Современные rootkit-технологии в Linux

Дмитрий Пукаленко
xdiman@gmail.com

Вступление

Механизмы функционирования руткитов для Linux - интересная и мало освещенная тема. Rootkit-технологии используются для таких задач, как сокрытие действий атакующего, подмена выдачи браузера, прозрачное проксирование, балансировка нагрузки, защита системы от атак.

Традиционно руткиты подразделяются на два больших класса: руткиты, работающие в пространстве пользователя (user-mode) и руткиты уровня ядра (kernel-mode). В данной статье я рассматриваю основные методы и нюансы реализации руткитов обоих типов.

User-mode руткиты

Руткиты, работающие непосредственно в пространстве пользователя, стары как мир. Большинство из них функционируют за счет модификации утилит администрирования (ls, cat, ps, top, netstat и др.), обеспечивающей фильтрацию их вывода и сокрытие заданных файлов, настроек, сетевых подключений и другой системной информации. Ярким примером такого руткита является shv.

Подход, основанный на модификации исполняемых файлов, несет в себе массу недостатков. Во-первых, он не универсален: для обнаружения скрытых объектов достаточно воспользоваться утилитой, не входящей в стандартный набор администратора - например, просмотреть содержимое каталога в Midnight Commander. Во-вторых, все современные IDS (например, Tripwire) периодически сверяют целостность бинарных файлов по размеру и контрольной сумме. Как следствие описанных недостатков, данный подход неприменим в современных системах. Рассмотрим его альтернативы.

User-mode руткит: перехват вызовов с помощью ptrace()

Система Linux предоставляет программисту широкие возможности для отладки приложений с помощью системного вызова ptrace(). Этот syscall позволяет получить практически полный контроль над сторонним процессом. Он имеет следующий прототип:

long ptrace(enum __ptrace_request request, pid_t pid, void *addr, void *data);

Первый параметр - команда отладчика, и назначение параметров addr и data меняется в зависимости от него. Для реализации руткита используются команды PTRACE_ATTACH, PTRACE_DETACH, PTRACE_GETREGS, PTRACE_SYSCALL, PTRACE_SETREGS, PTRACE_PEEKDATA, PTRACE_POKEDATA.

Рассмотрим вкратце логику работы руткита, основанного не перехвате ptrace().

1. Осуществляется перебор процессов, запущенных на данный момент в системе, и присоединение к ним, кроме проме процессов 0 и 1, руткита при помощи PTRACE_ATTACH. В дальнейшем можно присоединяться к новым процессам, отслеживая системный вызов fork(). Отлаживаемые процессы автоматически становятся потомками руткита.

2. Получение идентификатора какого-либо из потомков руткита, получившего сигнал отладчика SIGTRAP, посредством вызова waitpid(-1, &status, WUNTRACED).

3. Анализ системного вызова.

В момент отладочного брейка процесс находится непосредственно перед выполнением системного вызова. При этом номер системного вызова находится в регистре eax - regs.orig_eax, а в regs.ebx, regs.ecx, regs.edx, regs.edi, regs.esi находятся остальные параметры. Для получения значений регистров вызываем ptrace с параметром PTRACE_GETREGS. При этом в параметр data загружается адрес структуры user_regs_struct, хранящейся в <sys/user.h>.

Как фильтровать системные вызовы? Для сокрытия процессов и файлов необходимо обрабатывать те вызовы, в которых значение regs.orig_eax соответствует константе __NR_getdents (__NR_getdents64). Для подмены содержимого файлов - __NR_read, __NR_write, и т.д. Полный список системных вызовов доступен в <asm/unistd.h>. Контроль всех сетевых вызовов на 32-битной архитектуре обеспечивается перехватом единственного системного вызова __NR_socketcall, при этом номер конкретной функции находится в regs.ebx (полный список номеров функций находится в <linux/net.h>). На 64-битных архитектурах за каждую сетевую процедуру отвечает отдельный системный вызов.

Если перехваченный вызов нам интересен, можно приступить к анализу аргументов. Вызываем ptrace с пара-

метром PTRACE_PEEKDATA. При этом в параметре addr должен находиться реальный адрес в пространстве отлаживаемого процесса. Функция вернет 32-битное число (на 32-битных архитектурах), поэтому для получения полного содержимого анализируемого параметра нужно повторять в цикле процедуру вызова функции ptrace(..PTRACE_PEEKDATA..), постепенно увеличивая addr, необходимое число раз или до достижения нулевого байта, если речь идет о строке. После анализа и совершения необходимых приготовлений (например, сохранения адресов параметров и т.п.), можно переходить к следующему пункту.

4. Завершаем системный вызов:
ptrace(..PTRACE_SYSCALL..), и ожидаем его результат: waitpid(pid, &status, WUNTRACED). После чего снова вызываем ptrace(..PTRACE_GETREGS..) для получения контекста отлаживаемого процесса. В regs.eax будет получен результат системного вызова, который можно изменять - например, вернуть измененное количество прочитанных байт (для функции read()). Запись значений регистров обратно в пользовательский контекст выполняется с помощью PTRACE_SETREGS, с указанием в параметре data адреса структуры с измененными регистрами.

Далее - анализ и изменение параметров. Модификация памяти процесса выполняется с помощью PTRACE_POKEDATA, при этом в качестве параметра addr указывается адрес в пространстве процесса, в качестве параметра data - значение (обратите внимание: значение, а не адрес!) для записи по данному адресу. Постепенно инкрементируя addr, запишем необходимые данные в пространство процесса.

5. Выполняем инструкцию PTRACE_SYSCALL и возвращаем процессу управление, после чего ожидаем очередного сигнала от процесса-потомка при помощи waitpid(-1, &status, WUNTRACED).

Это базовый алгоритм ptrace-руткита, не учитывающий нюансов реализации. Рассмотрим основные проблемы и подводные камни, возникающие в процессе разработки.

Выделение памяти в адресном пространстве чужих процессов. Для этого можно применить грязный хак, заключающийся в следующем. На этапе инициализации процесс часто совершает системный вызов mmap2() для постраничного выделения памяти и munmap - для освобождения. Перехватив mmap2, можно модифицировать параметр, хранящийся в regs.edx: добавить туда PROT_WRITE|PROT_EXEC, и при необходимости увеличить размер запрашиваемой памяти. Когда же процесс вызовет munmap - можно модифицировать второй параметр вызова, выставив его равным 0, или даже подменить регистры так, чтобы была вызвана функция brk(), что приведет к выделению дополнительной памяти. Теперь у нас есть блок памяти в адресном пространстве процесса, который можно использовать для своих нужд.

Выполнение произвольного кода в контексте процесса. Для этого можно скопировать необходимый код в память процесса, полученную вышеописанным методом, сохранить контекст текущего процесса (при помощи PTRACE_PEEKUSER и PTRACE_POKEUSER, см. <sys/user.h>), изменить eip и исполнить записанный код пошагово, с помощью параметра PTRACE_SINGLESTEP, после чего восстановить сохраненный контекст.

Есть еще один интересный момент, связанный с сокрытием объектов в системе. Если администратор системы, в которой установлен руткит, попытается отладить какую-либо программу с помощью gdb или просто протрассировать ее при помощи strace, он получит -EPERM (Operation not permitted), что выдаст присутствие руткита. Для предотвращения такой возможности можно перехватывать сам системный вызов ptrace, делать PTRACE_DETACH, и помещать pid отлаживаемого процесса и pid отладчика в очередь до тех пор, пока отладчик повторно не вызовет ptrace с параметром PTRACE_DETACH. После чего можно снова присоединяться к процессу и продолжать контролировать его выполнение.

User-mode руткит: LD_PRELOAD

Переменная окружения LD_PRELOAD позволяет подгрузить свою библиотеку и подменить адреса библиотечных процедур, используемых в программе. Данную особенность системы также можно использовать с целью сокрытия тех или иных объектов. Недостаток этого метода в том, что перехватываются библиотечные функции, используемые программой, поэтому он узко специфичен. Но есть плюсы - например, перехватив SSL_read и SSL_write, можно читать и модифицировать незашифрованный трафик в обход SSL-шифрования!

Для использования данного метода переопределим в нашей .so библиотеке те функции, которые необходимо перехватывать, и обеспечим вызов оригинальной функции посредством ее динамической подгрузки из настоящей библиотеки с помощью dlopen/dlsym. Скомпилировав библиотеку, пропишем ее отдельной строкой в /etc/ld.so.preload, в результате чего все запускаемые впоследствии процессы будут работать с переопределенной функцией.

Что делать, если программа получает адреса функций динамически? В таком случае можно перехватывать вызовы dlopen и dlsym, и при совпадении параметров dlsym с перехватываемой функцией указывать адрес подставной процедуры.

Поскольку практически все программы используют функцию readdir для чтения элементов директории, ничто не мешает скрывать файлы и процессы, равно как и сам файл /etc/ld.so.preload.

Kernel-mode руткиты

User-mode руткиты обладают существенным недостатком: их очень легко обнаружить из режима ядра. При этом работа руткита в режиме ядра, напротив, предоставляет неограниченные возможности для манипуляции системой и оборудованием.

Самый распространенный метод, обеспечивающий функционирование руткита уровня ядра - это перехват системных вызовов путем подмены соответствующей записи в таблице системных вызовов sys_call_table. Детали этого метода заключаются в следующем. При обработке прерывания int 0x80 (или инструкции sysenter) управление передается обработчику системных вызовов, который после предварительных процедур передает управление на адрес, записанный по смещению %eax в sys_call_table. Таким образом, подменив адрес в таблице, мы получаем контроль над системным вызовом. Этот метод имеет свои недостатки: в частности, он легко детектируется антируткитами; таблица вызовов в современных ядрах не экспортируется; и кроме того, перехват некоторых системных вызовов (например, execve()) нетривиален.

Другим распространенным механизмом в kernel-mode руткитах является патчинг VFS (Virtual Filesystem Switch). Этот подход применяется в рутките adore-ng. Он основан на подмене адреса какой-либо из функций-обработчиков для текущей файловой системы.

Как и в Windows, широко используется сплайсинг - замена первых байтов кода системного вызова на инструкцию jmp, осуществляющую переход на адрес обработчика руткита. В коде перехвата обеспечивается выполнение проверок, возврат байтов, вызов оригинального кода системного вызова и повторная установка перехвата. Данный метод также легко детектируется.

Еще один метод - подмена самого обработчика прерываний в IDT.

Рассмотрим некоторые методы реализации kernel-mode руткита более подробно.

Kernel-mode руткит: подмена системных вызовов

Краеугольный камень этого метода - поиск адреса таблицы системных вызовов sys_call_table. Как уже упоминалось выше, в ядрах версии 2.6 таблица не экспортируется.

Первый способ решения этой проблемы - поиск таблицы в /proc/kallsyms.

```
# cat /proc/kallsyms | grep sys_call_table
c03596e0 R sys_call_table
```

Если администратор системы отключил kallsyms, то этот метод не сработает.

Существует также метод, основанный на поиске известного нам адреса sys_close там, где, предположительно, может находиться таблица. [1]

```
unsigned long *ptr;
unsigned long arr[4];
int i;
ptr=(unsigned long *)((init_mm.end_code + 4) &
0xfffffffc);

while((unsigned long)ptr < (unsigned
long)init_mm.end_data)
{
   if (*ptr == (unsigned long *)sys_close)
   {
      for(i=0;i<4;i++)
      {
         arr[i]=*(ptr+i);
         arr[i]=(arr[i] >> 16) & 0x0000ffff;
      }
      if(arr[0] != arr[2] || arr[1] != arr[3])
      {
         sys_call_table=(ptr - __NR_close);
         break;
      }
   }
   ptr++   ;
}
printk(KERN_INFO"sys_call_table base found at:
%x\n",sys_call_table);
```

Этот метод не дает стопроцентной гарантии обнаружения таблицы системных вызовов.

Зная, что реализация обработчика системных вызовов не менялась уже давно, мы можем использовать следующий алгоритм:

- получить адрес IDT;
- получить адрес обработчика прерывания 0x80;
- начать искать с этого адреса инструкцию call: там и будет искомый адрес таблицы.

Код должен получиться примерно таким: [2]

```
main ()
{
    unsigned sys_call_off;
    unsigned sct;
    char sc_asm[CALLOFF],*p;

    /* well let's read IDTR */
    asm ("sidt %0" : "=m" (idtr));
    printf("idtr base at 0x%X\n",(int)idtr.base);

    /* now we will open kmem */
    kmem = open ("/dev/kmem",O_RDONLY);
    if (kmem<0) return 1;

    /* read-in IDT for 0x80 vector (syscall) */
    readkmem (&idt,idtr.base+8*0x80,sizeof(idt));
    sys_call_off = (idt.off2 << 16) | idt.off1;
    printf("idt80: flags=%X sel=%X off=%X\n",
           (un-
signed)idt.flags,(unsigned)idt.sel,sys_call_off);

    /* we have syscall routine address now,
       look for syscall tabledispatch (indirect
call) */
    readkmem (sc_asm,sys_call_off,CALLOFF);
```

```
    p = (char*)memmem
(sc_asm,CALLOFF,"\xff\x14\x85",3);
    sct = *(unsigned*)(p+3);
    if (p) {
            printf ("sys_call_table at 0x%x, call
dispatch at 0x%x\n", sct, p);
    }
    close(kmem);
}
```

Этот код ориентирован на работу с памятью ядра из адресного пространства пользователя, однако может быть легко модифицирован для работы в режиме ядра. На сегодняшний день он является самым универсальным и надежным методом поиска таблицы системных вызовов.

Еще одна проблема метода перехвата системных вызовов заключается в невозможности перехватить execve() для контроля над создаваемыми процессами. Для ее решения можно использовать грязный хак: поиск свободного слота в sys_call_table, запись туда старого адреса execve(), перехват, и последующие вызовы через int 0x80.

Антируткиты, основанные на контроле целостности таблицы, обходятся простым поиском в пространстве ядра сохраненной копии таблицы и последующим ее изменением.

Kernel-mode руткит: обходимся без LKM

Возможность загрузки дополнительных модулей в ядро может быть отключена при его сборке. В таких случаях используется патчинг ядра "на лету", подробно описанный в статье Phrack #58, Linux on-the-fly kernel patching without LKM [3]. Рассмотрим здесь основные принципы этого метода.

Во-первых, производится поиск таблицы системных вызовов. После этого необходимо найти адрес kmalloc для выделения памяти в пространстве ядра. Лучше всего это сделать через kmalloc, но в крайнем случае можно воспользоваться ненадежным методом брутфорса, основанным на поиске самой часто вызываемой функции со вторым параметром GFP_KERNEL. Для примера приведен код из статьи[3], модифицированный для ядер 2.6.x. (См. Листинг 1).

При использовании этого кода следует учитывать, что значение GFP_KERNEL различно в разных версиях ядра.

После определения адресов можно временно заменить какой-либо системный вызов (например, olduname()) своим кодом с прошитым в нем адресом kmalloc для выделения необходимой памяти, в которую впоследствии будет скопирован код новых системных вызовов. После успешного выполнения этой процедуры и копирования кода новых системных вызовов в ядро производится подмена значений в sys_call_table в соответствии со стандартным алгоритмом.

Резюме

В этой статье рассмотрены основные технологии, применяемые в современных ring3 и ring0 руткитах для Linux. Некоторые методы остались за пределами статьи: например, патчинг VFS и смена обработчика прерываний - о них будет рассказано в следующей статье.

Список литературы

[1] dev0id, "Защита от исполнения в стеке (ОС Линукс)" http://www.securitylab.ru/contest/212111.php

[2] Phrack #58, "Linux on-the-fly kernel patching without LKM"
http://www.phrack.org/issues.html?issue=58&id=7#article

Листинг 1

```
unsigned long get_kma(int kmem, ulong pgoff)
{
    struct { uint a, f, cnt; } rtab[RNUM], *t;
    unsigned int       i, a, j, push1, push2;
    unsigned int       found = 0, total = 0;
    unsigned char      buf[0x10010], kcache[10], *p;
    unsigned long      ret;
    unsigned long      gfp;
    unsigned long  eax;
    unsigned long  cnt  = 0;

    // get kernel symbol directly via get_kernel_syms
    ret = get_sym("kmalloc");
    if (ret) return ret;

    // determine GFP_KERNEL value, varying on kernel version
    gfp = get_gfp();

    // try to bruteforce kmalloc address ;)
    for(i=(pgoff+0x100000);i<(pgoff+0x1000000);i+=0x10000)
    {
        // read data from kernel memory
       if (!readkmem(kmem, buf, i, sizeof(buf))) return 0;

           // look for mov GFP_KERNEL, edx and call
           for(p=buf;p<buf+0x10000;)
           {
           // look at opcode
               switch (*p++)
               {
               case 0xa1:
               // probably, mov to eax
               // so let's store eax value to variable
               // kmalloc probably does not take very big args
                   eax = *(unsigned*)p;
                   continue;

               case 0xba:
               // mov found, check for GFP_KERNEL
                   if(!memmem(p, 4, &gfp, 4)) continue;
                   push1 = *(unsigned*)p;
                   p += 4;
                   continue;

               case 0xe8:
               // call found, if mov was saved then break
                   if (push1) break;

               default:
               // some other shit
                   if(push1 && cnt < 0)
                   {
                       cnt++;
                       continue;
                   }
                   else cnt = 0;
                   push1 = 0;
                   continue;
```

```
                }

            // we have mov and call instructions, let's get address
            // and check if it is in kernel code section
            a = *(unsigned *) p + i + (p - buf) + 4;
            if((a & 0xF0000000) != pgoff) continue;

            // kmem_page_alloc test
            // very very bad and platform-dependent
            if(readkmem(kmem, kcache, a, 4))
            if(memmem(kcache, 4, "\x57\xf6\xc2\x10", 4)) continue;

#ifdef DEBUG
            /*printf("%x %0.2x %0.2x %0.2x %0.2x eax=0x%x\n",
                                        i+(p-buf),
                                        ((unsigned char*)&a)[0],
                                        ((unsigned char*)&a)[1],
                                        ((unsigned char*)&a)[2],
                                        ((unsigned char*)&a)[3],
                                        eax);*/
#endif
            p += 4;
            total++;

            // find address in table
            for (j = 0, t = rtab; j < found; j++, t++)
                if (t->a == a && t->f == push1) break;
            if (j < found) t->cnt++;
            else if (found >= RNUM)
            {
                return 0;
            }
             else
            {
                found++;
                t->a = a;
                t->f = push1;
                 t->cnt = 1;
            }
            push1 = push2 = 0;
        }
    }

    t = NULL;

    // find function called maximum number of times
    for (j = 0;j < found; j++) if (!t || rtab[j].cnt > t->cnt) t = rtab+j;

    if (t) return t->a;

    return 0;
}
```