перехват библиотечных функций в linux и bsd

крис касперски аргентинский болотный бобер nezumi el raton ака нутряк ибн мыщъх, no e-mail

как узнать какие функции вызывает подопытная программа? в windows существует целый арсенал шпионских средств, но linux-хакерам весь инструментарий приходится разрабатывать самостоятельно. сейчас мыщъх покажет как осуществляется перехват и подмена (!) системных и библиотечных функций в linux и *BSD

введение

При всей непохожести windows на linux между ними можно выделить общие черты. Обе системы образуют "слоеный пирог" из библиотек различных уровней иерархий. Ядерные функции windows NT сосредоточены в файле ntoskrnl.exe, доступ к которым осуществляется через прерывание INT 2Eh (NT 3.5x, NT4.x, W2K) или через INT 2Eh/sysenter (XP, Longhorn). В linux для той же цели используется прерывание IINT 80h (x86 BSD использует гибридный механизм, одновременно поддерживая как INT 80h, так и call far 0007h:00000000h).

Ядро реализует базовые функции ввода/вывода, распределения памяти, создания/завершения процессов и т. д., причем если NT предоставляет низкоуровневые полуфабрикаты, над которыми еще предстоит поработать, ядерные функции linux'a (они же "системные вызовы" или по-английски sys-calls) вполне юзабельны. Тем не менее прямые обращения к ядру с прикладного уровня встречаются редко. Вместо этого приложения предпочитают использовать системно-независимую библиотеку libc.so.x – отдаленный аналог KERNEL32.DLL из windows. Эта библиотека загружается в физическую память всего один раз, а затем проецируется на адресной пространство всех используемых ее процессов ("so" расшифровывается как "shared object [file]", а x – номер версии, например, "libc.so.6").

Помимо libc, существуют и другие библиотеки, например, libncurses.so.x, отвечающая за управление курсором и отрисовку псевдографики в текстовом режиме ("аналог" USER32.DLL). Библиотеки могут подключаться как на стадии загрузки elf-файла через таблицу символов (аналог таблицы импорта), так и динамически по ходу выполнения программы посредством вызова функций dlopen/dlsym (аналог LoadLibrary/GetProcAddress соответственно). Наконец, всякая программа содержит большое количество непубличных не экспортируемых функций, которые так же требуется перехватывать.

В штатный комплект поставки (out form box) некоторых UNIX'ов входят две "хакерские" утилиты truss и krace (в LINUX – strace), следящие за системными вызовами и ведущие подробный log (см. листинг 1). К сожалению, эти утилиты есть не на всех системах, и даже там где они есть, для решения поставленной задачи они непригодны: многие библиотечные функции обрабатываются локально и до ядра просто не доходят. К тому же strace работает через механизм ptrace, а он нерентабелен (т. е. не позволяет трассировать уже трассируемые программы) и с которым успешно сражаются многие защитные механизмы, черви и упаковщики. Наконец, в некоторых случаях приходится не только шпионить за вызовами, но и "подминать" чужие функции, заменяя их на свои.

Как это сделать?

```
mmap(0x0,4096,0x3,0x1002,-1,0x0)                          = 671657984 (0x2808b000)
break(0x809b000)                                          = 0 (0x0)
break(0x809c000)                                          = 0 (0x0)
break(0x809d000)                                          = 0 (0x0)
break(0x809e000)                                          = 0 (0x0)
stat(".",0xbfbff514)                                      = 0 (0x0)
open(".",0,00)                                            = 3 (0x3)
fchdir(0x3)                                               = 0 (0x0)
open(".",0,00)                                            = 4 (0x4)
stat(".",0xbfbff4d4)                                      = 0 (0x0)
open(".",4,00)                                            = 5 (0x5)
fstat(5,0xbfbff4d4)                                       = 0 (0x0)
fcntl(0x5,0x2,0x1)                                        = 0 (0x0)
__sysctl(0xbfbff38c,0x2,0x8096ab0,0xbfbff388,0x0,0x0) = 0 (0x0)
fstatfs(0x5,0xbfbff3d4)                                   = 0 (0x0)
break(0x809f000)                                          = 0 (0x0)
getdirentries(0x5,0x809e000,0x1000,0x809a0b4)             = 512 (0x200)
getdirentries(0x5,0x809e000,0x1000,0x809a0b4)             = 0 (0x0)
```

```
lseek(5,0x0,0)                                  = 0 (0x0)
close(5)                                        = 0 (0x0)
fchdir(0x4)                                     = 0 (0x0)
close(4)                                        = 0 (0x0)
fstat(1,0xbfbff104)                             = 0 (0x0)
break(0x80a3000)                                = 0 (0x0)
write(1,0x809f000,158)                          = 158 (0x9e)
exit(0x0)                                       process exit, rval = 0
```

Листинг 1 образец отчета truss

Рисунок 1 программа "hello,world", под "микроскопом" truss'a

обзор возможных методов

Условимся рассматривать универсальные методики перехвата, не требующие модификации ни подопытного файла, ни ядра и работающие под любой UNIX-подобной системой (возможно, с небольшими переделками).

Начнем с классики, то есть издалека. Один из самых популярных методов перехвата, активно используемый под windows, и называемый "**методом модификации [таблицы] импорта**" выглядит так:

- создаем отладочный процесс вызовом fork()/exec()/ptrace(PTRACE_TRACEME [в BSD — PT_TRACE_ME, в дальнейшем BSD-объявления будет приводится через слеш]) или подключаемся к уже запущенному процессу через ptrace(PTRACE_ATTACH/PT_ATTACH, pid, 0, 0);
- через функцию ptrace(PTRACE_PEEKTEXT/PT_READ_I, pid, addr, 0) читаем глобальную таблицу смещений (global offset table или got) — аналог таблицы импорта;
- посредством функции ptrace(PTRACE_POKETEXT/PT_WRITE_I, pid, addr, data) модифицируем указатели на нужные нам функции, заменяя их на offset thunk, где thunk — наш обработчик, внедренный в адресное пространство процесса тем или иным путем (например, при помощи той же PTRACE_POKETEXT/PT_WRITE_I);
    - контроль за динамически загружаемыми библиотеками осуществляется путем перехвата функций dlopen/dlsym, экспортируемыми libdl.so.x, но фактически реализованных в libc.so.x (там они называются _dl_open/_dl_sym соответственно);
    - непубличные функции самой программы и статических библиотек перехватываются путем внедрения команды jump thunk в их начало (естественно, оригинальное содержимое нужно где-то предварительно сохранить) с поиском по сигнатурам или с привязкой к фиксированным адресам;
- отсоединяемся от процесса через ptrace (PTRACE_DETACH/PT_DEATACH, pid, 0, 0), позволяя ему продолжить нормальное выполнение, но теперь уже с исправленной" глобальной таблицей смещений, вызывающей функции через наш хакерский thunk, который может протоколировать вызовы, мухлевать с аргументами или даже передавать управление подложным функциям;

Метод "модификации импорта" легко реализуется, надежен, но… не свободен от недостатков. В windows функции ReadProcessMemory/WriteProcessMemory не требуют от процесса, чтобы он находится под отладкой и подопытному приложению очень трудно им противостоять. Их linux-аналоги являются частью библиотеки ptrace, обломать которую очень легко (см. мою статью "методология защиты в мире UNIX", опубликованную в Хакере с год назад). К тому же подопытный процесс может вырваться из лап шпиона. Для этого ему достаточно породить дочерний процесс или сделать себе exec(), чтобы перезапуститься. В этом случае системный загрузчик перечитает исходный образ elf-файла с диска и все изменения в got'e будут потеряны. Чтобы этого не произошло, наш шпион должен следить за всеми потенциально опасными функциями, пускай и ценой усложнения реализации. И последнее (но самое главное) ограничение, — шпионаж носит сугубо локальный характер и может контролировать только дочерние процессы или процессы, явно переданные ему на "съедение".

А вот другой популярный прием, называемый методом "**подмены библиотеки**", так же позаимствованный из мира windows:

- создаем "обертку" (wrapper) вокруг интересующей нас библиотеки, экспортирующей те же самые функции, что и она;
- оригинальную библиотеку переименовываем или ложим на в другое место;
- функции-обертки определяют идентификатор вызывающего их процесса и если это действительно "их" процесс, совершают заранее запланированные действия (пишут вызов в log, подменяют аргументы или код возврата и т. д). Как определить id процесса? Это легко — ведь функции-обертки вызываются из контекста используемого их процесса и решение задачи сводится к выяснению идентификатора текущего процесса, возвращаемого функцией getpid;
- функция-обертка передает управление оригинальной функции основной библиотеки или своей собственной подложной функции;

За внешней простотой реализации такого подхода кроется целый ворох проблем. Создать обертки вокруг всех "системных" библиотек вручную практически нереально и эту работу необходимо автоматизировать, написав несложный парсер so-файлов и кодогенератор. Не обязательно генерировать готовый elf-файл — проще создать Си-программу и откомпилировать ее gcc.

"Глобальность" перехвата воздействует на все процессы и кривая "обертка" рушит ось так, что только дампы летят. Давайте не будем трогать системные библиотеки, а вместо этого изменим переменную LD_LIBRARY_PATH в окружении "подопытного" процесса. Она специально предусмотрена на тот случай, если отдельно взятому приложению требуется предоставить свой набор библиотек (статический и динамический загрузчики сначала ищут библиотеку в путях, указанных LD_LIBRARY_PATH, и только если там ее не оказывается переходят к файлу /etc/ld.so.conf, а затем к путям — /lib и /usr/lib), но если "подопытный" процесс попытается загрузить библиотеку по абсолютному пути, он сможет вырваться из-под нашего контроля!

Перспективнее всего осуществлять перехват путем **прямой модификации памяти** подопытного без обращения к ptrace. Как это можно сделать? Первым в голову приходит файл /proc/<pid>/mem, однако, в большинстве систем он недоступен даже root'у и приходится спускаться на уровень ядра, что сильно напрягает. Хакерские источники упоминают о двух других интересных файлах: **/dev/mem** (образ физической памяти компьютера до трансляции виртуальных адресов) и **/dev/kmem** (образ виртуальной памяти ядра). Файл /dev/kmem обычно с прикладного уровня недоступен и никаких библиотек прикладного уровня здесь нет, поэтому нам он совершенно неинтересен, а вот /dev/mem мы рассмотрим поподробнее.

/dev/mem

Чтение документации ("man mem") показывает, что файл /dev/mem имеется практически на всех UNIX-подобных системах, а если его вдруг нет, он может быть создан в любой момент следующими командами:

```
mknod -m 660 /dev/mem c 1 1
chown root:kmem /dev/mem
```

Листинг 2 создание файла-устройства /dev/mem, предоставляющего доступ к физической памяти компьютера с прикладного уровня

Здесь: 660 – права доступа, /dev/mem – имя файла (может быть любым, например /home/kpnc/nezumi), "c" – тип устройства (символьное устройство), "1 1" – устройство (физическая память). Файл /dev/mem (или как вы его назовете) свободно доступен с прикладного уровня, но только для root, что есть саксь.

Структура файла предельно проста — линейные смещения соответствуют физическим адресам. Допустим, нам известно, что по адресу FFFFh:FFF0h во всех BIOS'ах хранится команда перехода на загрузочный код, а за ним (как правило) лежит дата создания прошивки. Переводим "сладкую парочку" сегмент:смещение в линейный вид (linear == seg*10h + offset), получаем FFFF0h. Это и будет смещение в файле.

Рисунок 2 чтение содержимого BIOS'а через файл /dev/mem

Основной камень преткновения в том, что оперативная память компьютера используется UNIX'ом как кэш и потому один и те же физические страницы в различное время

могут соответствовать различным виртуальным адресам. Но это не проблема. Можно найти каталог страниц и выполнить трансляцию вручную. Указатель на текущий каталог храниться в регистре CR3, попытка доступа к которому с прикладного уровня возбуждает исключение, но поскольку каталог имеет довольно характерную структуру (описанную в документации на процессор), его легко найти простым сканированием физической памяти.

Часть виртуальных страниц, принадлежащих процессу, может быть выгружена на диск и тогда в файле /dev/mem ее не окажется. При хроническом недостатке оперативной памяти, значительный процент адресного пространства процесса попадает на диск и хотя совместно используемые библиотеки вытесняются в последнюю очередь, они все-таки вытесняются (особенно редко используемые функции). Это значит, что прежде чем ковыряться в /dev/mem необходимо загрузить соответствующую функцию в память. А как это сделать? Ну, например, просто вызывать ее. Вызов функции не гарантирует загрузки всех принадлежащих ей страниц, но нам этого и не надо! Достаточно, чтобы загрузилась первая страница (а она наверняка загрузится!), чтобы воткнуть в начало функции jump на свой thunk.

Для нейтрализации побочных эффектов обычно функцию вызывают с инвалидными аргументами, чтобы она завершилась без выполнения, однако, это грязный трюк, на который ведутся далеко не все функции. В частности, gets упорно ожидает ввода с клавиатуры даже если в качестве указателя ей передать нуль. Если память, принадлежащая функции, доступна на чтение (а в linux/BSD она доступна), нам достаточно просто прочитать несколько байт от начала функции — это гарантированно загрузит принадлежащую ей страницу в физическую память. Собственно говоря, для поиска перехватываемой функции в /dev/mem нам все равно потребуется ее сигнатура, так что без чтения здесь не обойтись.

Весь вопрос в том, как определить адрес функции? Существует по меньшей мере два пути: получить указатель средствами языка Си или вызывать dlsym. В обоих случаях результаты будут различны, что следующая программа и подтверждает:

```
#include <dlfcn.h>

int a; unsigned char *x;
x = (unsigned char*) gets;
printf("\nx = gets:%08Xh",x);
for(a=0;a<0x60;a++) printf("%02X ",x[a],(a%0x10)?"":printf("\n%08Xh:",(a+x)));

x = dlopen("libc.so.6",RTLD_NOW);
printf("\n\nbase libc.so.6:%08Xh",x);
for(a=0;a<0x60;a++) printf("%02X ",x[a],(a%0x10)?"":printf("\n%08Xh:",(a+x)));

x= dlsym(x,"gets");
printf("\n\nglsym(,\"gets\"):%08Xh",x);
for(a=0;a<0x60;a++) printf("%02X ",x[a],(a%0x10)?"":printf("\n%08Xh:",(a+x)));
printf("\n");
```

Листинг 3 программа get_addr.c, определяющая адрес функции gets

Компилируем ("`gcc get_addr.c -o get_addr -ldl`") и запускаем полученный файл на выполнение (ключ -ldl подключает библиотеку dl, экспортирующую функции dlopen и dlsym). На мыщъхином компьютере результат выглядит так:

```
x = gets:08048364h
08048364h:FF 25 A8 98 04 08 68 08 00 00 00 E9 D0 FF FF FF
08048374h:FF 25 AC 98 04 08 68 10 00 00 00 E9 C0 FF FF FF
08048384h:FF 25 B0 98 04 08 68 18 00 00 00 E9 B0 FF FF FF
08048394h:FF 25 B4 98 04 08 68 20 00 00 00 E9 A0 FF FF FF
080483A4h:00 00 00 00 00 00 00 00 00 00 00 00 31 ED 5E 89
080483B4h:E1 83 E4 F0 50 54 52 68 90 86 04 08 68 30 86 04

base libc.so.6:400179E8h
400179E8h:00 C0 02 40 D8 79 01 40 30 B5 15 40 6C 66 01 40
400179F8h:88 77 01 40 10 7C 01 40 00 00 00 00 30 B5 15 40
40017A08h:80 B5 15 40 78 B5 15 40 50 B5 15 40 58 B5 15 40
40017A18h:60 B5 15 40 00 00 00 00 00 00 00 00 00 00 00 00
40017A28h:68 B5 15 40 70 B5 15 40 40 B5 15 40 48 B5 15 40
40017A38h:38 B5 15 40 00 00 00 00 00 00 00 00 98 B5 15 40

glsym(,"gets"):4008CE60h
4008CE60h:55 89 E5 57 56 53 83 EC 2C 8B 75 08 E8 AC 4D FB
4008CE70h:FF 81 C3 AF E7 0C 00 C7 45 E0 00 00 00 00 8B 93
4008CE80h:5C 9C FF FF 0F B7 02 25 00 80 FF FF 66 85 C0 75
4008CE90h:0E 8B 83 34 02 00 00 85 C0 0F 85 07 01 00 00 0F
```

```
4008CEA0h:B7 02 25 00 80 FF FF 66 85 C0 0F 84 E3 00 00 00
4008CEB0h:8B 42 04 3B 42 08 0F 83 C8 00 00 00 0F B6 08 40
```

Листинг 4 результат работы программы get_addr

Указатель на функцию gets, судя по адресу (08048364h), смотрит на секцию .plt, то есть находится во "владениях" текущего процесса. Первые байты функции равны FFh 25h A8h 98h 04h 08h, что соответствует команде JMP [080498A8h]. Выходит, это еще не сама функция, а только переходник к ней! Адрес 080498A8h хранится в двойном слове лежащим в глобальной таблице смещений (got). Модификация plt/got обеспечивает перехват функции лишь в пределах текущего процесса, что с одной стороны очень даже хорошо, но с другой — весьма хреново.

Базовый адрес библиотеки libc (400179E8h) лежит в непосредственной близости от истинного адреса функции gets (4009CE60h), возвращаемым dlsym. В глаза сразу же бросается классический пролог 55h/89h E5h (PUSH EBP/MOV EBP,ESP), который мы и будем патчить для перехвата, но сперва разберемся как работать с /dev/mem.

Рисунок 3 Midnight Commander не может просмотреть файл /dev/mem

/dev/mem это необычный файл. Если в Midnight Commander'e подвести к нему курсор и нажать <F3> — ни хрена не выйдет (и не войдет)! Тоже самое произойдет если попытаться просмотреть другой файл, представляющий устройство /dev/mem (например, ранее созданный нами /home/kpnc/nezumi). Некоторые источники утверждают, что функция fopen обламывается с открытием /dev/mem и нужно юзать низкоуровневые функции операционной системы: open/read/write. Мыщъх проверил: на KNOPPIX (основан на Debian) и FreeBSD функции fopen/fread/frwite работают нормально, но, возможно, на других системах они ведут себя не так, поэтому не будем высаживаться и сделаем как говорят.

Маленький нюанс — ***под 4.5 BSD (более свежие версии не проверял) read всегда возвращает позитивный результат даже если море, тьфу, /dev/mem уже кончился, поэтому закладываться на возвращаемое ее значение нельзя.***

```
#include <fcntl.h>
#define PAGE_SIZE     0x1000

int fd;
char buf_page[PAGE_SIZE];

// открываем /dev/mem на чтение и запись
// (подробнее см. "man 2 open")
if ((fd=open("/dev/mem", O_RDWR, 0))==-1) return printf("/dev/mem open error\n");

// перемещаемся в начало (необяз.)
if (lseek(fd, 0, SEEK_SET) == -1) return printf("/dev/mem seek error\n");

// читаем 0x1000 байт в буфер
if (read(fd, buf_page, 0x1000) != 0x1000) return printf("/dev/mem read error\n");
```

Листинг 5 фрагмент программы, демонстрирующей работу с файлом /dev/mem

Кстати говоря, консольный шестнадцатеричный редактор hexedit из комплекта поставки KNOPPIX показывает /dev/mem вполне нормально, а вот графическая версия khexedit, позаимствованная оттуда же, не показывает ни хрена.

Рисунок 4 редактор khexedit не может просмотреть файл /dev/mem

А давайте проведем небольшой эксперимент! Экспериментирование — это основное занятие мыщъх'ей (после ганжа, конечно). Возьмем какую-нибудь редко используемую библиотечную функцию (например, gets) и отпатчим ее по полной программе, внедрив в начало байт C3h, соответствующей машинной инструкции RETN, а потом вызовем ее и посмотрим получилось ли у нас или нет.

Запускаем IDA PRO, загружаем libc.so.6, переходим к функции gets (<Ctrl-G>, "gets", <ENTER>) и смотрим какие байты расположены в начале функции (чтобы IDA PRO отображала машинный код рядом с инструкциями необходимо в меню Options выбрать пункт "Text representation" и в поле "Number of opcode bytes" поставить "7"). Если нет IDA PRO, содержимое функции можно определить с помощью нашей программы, приведенной в листинге 3. На мыщъхином компьютере первые 10h байт функции gets выглядят так: 55h 89h E5h 57h 56h 53h 83h ECh 2Ch 8Bh 75h 08h E8h ACh 4Dh FBh.

Рисунок 5 функция gets в дизассемблере IDA PRO

Открываем /dev/mem в hexeditor'e ("$hexedit /dev/mem"), давим <Ctrl-S> (search) и вводим эту последовательность без суффикса 'h' и без пробелов: "5589E557565383EC2C8B7508E8AC4DFB". Редактор подумает немного и выдаст результат. У мыщъх'а функция gets обнажилась в памяти по адресу 6BA8E60h. Это физический адрес и он непостоянен. Данная страница может многократно вытесняться из памяти и загружаться по совершенно другим адресам.

Рисунок 6 редактор hexedit нашел функцию gets по ее сигнатуре

Нажмем <Ctrl-S> еще раз, чтобы убедиться, что данное вхождение — единственное. Если искомая последовательность присутствует в памяти по нескольким адресам, это значит, что либо произошла коллизия (совпадение с другой функций) и тогда искомую последовательность необходимо удлинить еще на несколько байт, либо в память загружено несколько библиотек, содержащих одну и ту же реализацию функции gets (или библиотека, экспортирующая gets, попала в дисковый кэш) и тогда нам нужно обратить внимание на младшие 3 байта: у нашей функции физические и виртуальные адреса будут равны, поскольку, адрес начала страницы всегда кратен 1000h. Если же ни одного вхождения не найдено — функция gets отсутствует в памяти (не загружена библиотека или страница вытеснена на диск) и тогда мы будем должны ее загрузить. Другая причина — искомая последовательность пересекла границу страницы памяти, а, как мы уже говорили, порядок следования физических страниц не совпадает с виртуальным. В данном случае, все хорошо: между началом gets и концом физической страницы расположено PAGE_SIZE - (addres_of_func % PAGE_SIZE) = 1000h – (4008CE60h%0x1000) == 1A0h байт, что более, чем достаточно для поиска, но чтобы мы стали делать, если бы эта дистанция равнялась всего нескольким байтам?! А ничего — просто искали бы функцию в памяти не с начала самой функции, а с начала принадлежащей ей страницы, то есть: if (!memcmp(dlsym(lib_name, func_name) & 0xFFFFF000, buf_page)). В этом случае нам достаточно, чтобы между началом функции и концом страницы было всего 5 байт, необходимых для внедрения команды jump thunk. А если этих байт нет? Тогда необходимо либо искать следующую страницу и внедряться в середину функции (но это самый тяжкий вариант и здесь он не рассматривается), либо ставить в начало функции CCh и ловить исключение из ядра (см. врезку "проблемы стабильности").

Короче говоря, все проблемы решаемы, так что не будет высаживаться, а лучше подготовим тестовую программу, которая будем вызывать функцию gets. Один из вариантов реализации выглядит так:

```
char buf[666];
while(strcmp(buf,"exit")) printf(".",gets(buf));
```

Листинг 6 тестовая программа demo.c, используемая для экспериментов с функций gets

Не выходя из hex-редактора, откомпилируем ее ("`gcc demo.c -o demo`") и запустим на выполнение ("`./demo`"). Программа выполняется как и положено — ожидает ввода с клавиатуры и выходит по "exit". А вот как мы ее хакнем! Изменяем первый байт функции на C3h, сохраняем изменения по <F2> и запускаем ./demo еще раз. На этот раз, функция gets немедленно возвращает управления не обращая никакого внимания на клавиатуру и экран заполняется стройными рядами точек. Ура! У нас получилось!

Рисунок 7 это не звездное небо и не матрица, это — результат успешного хака функции gets под FreeBSD

Модификация gets воздействует как на уже запущенные, так и на в последствии запускаемые процессы, причем процессу очень сложно обнаружить, что его хакнули! (***примечание***: *если gets уже находится в ожидании ввода, то замена 55h на C3h не приводит к немедленному выходу из функции и результат будет заметен только при ее следующем вызове*).

Возникает вопрос: насколько это надежно? Что произойдет, если модифицированная страница в результате нехватки памяти отправиться в изгнание или какой-нибудь процесс

попытаться загрузить хакнутую библиотеку еще раз? Гарантирует ли операционная система непротиворечивость ситуации? Документация (см. "man mem") не дает ответа на поставленный вопрос и это правильно, поскольку модификация страниц отслеживается не самой операционной системой, а процессором. При любом записи в страниц (не важно каким путем она произошла — хоть инструкцией mov, хоть через /dev/mem), процессор устанавливает dirty-флаг, сообщая операционной системе, что при вытеснении страницы ее следует сбрасывать на диск, что операционная система и делает. Специального обработчика для поддержки "хакнутых" страниц нет, они обрабатываются так же как остальные (то есть, так как нам надо). Никакой процесс не может "перезагрузить" хакнутую библиотеку, поскольку операционная система не видит никакой необходимости считывать с медленного диска то, что уже находится в оперативной памяти. Теоретически, если все процессы выгрузят модифицированную библиотеку, спустя какое-то время операционная система действительно выбросит ее из памяти и при повторной загрузке начнет дрыгать диском. Тогда наш хак пойдет лесом, но это крайне маловероятная ситуация, которой к тому же можно противостоять путем перехвата dlclose.

Объединив все вышесказанное, мы сможем реализовать автоматический патчер, внедряющий любой код в произвольные функции. Для простоты мы ограничимся внедрением C3h в начало. Исходный текст, предлагаемый мыщъх'ем выглядит так:

```
#include <stdio.h>
#include <fcntl.h>
#include <dlfcn.h>
main(int c, char **v)
{
        #define PAGE_SIZE      0x1000          // размер страницы (не менять!)
        #define MIN_SG_SIZE    0x10            // размер сигнатуры для поиска
        #define LIB_NAME       "libc.so.6"     // перехватываемая библиотека по умолчан.
        #define FNC_NAME       "gets"          // перехватываемая функция по умолчанию
        #define MAX_MEM        (512*1024/4)    // макс. размер физ. памяти (для BSD)

        int a, fd;
        unsigned char *p;
        int f=0; char *p_lib, *p_fnc;
        int fuck_a; char fuck[]="-\\|/";
        unsigned char page_buf[PAGE_SIZE];

        // определяем что падчить
        if (c<3) p_lib=LIB_NAME,p_fnc=FNC_NAME; else p_lib=v[1],p_fnc=v[2];
        printf("patch %s::%s\n",p_lib,p_fnc);

        // определяем адрес функции для патча
        p = dlopen(p_lib,RTLD_NOW);if (!p) return printf("%s not found\n",p_lib);
        p = dlsym(p,p_fnc); if (!p) return printf("%s not found\n",p_fnc);

        // вычисляем расстояние до конца страницы
        if (((unsigned int)p)%PAGE_SIZE < MIN_SG_SIZE)
                return printf("can't find func! too close to end of the page!\n"\
                              "decrease MIN_SG_SIZE and try again!\n");

        // открываем /dev/mem
        if ((fd=open("/dev/mem",O_RDWR,0))==-1) return printf("/dev/mem open error\n");

        // перемещаемся в начало (необяз.)
        if (lseek(fd, 0, SEEK_SET) == -1) return -1;

        // ищем в /dev/mem нашу функцию
        while(fuck_a<MAX_MEM)
        {
                // читаем по одной странице пока не дойдем до конца
                if (read(fd, page_buf, PAGE_SIZE) != PAGE_SIZE) break;

                // крутим жопой, создавая видимость бурной деятельности
                printf(":%c\r",fuck[(++fuck_a)&3]);

                // сравниваем содежимое физ. памяти с первыми MIN_SG_SIZE байтами ф-ции
                if (!memcmp(&page_buf[((unsigned int)p)%PAGE_SIZE],p,MIN_SG_SIZE))
                        printf("%s found at %08Xh\n",p_fnc,
                        (a=lseek(fd,0,SEEK_CUR))-((unsigned int)p)%PAGE_SIZE,f++);
        }

        // выход, если найдено 2 или более вхождений
        if (--f) return printf("-err:don't know which page i have to fix\n"\
                               "increse MIN_SG_SIZE and try again!\n");
```

```
        // выход не найдено ни одного вхождения
        if (f) return printf("-not find\nincrease physical memory and try again\n");

        // ПАТЧИМ
        //-------------------------------------------------------------------------
        printf("OK\n\nbefore patch:\n");

        // перемещаемся на страницу назад для чтения буфера
        if (lseek(fd,a-PAGE_SIZE,SEEK_SET)==-1) return -1;
        if (read(fd,page_buf,PAGE_SIZE) != PAGE_SIZE) return -1;

        // печатаем оригинальное содержимое
        for (f=0;f<0x10;f++) printf("%02X ",page_buf[((unsigned int)p)%PAGE_SIZE+f]);

        // ставим C3h (ret)
        if (page_buf[((unsigned int)p)%PAGE_SIZE]==0xC3)
                page_buf[((unsigned int)p)%PAGE_SIZE] = 0x55;
        else
                page_buf[((unsigned int)p)%PAGE_SIZE] = 0xC3;

        // перемещаемся на ~~страницу~~страницу назад для записи буфера
        if (lseek(fd,a-PAGE_SIZE,SEEK_SET)==-1) return -1;
        if (write(fd,page_buf,PAGE_SIZE) != PAGE_SIZE) return -1;

        // печатаем отпаченное содержимое
        printf("\n\nafter patch:\n");
        if (lseek(fd,a-PAGE_SIZE,SEEK_SET)==-1) return -1;
        if (read(fd,page_buf,PAGE_SIZE) != PAGE_SIZE) return -1;

        for (f=0;f<0x10;f++) printf("%02X ",page_buf[((unsigned int)p)%PAGE_SIZE+f]);
        printf("\n\n");
}
```

Листинг 7 автоматический перехватчик mem.c, внедряющий в начало функции gets команду retn

Несколько замечаний к программе: для уменьшения количества коллизий и ускорения поиска, сравнение ведется с привязкой к смещению внутри страницы (за это отвечает конструкция page_buf[((unsigned int)p)%PAGE_SIZE]). Загрузка страниц в память происходит автоматически за счет работы memcmp. Специально заботиться об этом не надо. Программа обрабатывает ситуации, когда искомая последовательность встречается в памяти более одного раза или "разрезается" страницей напополам. В этом случае, она советует увеличить (уменьшить) константу MIN_SG_SIZE, отвечающую за размер сигнатуры, и повторить попытку еще раз. Естественно, в автономном коде (например, коде червя) необходимо организовать дополнительный цикл, который здесь не показан, чтобы не захламлять листинг второстепенными деталями.

Компилируем программу ("`gcc mem.c -O2  -o mem -ldl`") и запускаем ее на выполнение. Первый ключ командой строки — имя библиотеки, второй — имя функции, которую нужно хакнуть. При запуске без аргументов хачится функция gets из библиотеки libc.so.6. Если в начале функции уже стоит C3h, программа пытается восстановить стандартный пролог и пишет 55h, то есть работает как триггер (попытка хака функции с нестандартным прологом закончиться плачевно).

инъекция кода

Классический алгоритм внедрения shell-кода выглядит так: сохраняем несколько байт перехватываемой функции и ставим jump на свой thunk, который делает что задумано, выполняет сохраненные байты и передает управление оригинальной функции, которая может вызываться как по jump, так и по call (подробнее этот вопрос рассмотрен в статье "crackme, прячущий код на API-функциях", опубликованной в Хакере).

Самое сложное — выбрать место для размещения thunk'a. Это должна быть память доступная всем процессам, а такой памяти в нашем распоряжении нет! Мы знаем, что "подопытная" библиотека отображается на адресное пространство ~~всех~~ каждого процесс~~ов~~а, но это пространство уже занято! Наскрести пару десятков байт, отведенных под выравнивание вполне реально, только нам этого не хватит! Приходится хитрить.

Самое простое: разместить код перехватчика в какой-нибудь "ненужной" функции, например, gets, а в начало всех перехватываемых функций внедрить… нет, не jump, поскольку в

этом случае перехватчик не сможет определить откуда пришел вызов, а call gets. Внутри gets, перехватчик выталкивает из стека адрес возврата, уменьшает его на длину команды call (в 32-разрядном режиме — 5 байт) и получает искомый указатель на функцию. Зная указатель, можно определить имя функции — в этом нам поможет функция dladdr из GNU Extensions. В POSIX она не входит, но поддерживается практически всеми UNIX'ами, так что на этот счет можно не волноваться. (***Примечание**: напоминаем, что при внедрении в gets, равно как и любую другую функцию, мы можем пересекать границы страниц, поскольку за концом текущей страницы наверняка находится совсем посторонняя область памяти! если же возникает необходимость модифицировать функцию gets целиком, необходимо найти все принадлежащие ей страницы, тем же самым методом, которым мы нашли первую из них*).

Проблема в том, что dladdr находится в библиотеке libdl.x.so, которой может и не быть в памяти конкретно взятого процесса, а если она там есть, то хрен знает по какому адресу загружена. Некоторые хакеры утверждают, что в thunk-коде можно использовать только прямые вызовы ядра через интерфейс INT 80h, а все остальные функции недоступны. На самом деле это не так! Как показывает дизассемблер, dladdr это всего лишь "обертка" вокруг _dl_addr, реализованной в libc.so.x, а она-то доступна наверняка! Вот только на базовый адрес загрузки закладываться ни в коем случае нельзя и вызов должен быть относительным.

Простейшая подпрограмма генерации относительного вызова выглядит так:

```
unsigned char buf_code[]={0xE8, 0x0, 0x0, 0x0,0x0}; // call 00000h
call_r(char *lib_name, char *from, char *to, int delta)
{
        unsigned char *base, *from, *to;

        base = dlopen(lib_name,RTLD_NOW); if (!base) return -1;
        from = dlsym(base,from); if (!from) return -1;
        to   = dlsym(base,to); if (!to) return -1;

        *((unsigned int*)&buf_code[1]) = to - from - sizeof(buf_code) - delta;
        return 666;
}
```

Листинг 8 подпрограмма генерирует относительный вызов и помещает его в глобальный буфер buf_code, lib_name – имя хакаемой библиотеки, from – имя функции, из которой будет осуществляться вызов (например, gets), to – имя функции, которую нужно вызывать (например, write), delta – смещение инструкции call от начала thunk-кода

Функция call_r вызывается из программы-инсталлятора (например, нашей mem.c) и генерирует относительный вызов call по адресу from на адрес to. Она может использоваться для вызова любых функций, а не только _dl_addr.

Модернизируем программу mem.c и отпатчим функцию gets так, чтобы она выводила символ "*" на экран. Мы будем вызывать функцию write из библиотеки libc со следующими параметрами: `write(1,&"*",1)`. Обратите внимание на конструкцию &"*" — мы заталкиваем в стек символ "*" и передаем функции его указатель. А что еще остается делать? Сегмент данных ведь недоступен! Приходится использовать стек! При желании туда можно затолкать не только один символ, но и ASCIIZ-строку (только не забудьте потом вытолкнуть обратно — некоторые забывают, в результате чего имеют несбалансированный стек и получают segmentation fault).

```
// начало thunk-кода
// заталкиваем в стек аргументы функции write,
// но саму функцию еще не вызываем, т.к. не знаем ее адреса
unsigned char buf_pre[]={     0x6A,0x2A,     /* push 2Ah    */
                              0x8B,0xDC,     /* mov ebx,esp */
                              0x33,0xC0,     /* xor eax,eax */
                              0x40,          /* inc eax     */
                              0x50,          /* push eax    */
                              0x53,          /* push ebx    */
                              0x50           /* push eax    */
                              };

// сюда записывается сгенерированный относительный вызов функции write
unsigned char buf_code[]={0xE8,0x0,0x0,0x0,0x0};

// конец thunk-кода
// выталкиваем аргументы из стека вместе с символом "*"
// и возвращаемся по ret
unsigned char buf_post[]={
```

```
                                0x83,0xC4,0x10,/* add esp,10  */
                                0xC3           /* ret         */
                                };

// буфер в который будет записан собранный thunk-код в следующей последовательности:
// buf_pre + buf_code + buf_post
unsigned char buf_dst[sizeof(buf_pre)+sizeof(buf_code)+sizeof(buf_post)];


// генерируем относительный вызов write
call_r("libc.so.6", "gets", "write", sizeof(buf_pre));

// собираем thunk-код
memcpy(buf_dst,buf_pre,sizeof(buf_pre));
memcpy(buf_dst + sizeof(buf_pre), buf_code, sizeof(buf_code));
memcpy(buf_dst + sizeof(buf_pre) + sizeof(buf_code), buf_post, sizeof(buf_post));

…
// ПАТЧИМ
//--------------------------------------------------------------------------
…
// ставим C3h (ret) или восстанавливаем стандартный пролог обратно
//if (page_buf[((unsigned int)p)%PAGE_SIZE]==0xC3)
//page_buf[((unsigned int)p)%PAGE_SIZE] = 0x55;
//else page_buf[((unsigned int)p)%PAGE_SIZE] = 0xC3;

// копируем thunk-код поверх функции gets
memcpy(&page_buf[((unsigned int)p)%PAGE_SIZE],
                        buf_dst,sizeof(buf_dst));
```

Листинг 9 модернизированный вариант программы mem.c, внедряющий в начало gets вызов write(1,&"*",1);

Компилируем программу и убеждаемся, что она работает, вплотную приближая нас к созданию полноценного перехватчика. Чуть-чуть усложнив thunk-код, мы сможем не только загаживать экран, но и сохранять log в файл!

Программировать в машинных кодах очень неудобно и возникает естественное желание задействовать Си и другие языки высокого уровня. И это вполне возможно! Поскольку thunk код вызывается в контексте вызывавшего его процесса, он может загружать свои собственные динамические библиотеки, вызывая dlopen/dlsym. На машинном коде пишется лишь крохотный загрузчик, а основной код перехватчика сосредотачивается в динамической библиотеке, которую можно написать и на Си.

Кстати говоря, отказываться от функции gets совершенно необязательно и мы можем перенести ее функционал в нашу динамическую библиотеку! Только переносить необходимо именно функционал (то есть переписывать функцию заново), а не пытаться копировать код — gets вызывает "свои" подфункции по относительным адресам. При перемещении ее тела на другое место они изменяться и… здравствуй, segmentation fault!

Рисунок 8 результат работы кода, "впрыснутого" в gets (звездочки и точки идут косяками за счет буферизации)

перехват функций не во сне, а наяву

Самое сложное в перехвате — это определить границы машинных инструкций, поверх которых записывается команда перехода на перехватчик (он же thunk, расположенный в нашем случае в теле функции gets). По-хорошему, для решения этой задачи требуется написать мини-дизассемблер, но… это же сколько всего писать придется! А можно ли без него обойтись? Можно!

В начале большинства библиотечных функций расположен стандартный пролог вида PUSH EBP/MOV EBP,ESP/SUB ESP,XXXh (55h/89h E5h/ 83h ECh XXh), дающий нам пять байт — необходимый минимум для внедрения! Встречаются и другие, слегка видоизмененные прологи, например: PUSH EBP/MOV EBP,ESP/PUSH EDI/PUSH ESI (55h/89h E5h/ 57h/ 56h); PUSH EBP/MOV EAX, 0FFFFFFFFh/MOV EBP, ESP (55h/B8h FFh FFh FFh FFh/89h E5h); PUSH EBP/XOR EAX, EAX/MOV EBP,ESP (55h/31h C0h/89h E5h). Хороший перехватчик должен их учитывать.

Таким образом, наш перехватчик должен проверить первые 5 байтов перехватываемой функции и, если они совпадают со стандартным (или слегка оптимизированным) прологом, скопировать этот пролог в свое тело и выполнить его перед передачей управления оригинальной функции. А куда его можно скопировать? Сегмент данных, как уже говорилось, нам недоступен, стек трогать нельзя (перед передачей управления на функции он должен быть восстановлен), а сегмент кода запрещен от модификации.

Существует по меньше мере три решения: во-первых, мы можем вызывать функцию mprotect, присвоив кодовой странице атрибут writable, (но это некрасиво), во-вторых, трогать стек все-таки можно: забрасываем пролог на верхушку, забрасываем туда же копию всех аргументов (а сколько у функции аргументов? да хрен его знает, вот и приходится копировать с запасом) и передаем ей управление как ни в чем не бывало (но это уже не просто "некрасиво", это вообще уродство). В-третьих, мы можем поступить так:

```
// "коллекция" разнообразных прологов для сравнения
unsigned char prolog_1[]={0x55h,0x89,0xE5,0x83,0xEC};
unsigned char prolog_2[]={0x55,0x89,0xE5,0x57,0x56};

// буфер в который будет записан сгенерированный код
unsigned char buf_code[1024];

// определяем адрес перехватываемой функции
p = msym(base, fnc_name);

// если в начале перехватываемой функции расположен prolog_1
// внедряем в ее начало call на prepare_prolog_1
if (!memcmp(p,prolog_1,sizeof(prolog_1))
        call_r(base, fnc_name, "gets", 0);

// если в начале перехватываемой функции расположен prolog_2
// внедряем в ее начало call на prepare_prolog_2
if (!memcmp(p,prolog_1,sizeof(prolog_2))
        call_r(base,fnc_name,"gets", offset prapare_prolog_2-offset prepare_prolog_1);
```

Листинг 10 фрагмент программы-инсталлятора, анализирующей пролог перехватываемой функции и устанавливающей обработчик с соответствующим прологом

```
; // заносим номер "нашего" пролога в регистр EAX,
; // чтобы перехватчик знал какой ему пролог эмулировать
; // ВНИМАНИЕ! этот код засирает EAX и не работает на fastcall-функциях,
; // для поддержки которых регистры трогать нельзя, а номер пролога класть на стек,
; // восстанавливая его перед передачей управления оригинальной функции
prepare_prolog_1:
        MOV EAX, 0x1
        JMP short do_begin

prepare_prolog_2:
        MOV EAX, 0x2
        JMP short do_begin

prepare_prolog_n:
        MOV EAX, 0x2
        JMP do_begin

do_begin:
        // ОСНОВНОЙ КОД ПЕРЕХВАТЧИКА
        // ДЕЛАЕМ ЧТО ЗАДУМАНО
        // [ESP+4]+5 содержит адрес вызванной функции
        // это поможет нам отличить перехваченные функции друг от друга
        …
        …
        …
        // ПЕРЕДАЧА УПРАВЛЕНИЯ ПЕРЕХВАЧЕННОЙ ФУНКЦИИ
        // С ЭМУЛЯЦИЕЙ ЕЕ "РОДНОГО" ПРОЛОГА
        DEC EAX
        JZ prolog_1
        DEC EAX
        JZ prolog_2
        …

prolog_1: ; // эмулируем выполнение пролога типа PUSH EBP/MOV EBP,ESP/SUB ESP,XXX
        PUSH EBP
        MOV  EBP,ESP
```

```
        SUB ESP, byte ptr [EAX]       ; берем XXh из памяти
        INC EAX                       ; на след. машинную команду
        JMP EAX

prolog_2: ;// эмулируем выполнение пролога типа PUSB EBP/MOV EBP,ESP/PUSH EDI/PUSH ESI
        PUSH EBP
        MOV EBP, ESP
        PUSH EDI
        PUSH ESI
        JMP EAX
```

Листинг 11 базовый код перехватчика (расположенный в gets), поддерживающий несколько различных прологом

Программа-инсталлятор анализирует пролог перехватываемой функции и, в зависимости от результата, внедряет в ее начало либо call prepare_prolog_1 либо call prepare_prolog_2, где prepare_prolog_x – метка, расположенная внутри thunk-кода, помещенного нами в функцию gets. Команда call занимает 5 байт и потому в аккурат накладывается на команду SUB ESP,XXh так, что XXh оказывается прямо за ее концом. Поэтому, сохранять XXh в теле самого перехватчика не нужно!!! Команда SUB ESP, byte ptr [EAX], вызываемая из thunk-кода эмулирует выполнение SUB ESP,XXh на ура!

Приведенный пример портит регистр EAX и работает только с cdecl и stdcall функциями. Перехват fastcall-функции, передающих аргументы через EAX, по этой схеме невозможен. Однако, оригинальный EAX можно сохранять в стеке и восстанавливать непосредственно перед передачей управления перехваченной функции, но в этом случае JMP EAX придется заменить на RETN, а на верхушку стека предварительно положить адрес для перехода.

Вот, собственно говоря, и все. Скелет перехватчика успешно собран и готов к работе. Остается дописать "боевую начинку". Это может быть и логгер, протоколирующий вызовы, и анти-протектор, блокирующий вызовы некоторых функций (например, удаление файла), и макро-машина, "подсовывающая" функциям клавиатурного ввода готовые данные и… да все что угодно!

>>> врезка проблемы стабильности

Сконструированный нами перехватчик довольно капризен по натуре и периодически падает без всяких видимых причин. Почему это происходит? Рассмотрим функцию func со стандартным прологом вида PUSH EBP/MOV EBP,ESP. Допустим, процесс A выполнил команду PUSH EBP и только собирался приступить к выполнению MOV EBP,ESP как был прерван системным планировщиком и управление получил наш процесс B, осуществляющий перехват функции func путем внедрения в ее начало инструкции call. Когда процесс A возобновит свое выполнение, команды MOV EBP ESP там уже и не окажется, а будет торчать хвостовая часть от call при выполнении которой все пойдет в разнос. Конечно, вероятность такого события исчезающе мала, но в особо ответственных случаях с ней все-таки стоит считаться.

Чтобы "обезопасить" перехват, необходимо сократить длину внедряемой инструкции до одного байта, но… таких инструкций просто нет! То есть как это нет? А INT 03h (CCh) на что? Традиционно она используется для организации точек останова и на прикладном уровне защищенного режима возбуждает исключение, которое легко перехватить из ядра, а точнее из загружаемого модуля. Об этом уже писалось в статье "Handling Interrupt Descriptor Table for fun and profit", опубликованной в 59 номере phrack, так что не будем повторяться.

Заметим, что CCh конфликтует с некоторыми защитными механизмами и, естественно, с отладчиками, поэтому лучше внедрять не INT 03h, а какую-нибудь "запрещенную" однобайтовую команду типа CLI (FAh), возбуждающую исключение, которое мы будем отлавливать.

>>> врезка точки останова

Отладчики могут устанавливать программные точки останова на библиотечные функции, внедряя в их начало команду INT 03h (CCh). В этом случае наш перехватчик не сможет распознать пролог, что не есть хорошо.

Выход очевиден — сравнивать только 2й, 3й, 4й и 5й байты пролога, игнорируя 1й байт. А что делать с точкой останова? Если записать call поверх нее, то она будет затерта и отладчик потеряет контроль за функцией, что в некоторых случаях неприемлемо и тогда

необходимо внедряться со 2го байта, но в этом случае команда call полностью затрет SUB ESP,XXh и XXh придется сохранять где-то в другом месте.

заключение

Механизмы перехвата API-функций под windows хорошо исследованы и предложить радикально новый трюк довольно трудно. UNIX-системы исследованы намного хуже и таят множество нераскрытых возможностей, притягивающий хакеров и прочих творческих людей. Описанный мыщъх'ем способ — не единственный и, вероятно, не самый удобный, к тому же до "промышленного применения" ему еще расти и расти. Тем не менее, в мыщъхиных утилитах, написанный на скорый хвост, он вполне нормально работает — как под linux'ом, так и под BSD.