ 0x9CC03BD;
      PatchDword(a,x);
      a = a + 4;
}
```

Естественно, чтобы написать скрипт, необходимо знать, откуда и докуда шифровать, а также ключ шифровки, для чего необходимо проанализировать алгоритм расшифровщика (см. **листинг 1**).

Кстати говоря, IDA Pro не обновляет модифицируемый код в окне дизассемблера (точнее обновляет, но делает это как-то странно), поэтому нам необходимо нажать <U>, разрушая ранее дизассемблированные инструкции в поток байт, а затем <C> для превращения их в дизассемблерный код.

Другой способ противодействия упаковщикам заключается в подключении (attach) к уже запущенному процессу (задолго после того, как упаковщик все уже распаковал). В gdb за это делается так: «gdb --pid=<PID>», где PID – идентификатор исследуемого процесса, который можно узнать с помощью команды «ps -a». Однако это не самый лучший путь, поскольку мы вторгаемся в программу уже после инициализации кучи структур данных и снятый дамп может оказаться неработоспособным. К тому же из-за игр с extern и несоответствия _start реальной точке входа, существующие UNIX-дамперы не могут реконструировать elf-файл, получая Segmentation fault. Правда, можно воспользоваться утилитой PD (более подробно она рассматривается в разделе, посвященном упаковщику UPX), указав «волшебный» ключик -l, предписывающий не трогать секцию .got, тогда Segmentation fault станет вызывать сдампленный файл, но зато он будет полностью расшиф-

рован, что (теоретически) должно существенно упростить дизассемблирование, но практически из-за отсутствия символьных имен библиотечных функций анализ рискует превратиться в пытку.

Если же снимать дамп необязательно и достаточно просто «посмотреть», что делает упакованная программа, можно использовать утилиту ltrace, сеанс работы с которой показан ниже. Как видно, ELFCrypt совсем не пытается ей противостоять.

Листинг 8. Результат работы ltrace

```
__libc_start_main(0x80483c4, 1, 0xbffffb34, 0x8048410, 0x8048470 <unfinished ...>
printf(0xbfffffac0, 0x40017a50, 0xbfffffad8, 0x804842b, 0x6c6c6568) = 13
hello, world!
gets(0xbfffffac0, 0x40017a50, 0xbfffffad8, 0x804842b, 0x6c6c6568) = 0xbfffffac0
+++ exited (status 192) +++
```

Помимо самых вызываемых функций, ltrace также отражает и адреса возврата (в листинге они выделены полужирным), что позволяет нам, устанавливая на них аппаратные точки останова, врываться в любое место уже распакованной программы! В общем, не жизнь, а красота! Однако не будем забывать, что ELFCrypt это даже не упаковщик, а экспериментальная студенческая поделка. Но справившись с ним, мы сможем справиться и с гораздо более сложными программами…

UPX

Это один из наиболее древних упаковщиков, созданный тройкой магов Markus F.X.J. Oberhumer, Laszlo Molnar и John F. Reiser, поддерживающий рекордное количество форматов файлов (от Amiga до UNIX) и расшифровывающий свою аббревиатуру как the «Ultimate Packer for eXecutables». Свежую версию вместе с исходными текстами можно бесплатно скачать с «родного» сайта проекта: http://www.upx.org или с «кузни»: http://upx.sourceforge.net.

UPX не имеет никакого защитного кода, никак не противодействуя ни отладке, ни дизассемблированию, более того, он даже содержит встроенный распаковщик, за который «отвечает» ключ командной строки -d.

С коммерческой точки зрения UPX выгоден тем, что упакованные им файлы работают практически на всем спектре UNIX-подобных систем, однако наличие встроенного упаковщика делает его совершенно бесполезным для защиты программ. Но это еще как посмотреть! Доступность исходных текстов позволяет слегка модифицировать структуру упаковываемого файла так, что родной распаковщик уже не сможет с ней работать.

Самое простое, что можно сделать – это затереть сигнатуру «UPX!», расположенную в конце файла, тогда UPX не сможет распознать упакованный файл, и встроенный распаковщик откажется с ним работать:

Листинг 9. Сигнатура «UPX!», расположенная в конце упакованных файлов

```
000013B390:  92 24 FF 00 55 50 58 21 | 0D 0C 08 07 8F F1 E8 8C  T$  UPX!♪♀◘•ПёшМ
000013B3A0:  05 97 B4 63 8C 6F 43 00 | 19 EC 0D 00 00 41 52 00  ♣Ч┤cМoC ↓♪  AR
000013B3B0:  49 14 00 37 80 00 00 00 |                         I 7A
```

Проведем небольшой эксперимент. Откроем упакованный файл в любом hex-редакторе и запишем поверх «UPX!» что-то свое, например: «6669».

Листинг 10. Затертая сигнатура

```
000013B390:  92 24 FF 00 55 50 58 21 | 0D 0C 08 07 8F F1 E8 8C  T$  6669♪♀◘•ПёшМ
000013B3A0:  05 97 B4 63 8C 6F 43 00 | 19 EC 0D 00 00 41 52 00  ♣Ч┤cМoC ↓♪  AR
000013B3B0:  49 14 00 37 80 00 00 00 |                         I 7A
```

Файл запускается так же, как и раньше, но теперь UPX наотрез отказывается его распаковывать:

Листинг 11. Встроенный распаковщик UPX не смог распаковать файл с затертой сигнатурой

```
root@5[upx-2.01-i386_linux]# ./upx -d elinks

                       Ultimate Packer for eXecutables
                          Copyright (C) 1996,1997,1998,1999,2000,2001,2002,2003,2004,2005,2006
                          UPX 2.01        Markus Oberhumer, Laszlo Molnar & John Reiser   Jun 06th 2006

        File size         Ratio      Format      Name
   --------------------   ------   -----------   -----------

upx: elinks_2: NotPackedException: not packed by UPX
Unpacked 0 files.
```

Поскольку UPX не использует libc и работает через интерфейс системных вызовов, то динамические библиотеки подключаются только после того, как распаковка будет завершена. А это значит, что, установив точку останова на функцию _dl_map_object_deps(), мы сорвем банк, ворвавшись в уже распакованную программу.

Так же сработает подключение отладчика к активному процессу. При желании можно получить не только сырой дамп, но и готовый к работе elf-файл. К сожалению, прямых аналогов знаменитого proc-dump под UNIX нет (команда «generate-core-file» отладчика gdb создает файл, пригодный для дизассемблирования, но, увы, не запуска), но некоторые действия в этом направлении уже наблюдаются. Утилита PD, исходный код которой (с объяснением принципов его работы) опубликован в 63 номере журнала «PHRAK» (www.phrack.org/phrack/63/p63-0x0c_Process_Dump_and_Binary_Reconstruction.txt), легко дампит большинство простых файлов, но со сложным пока еще не справляется (однако оставляет шанс доработать их руками). Самое печальное, что PD не может снимать дампы программ, исполняющихся под отладчиком (а ведь в мире Windows хакеры поступают именно так – находят оригинальную точку входа отладчиком, после чего зовут proc-dump или его более современный аналог PE-TOOLS). Однако существует возможность отсоединиться от процесса коман-

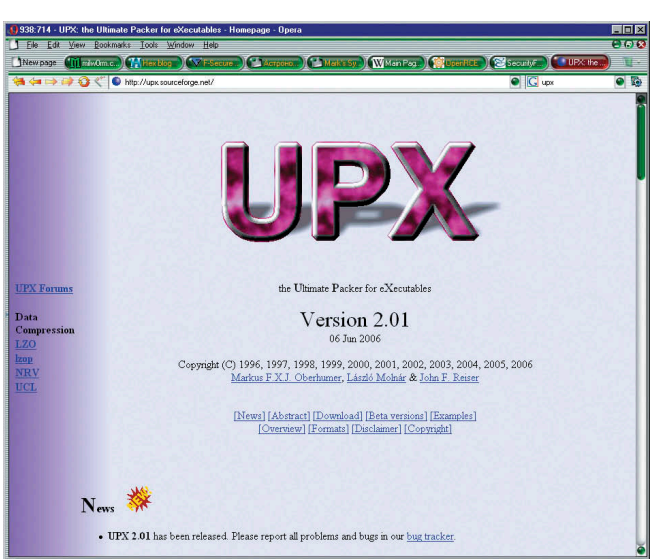

Рисунок 4. Авторская страница упаковщика UPX

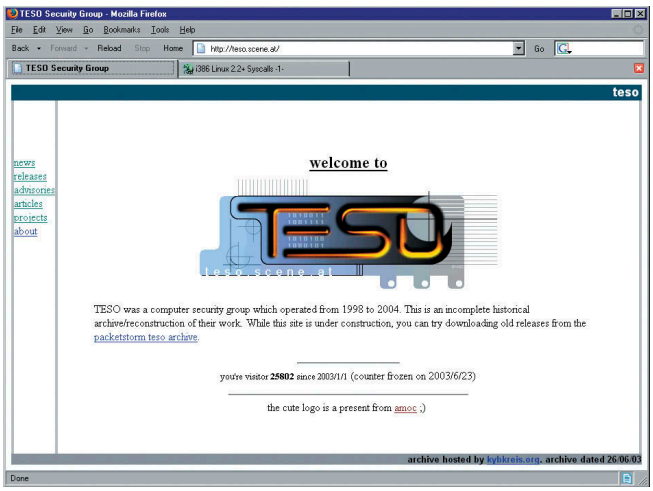

Рисунок 5. Сайт группы TESO

дой «detach», и до выхода из отладчика он будет находиться в «замороженном» состоянии, что позволяет беспрепятственно снять с него дамп.

Там же в статье присутствует ссылка на базовый сайт проекта (http://www.reversing.org), но никаких новых версий там нет, так что будем пользоваться тем, что дают. Ниже показан сеанс работы с утилитой PD:

Листинг 12. Снятие дампа с последующей реконструкцией elf-файла

```
; запускаем упакованный процесс на выполнение
root@5[src]# ./demo
; определяем его pid
root@5[src]# ps -a
  PID TTY          TIME CMD
 9771 pts/7    00:00:00 demo
 9779 pts/5    00:00:00 ps
; дампим процесс в файл
root@5[src]# ./pd -o dumped 9771
pd V1.0 POF <ilo@reversing.org>
download last version from: http://www.reversing.org
source distribution for testing purposes..

performing search..
only PAGESZ method implemented in this version
AT_PAGESZ located at: 0xbffffbc8

gather process information and rebuild:
-loadable program segments, elf header and minimal size..

analyzing dynamic segment..

Agressive fixing Global Object Table..
 vaddr: 0x8049620 daddr: 0x8049000 foffset: 0x620
* plt unresolved!!!

section headers rebuild
this distribution does not rebuild section headers

saving file: dumped

Finished.
; утилита PD окончила процесс дампинга
; запускаем полученный процесс на выполнение
root@5[src]# ./dumped
```

А вот ltrace работать не будет, поскольку она нуждается в секциях .dynsym или .dynstr, которых в файлах, упакованных UPX, нет! Тем не менее, как было показано выше, это все равно не спасает его от взлома.

Burneye

Первый UNIX-упаковщик с претензией на протектор, созданный молодым 25-летним хакером по кличке Scut (scut@segfault.net), он же «The Tower», живущим в Западной Германии и входящим в группу TESO, в настоящее время работающей над linice – аналогом soft-ice под UNIX.

Burneye – это экспериментальный протектор, распространяющийся на бесплатной основе. Сначала его исходные тексты были недоступны, но затем (под напором общественности) выложено ~30% от общего объема кода, а затем и весь проект целиком. Все это добро можно скачать с packetstorm.

Архив packetstorm.linuxsecurity.com/groups/teso/burneye-1.0-linux-static.tar.gz содержит откомпилированную версию, работающую под LINUX и частично под BSD («частично» потому, что иногда падает), в packetstorm.linuxsecurity.com/groups/teso/burneye-stripped.tar.gz лежит 30% исходных текстов и несколько статей с новыми, но так и не реализованными идеями по усилению защиты, а packetstorm.linuxsecurity.com/groups/teso/burneye-1.0.1-src.tar.bz2 включает в себя все исходные тексты.

Протектор умеет шифровать файлы по алгоритмам SHA1 и RC4, требуя от пользователя пароль при запуске. Теоретически взломать программу можно и без знания пароля (криптография не стоит на месте, и подходящий переборщик можно найти на http://byterage.hackaholic.org/source/UNFburninhell1.0c.tar.gz), но в практическом плане гораздо проще купить одну-единственную лицензионную копию, а потом выложить ключ на всеобщее обозрение. Чтобы этого не произошло, в протектор заложена возможность «привязки» к оборудованию пользователя (так называемый fingerprint). Это довольно интересная тема, но лучше оставим ее на потом, сосредоточившись исключительно на распаковке.

Burneye состоит из множества вложенных друг в друга расшифровщиков, генерируемых произвольным образом, что, впрочем, не сильно препятствует его трассировке, поскольку расшифровщики реализованы как процедуры. (Имеющиеся антидизассемблерные приемы сводятся к прыжку в середину команды и легко обходятся как в IDA Pro, так и в hiew).

Во всем протекторе содержится всего один антиотладочный прием, препятствующий трассировке под gdb и отладчиком, интегрированным в IDA Pro.

Листинг 13. Дизассемблерный листинг единственного антиотладочного приема в Burneye

```
LOAD:053714A7           mov ebx, 5          ; SIGTRAP
; обработчик
LOAD:053714AC           mov ecx, offset anti_handler
LOAD:053714B1           mov edx, 30h        ; signal
LOAD:053714B6           mov eax, edx
; signal(SIGTRAP, anti_handler);
LOAD:053714B8           int 80h
LOAD:053714BA           add esi, offset word_5375A00
LOAD:053714C0           mov [ebp-2DCh], esi
LOAD:053714C6           int 3     ; Trap to Debugger
; если ноль, мы под отладчиком
LOAD:053714C7           cmp anti_debug, 0
LOAD:053714CE           jnz short debugger_not_present
...
; обработчик сигнала SIGTRAP (получает управление только
; при запуске без отладчика)
LOAD:05371A0C  anti_handler:
LOAD:05371A0C           push      ebp
```

```
LOAD:05371A0D           mov ebp, esp
; увеличиваем секретную переменную
LOAD:05371A0F           inc anti_debug
LOAD:05371A15           leave

LOAD:05371A16           retn      ; выходим из обработчика
```

Программа устанавливает свой собственный обработчик (в приведенном выше листинге он обозначен как anti_handler), ожидающий прихода сигналов типа SIGTRAP, а затем вызывает прерывание INT 03h. При нормальном развитии событий управление получает anti_handler, увеличивающий значение переменной anti_debug, которая тут же проверяется со своим первоначальным значением. При работе под отладчиком сигнал SIGTRAP «поглощается» отладчиком, и anti_handler управления не получает.

Основная проблема протектора в том, что переменная anti_debug проверяется в одном-единственном месте, один-единственный раз! Чтобы обойти защиту, можно либо записать в переменную anti_debug любое значение, отличное от нуля, либо трассировать программу вплоть до достижения INT 03h, после чего «вручную» изменить значение регистра $pc на anti_debug и спокойно продолжить отладку. Так же можно заменить инструкцию «cmp anti_debug, 0» на «cmp anti_debug, 1» (но это только в том случае, если в программе нет проверки целостности собственного кода). Короче, вариантов много, но все они требуют участия человека, что напрягает.

Когда борьба с Byrneye всех хакеров окончательно достала, некто по имени ByteRage (byterage@yahoo.com) написал автоматический распаковщик – burneye unwrapper, по обыкновению бесплатно распространяемый в исходных текстах. Впрочем, называть «исходными текстами» крошечную C-программу, представляющую собой загружаемый модуль ядра, можно только с большой натяжкой.

Качаем http://byterage.hackaholic.org/source/burndump.c, компилируем своим любимым компилятором «gcc c burndump.c» (на некоторых системах необходимо явно указать включаемые файлы «gcc c I/usr/src/linux/include burndump.c») и загружаем внутрь ядра «insmod burndump» (естественно, для этого необходимо иметь права root). Теперь burndump будет сидеть резидентно в памяти и перехватывать системный вызов brk(), который нужен упаковщику для расширения сегментов elf-файла в памяти. К моменту вызова этой функции файл уже распакован – остается только снять с него дамп и записать на диск. Чтобы не писать все подряд, необходимо как-то отождествить упаковщик. В burndump за это отвечает следующая малопонятная конструкция:

Листинг 14. Фрагмент burndump, отождествляющий упаковщик по «сигнатуре»

```
codeptr = current->mm->start_code + 1;
/* caller == burneye ??? */
if ((codeptr >> 16) == 0x0537)
      printk("<1> 7350 signature 0x0537 found!\n");
```

Но все сразу же становится ясным, если взглянуть на файл, обработанный протектором Burnyey: как видно, протектор располагает себя по довольно нехарактерным адресам, и дампер просто сравнивает 16 старших байт адреса, вызывающего функцию brk().

Листинг 15. Фрагмент файла, упакованного протектором Burneye

```
.05371035: FF3508103705  push     d,[05371008]
.0537103B: 9C            pushfd
.0537103C: 60            pushad
.0537103D: 8B0D00103705  mov      ecx,[05371000]
.05371043: E93A000000    jmp      .005371082  ---↓ (1)
```

После запуска упакованного файла на диске автоматически образуется распакованный ./burnout, который уже не привязывается к машине и который можно свободно отлаживать или дизассемблировать в свое удовольствие.

Выгрузка резидентного модуля из памяти осуществляется командой «rmmod burndump», но не спешите с ним расставаться! Слегка доработав исходный текст, мы сможем распаковывать и другие протекторы (когда они появятся), а не только один лишь Burneye. Дампер уровня ядра – это вещь! Это настоящее оружие, с которым очень трудно справиться на прикладном уровне! (Впрочем, Burneye с легкостью снимается и PD). Короче, победу над Burneye можно считать полной и окончательной.

Shiva

Весьма амбициозный протектор, созданный двумя гуру Neel Mehta и Shaun Clowes (Email: shiva@secureреality.com.au) и представленный ими на конференции Black Hat, проходившей в Азии в 2003 году.

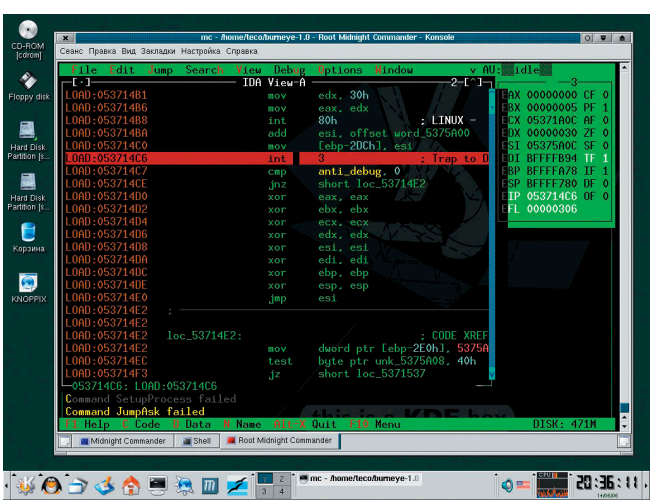

Рисунок 6. Обход антиотладочного приема в отладчике, интегрированном в IDA Pro

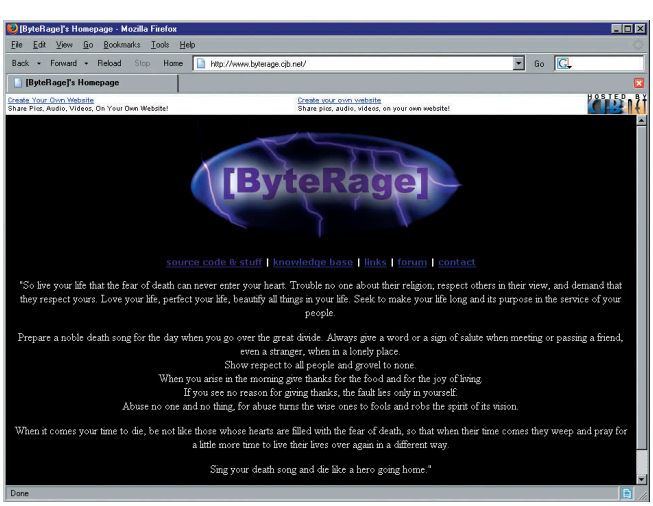

Рисунок 7. Сайт создателя утилиты ByteRage

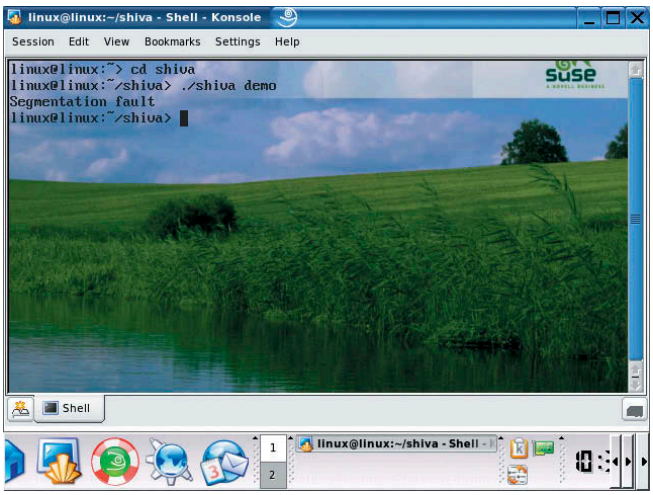

Рисунок 8. Неизменный Segmentation fault при попытке запустить протектор shiva, возникающий на всех доступных мне ядрах/машинах

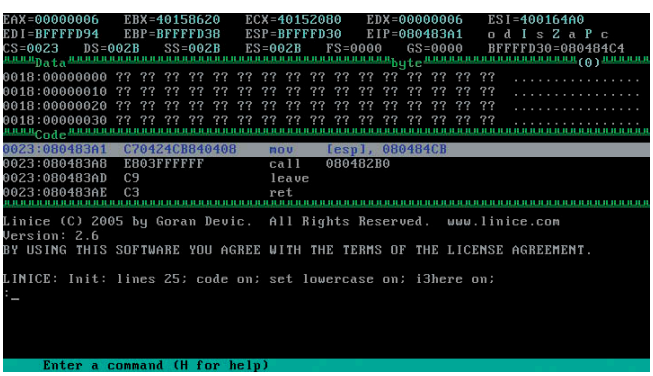

Рисунок 9. Внешний вид ядерного отладчика linice, своеобразного аналога soft-ice для Linux

Исходные тексты не разглашаются (как будто там есть что скрывать!), а сам бинарник можно скачать как с сайта разработчиков www.securereality.com.au/archives/shiva-0.95.tar.gz, так и с сервера Black Hat: blackhat.com/presentations/bh-usa-03/bh-us-03-mehta/bh-us-03-shiva-0.96.tar, причем версия с Black Hat посвежее будет, что наводит на определенные размышления. Там же, на Black Hat, можно найти тексты мультимедийной презентации от обоих разработчиков и в pdf. Первый: www.blackhat.com/presentations/bh-usa-03/bh-us-03-mehta/bh-us-03-mehta.pdf и второй: http://www.blackhat.com/presentations/bh-asia-03/bh-asia-03-halvar.pdf.

Разработчики реализовали мощную антиотладку, многоуровневую динамическую шифровку, эмуляцию некоторых процессорных инструкций… в общем получился почти что Armadillo, только под Linux. Но если Armadillo хоть как-то работает, то Shiva на всех доступных мне системах выпадет в Segmentation fault. Конкретно тестировались: KNOPPIX с ядрами 2.6.7/4.2.7 и SUSE с ядром 2.6.8, пускаемых как под VMWare, так и на «живой» машине с процессором AMD Athlon-1700.

Поэтому вся информация, приведенная ниже, получена исключительно путем дизассемблирования и отладки протектора.

Начнем с отладки, так как под Linux это самый больной вопрос. Операционная система предоставляет механизм ptrace, которым пользуется gdb и подавляющее большинство остальных отладчиков (отладчик, интегрированный в IDA Pro, ALD – Assembly Language Debugger и т. д.). Собака зарыта в том, что механизм ptrace нерентабелен, то есть программу, уже находящую под отладкой, отлаживать нельзя! Shiva воспользовался этим фактом, породив дочерний процесс, отлаживающий сам себя (ну все, как у Armadillo!), чем надежно защитился как от трассировки, так и от вызова PTRACE_ATTACH, поскольку он тоже работает через ptrace!

Найти же удобоваримый отладчик, работающий в обход ptrace, оказалось, на удивление, сложной задачей (тем более что в дополнение к этому Shiva распознает TRAP-флаг, анализируя бит трассировки в регистре EFLAGS процессора и выполняет контроль таймингов). Поиск обнаруживает только кладбища заброшенных проектов.

Заброшенный, заново воскрешенный и снова заброшенный ядерный отладчик The-DUDE (http://the-dude.sourceforge.net), другой ядерный отладчик – privateICE (http://pice.sourceforge.net) – поддерживает лишь фиксированный набор ядер, среди которых нет ни одного «моего». Из всех отладчиков удалось запустить лишь linice (http://www.linice.com), да и то в VGA-режиме.

Морской волк Chris Eagle (cseagle@nps.navy.mil) пошел другим путем и на той же самой конференции продемонстрировал автоматический депротектор. Вместо поиска отладчиков, работающих в обход ptrace, он разработал отладчик-эмулятор x86-процессора, выполненный в виде подключаемого модуля для IDA Pro и бесплатно распространяемый в исходных текстах: http://sourceforge.net/projects/ida-x86emu, однако имейте в виду, что для его компиляции требуется IDA SDK, который есть не у всех.

Текст мультимедийной презентации с описанием методики взлома лежит на Black Hat: http://www.blackhat.com/presentations/bh-federal-03/bh-federal-03-eagle/bh-fed-03-eagle.pdf, а набор утилит для взлома (включающий в себя автоматический распаковщик и несколько полезных скриптов для IDA Pro, упрощающих расшифровку), находится в соседнем файле: www.blackhat.com/presentations/bh-federal-03/bh-federal-03-eagle/bh-federal-03-eagle.zip.

Теперь что касается расшифровки. Чтобы противостоять дампу даже на уровне ядра, Shiva использует динамическую расшифровку по требованию (on demand). Неиспользуемые в данный момент страницы заполняются байтами CCh, представляющими собой инструкцию INT 03h, передающую управление материнскому процессу-отладчику при попытке их выполнения, что сигнализирует о необходимости их расшифровки, которая осуществляется «подкачкой» недостающих байтов из «резервного» хранилища (Armadillo, помнится, менял атрибуты доступа страниц). Разумеется, этот трюк работает только с кодом, но не с данными, и их приходится расшифровывать статическим расшифровщиком.

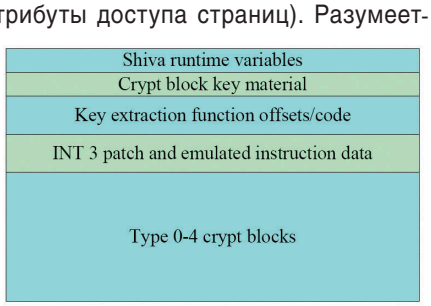

Рисунок 10. Структура файла, зашифрованного Shiva

В дополнение к этому Shiva заменяет в расшифро-

ванных блоках инструкции PUSH, JMP и CALL на INT 03h и эмулирует их выполнение. Все очень просто. Shiva держит в памяти специальную таблицу с адресами замещенных инструкций, и если TRAP по выполнению INT 03 приходит по одному из этих адресов – задействуется механизм эмуляции. В практическом плане это означает, что даже расшифровав все зашифрованные блоки, мы все равно не сможем избавиться от Siva RTL (среды исполнения) и будем вынуждены «тащить» протектор за собой, если, конечно, не декодируем эту таблицу адресов и не восстановим «украденные» команды.

Для противодействия дизассемблеру Shiva генерирует большое количество полиморфного кода и постоянно совершает прыжки в середину инструкций, что ужасно напрягает.

Короче говоря, Shiva – это кривая калька с Armadillo и к тому же неработающая. В то время как под Windows протектор Armadillo уже давно не является чудом инженерной мысли, ситуация в мире UNIX напоминает СССР в эпоху «персональных компьютеров коллективного использования».

Заключение

Через несколько лет, когда рынок закрытого программного обеспечения под UNIX достигнет критической точки, упаковщики исполняемых файлов, возможно, начнут играть существенную роль, но пока же они годятся разве что для забавы и… подготовки к схватке с по-настоящему серьезным противником.

Основные характеристики наиболее популярных UNIX-упаковщиков (неблагоприятные для хакеров свойства затемнены)

Характеристика	ELF-Crypt	UPX	Byrneye	Shiva
anti-debug	нет	нет	да	да
anti-dissembler	есть	нет	да	да
anti-ltrace	нет	да	да	да
allow to attach	да	да	да	нет
anti «procdump»	да	нет	нет	да
интерфейс	libc	syscall	syscall	syscall
содержит распаковщик	нет	да	нет	нет
взломан	да	да	да	да

Хакерские утилиты под UNIX уже пишутся, и к тому моменту, когда защитные механизмы выйдут на арену, разработчики с удивлением обнаружат, что ситуация уже совсем не та, что пару лет назад, и теперь им противостоят не пионеры, ковыряющие внутренности UNIX в свободное время от основных дел, а хорошо подготовленные специалисты, которых никаким протектором не напугаешь.