Przepełnienia jądra Unix

Omówimy luki na poziomie jądra oraz rozwój solidnych, niezawodnych exploitów dla jądra Unix. Zidentyfikujemy kilka ogólnych problemów w różnych jądrach, które mogą prowadzić do warunków do wykorzystania, a my przedstawimy kilka przykładów znanych błędów. Po zapoznaniu się z różnymi typami luk w jądrze, kontynuujemy rozdział, skupiając się na dwóch exploitach, które zostały znalezione w systemach operacyjnych OpenBSD i Solaris podczas wstępnych badań przeprowadzonych na potrzeby tego rozdziału. Omawiane przez nas luki skutkują dostępem na poziomie jądra do zasobów systemu operacyjnego we wszystkich wersjach OpenBSD i Solaris. Dostęp na poziomie jądra ma dość poważną konsekwencję w postaci łatwej eskalacji uprawnień, a co za tym idzie, całkowitego złamania wszelkich rodzajów wymuszania bezpieczeństwa na poziomie jądra, takich jak chroot, systrace i wszelkich innych produktów komercyjnych, które zapewniają funkcje systemu operacyjnego zaufanego B1. Będziemy również kwestionować proaktywne bezpieczeństwo OpenBSD i jego niepowodzenie w walce z exploitami na poziomie jądra. Miejmy nadzieję, że da ci to motywację i ducha do atakowania innych rzekomo bezpiecznych od podstaw systemów operacyjnych.

Rodzaje luk w zabezpieczeniach jądra

Istnieje wiele funkcji i złych praktyk kodowania, które mogą prowadzić do nadających się do eksploatacji warunków w ziarnie. Omówimy te słabości i podamy przykłady z różnych jąder, podając wskazówki, na co zwracać uwagę podczas przeprowadzania audytów. Znakomity artykuł i audyt Dawsona Englera „Using Programmer-Written Compiler Extensions to Catch Security Holes" dostarcza doskonałych przykładów tego, czego należy szukać podczas polowania na luki w kernelu. Chociaż zidentyfikowano wiele możliwych złych praktyk kodowania specyficznych dla luk na poziomie jądra, niektóre potencjalnie niebezpieczne funkcje wciąż były pomijane, nawet podczas rygorystycznych audytów kodu. Przepełnienie stosu jądra OpenBSD, przedstawione w tym rozdziale, należy do kategorii funkcji rzadko audytowanej. Kernel Land zawiera potencjalnie niebezpieczne funkcje, które mogą być źródłem przepełnień, podobnie jak interfejsy API środowiska użytkownika strcpy i memcpy. Te funkcje i różne błędy logiczne można wyabstrahować w następujący sposób:

■■ Problemy z liczbami całkowitymi ze znakiem

■■ luki w zabezpieczeniach buf[user_control_index]

■■ funkcje kopiowania/kopiowania

■■ Przepełnienia liczb całkowitych

■■ funkcje malloc/free

■■ funkcje kopiowania/kopiowania

■■ problemy arytmetyczne liczb całkowitych

■■ Przepełnienia bufora (stos/stos)

■■ kopiowanie i kilka innych podobnych funkcji

■■ odczyt/zapis z węzła v do bufora jądra

■■ Przepełnienia ciągu formatującego

■■ dziennik, funkcje drukowania

■■ Błędy projektowe

■■ modload, ptrace

Przyjrzyjmy się niektórym publicznie ujawnionym lukom na poziomie jądra i przeanalizujmy różne problemy związane z eksploatacją na rzeczywistych przykładach. Dwa przepełnienia jądra OpenBSD (przedstawione w Phrack 60, Artykuł 0x6), jeden wyciek informacji o jądrze FreeBSD i błąd projektowy Solarisa są przedstawione jako studia przypadków.

2.1 - OpenBSD select() kernel stack buffer overflow

```
sys_select(p, v, retval)

register struct proc *p;

void *v;

register_t *retval;

{

register struct sys_select_args /* {

syscallarg(int) nd;

syscallarg(fd_set *) in;

syscallarg(fd_set *) ou;

syscallarg(fd_set *) ex;

syscallarg(struct timeval *) tv;

} */ *uap = v;

fd_set bits[6], *pibits[3], *pobits[3];

struct timeval atv;

int s, ncoll, error = 0, timo;

u_int ni;

[1] if (SCARG(uap, nd) > p->p_fd->fd_nfiles) {

/* forgiving; slightly wrong */

SCARG(uap, nd) = p->p_fd->fd_nfiles;

}

[2] ni = howmany(SCARG(uap, nd), NFDBITS) * sizeof(fd_mask);

[3] if (SCARG(uap, nd) > FD_SETSIZE) {

[deleted]

#define getbits(name, x)

[4] if (SCARG(uap, name) && (error = copyin((caddr_t)SCARG(uap, name),

(caddr_t)pibits[x], ni)))
```

```
goto done;
[5] getbits(in, 0);
getbits(ou, 1);
getbits(ex, 2);
#undef getbits
[deleted]
```

Aby nadać sens wybranemu kodowi syscall, musimy wyodrębnić makro SCARG z plików nagłówkowych.

```
sys/systm.h:114
...
#if BYTE_ORDER == BIG_ENDIAN
#define SCARG(p, k) ((p)->k.be.datum) /* get arg from args
pointer */
#elif BYTE_ORDER == LITTLE_ENDIAN
#define SCARG(p, k) ((p)->k.le.datum) /* get arg from args
pointer */
sys/syscallarg.h: line 14
#define syscallarg(x)
union {
register_t pad;
struct { x datum; } le;
struct {
int8_t pad[ (sizeof (register_t) < sizeof (x))
? 0
: sizeof (register_t) - sizeof (x)];
x datum;
} be;
}
```

SCARG() to makro, które pobiera elementy struktury struct sys_XXX_args (XXX reprezentuje nazwę wywołania systemowego), które są jednostkami pamięci dla danych związanych z wywołaniem systemowym. Dostęp do elementów tych struktur odbywa się za pośrednictwem SCARG() w celu zachowania wyrównania wzdłuż granic rozmiaru rejestru procesora, dzięki czemu dostęp do pamięci będzie szybszy i bardziej wydajny. Wywołanie systemowe musi zadeklarować przychodzące argumenty

w następujący sposób, aby użyć makra SCARG(). Poniższa deklaracja dotyczy struktury przychodzących argumentów wywołania systemowego select():

sys/syscallarg.h: line 404

struct sys_select_args {

[6] syscallarg(int) nd;

syscallarg(fd_set *) in;

syscallarg(fd_set *) ou;

syscallarg(fd_set *) ex;

syscallarg(struct timeval *) tv;

};

Tę konkretną lukę można opisać jako niewystarczającą kontrolę argumentu nd (w przykładzie kodu można znaleźć dokładną linię kodu oznaczoną [6]), która służy do obliczania parametru długości dla parametru user-land-to-kernel operacje kopiowania na lądzie. Chociaż istnieje kontrola [1] na argumencie nd (nd reprezentuje najwyższy numer deskryptora plus jeden w dowolnym z zestawów fd_set), który jest sprawdzany z plikami p->p_fd->fd_n (liczba otwartych deskryptorów, które jest trzymać). Ta kontrola jest niewystarczająca. nd jest zadeklarowane jako oznaczone [6], więc może być podane jako ujemne; w związku z tym kontrola większa niż zostanie ominięta . Ostatecznie nd jest używane przez makro howmany() [2] w celu obliczenia argumentu długości dla operacji kopiowania ni.

#define howmany(x, y) (((x)+((y)-1))/(y))

ni = ((nd + (NFDBITS-1)) / NFDBITS) * sizeof(fd_mask);

ni = ((nd + (32 - 1)) / 32) * 4

Po obliczeniu ni następuje kolejne sprawdzenie argumentu nd [3]. To sprawdzenie również jest zaliczone, ponieważ programiści OpenBSD konsekwentnie zapominają o sprawdzaniu podpisu w argumencie nd. Sprawdź [3], aby określić, czy przestrzeń przydzielona na stosie jest wystarczająca dla następujących operacji kopiowania, a jeśli nie, to zostanie przydzielona wystarczająca ilość miejsca na stercie. Biorąc pod uwagę nieadekwatność podpisanego czeku, zdamy czek [3] i będziemy nadal korzystać z przestrzeni stosu. Wreszcie, makro getbits() [4, 5] jest zdefiniowane i wywoływane w celu pobrania dostarczonych przez użytkownika fd_sets (readfds, writefds,exceptfds - tablice te zawierają deskryptory do przetestowania pod kątem gotowości do odczytu, gotowości do zapisu lub mają wyjątkowe warunek w toku). Oczywiście, jeśli argument nd zostanie podany jako ujemna liczba całkowita, operacja kopiowania (w ramach getbitów) nadpisze fragmenty pamięci jądra, co może prowadzić do wykonania kodu, jeśli zostaną użyte pewne sztuczki przepełnienia jądra. Ostatecznie, gdy wszystkie elementy są ze sobą powiązane, ta luka przekłada się na następujący pseudokod:

vuln_func(int user_number, char *user_buffer) {

char stack_buf[1024];

if(user_number > sizeof(stack_buf))

goto error;

copyin(stack_buf, user_buf, user_number);

```
/* copyin is somewhat the kernel land equivalent of memcpy */
}
```

2.2 - OpenBSD setitimer() kernel memory overwrite

```
sys_setitimer(p, v, retval)
struct proc *p;
register void *v;
register_t *retval;
{
register struct sys_setitimer_args /* {
[1] syscallarg(u_int) which;
syscallarg(struct itimerval *) itv;
syscallarg(struct itimerval *) oitv;
} */ *uap = v;
struct itimerval aitv;
register const struct itimerval *itvp;
int s, error;
int timo;
[2] if (SCARG(uap, which) > ITIMER_PROF)
return (EINVAL);
[deleted]
[3] p->p_stats->p_timer[SCARG(uap, which)] = aitv;
}
splx(s);
return (0);
}
```

Tę lukę można sklasyfikować jako nadpisanie pamięci jądra z powodu niewystarczającego sprawdzenia kontrolowanej przez użytkownika liczby całkowitej indeksu, która odwołuje się do wpisu w tablicy struktury jądra. Liczba całkowita reprezentująca indeks została użyta do niedoszacowania struktury, a tym samym zapisania w dowolnych lokalizacjach w pamięci jądra. Było to możliwe dzięki luce w zakresie podpisu w walidacji indeksu względem stałej liczby całkowitej (która reprezentuje największą dozwoloną liczbę indeksu). Ten numer indeksu jest [1] argumentem wywołania systemowego; jest to fałszywie deklarowane jako liczba całkowita bez znaku w bloku tekstu komentarza (wskazówka, /**/) [1]. Argument, który jest faktycznie zadeklarowany jako liczba całkowita ze znakiem w wierszu 369 sys/syscallargs.h (sprawdzony w OpenBSD 3.1), umożliwia w ten sposób aplikacjom użytkownika

dostarczanie wartości ujemnej, co prowadzi do uniknięcia wykonanych kontroli walidacji przez [2]. W końcu jądro skopiuje strukturę dostarczoną przez użytkowników do pamięci jądra, używając argumentu who jako indeksu do bufora struktur [3]. Na tym etapie dokładnie obliczona liczba ujemna, której liczba całkowita umożliwia wpisanie do struktury poświadczeń procesu lub użytkownika, podnosząc w ten sposób uprawnienia. Tę lukę można przetłumaczyć na następujący pseudokod, aby zilustrować możliwy wzorzec luki w różnych jądrach:

```
vuln_func(int user_index, struct userdata *uptr) {

if( user_index > FIXED_LIMIT )

goto error;

kbuf[user_index] = *uptr;

}
```

2.3 - FreeBSD accept() kernel memory infoleak

```
int

accept(td, uap)

struct thread *td;

struct accept_args *uap;

{

[1] return (accept1(td, uap, 0));

}

static int

accept1(td, uap, compat)

struct thread *td;

[2] register struct accept_args /* {

int s;

caddr_t name;

int *anamelen;

} */ *uap;

int compat;

{

struct filedesc *fdp;

struct file *nfp = NULL;

struct sockaddr *sa;

[3] int namelen, error, s;
```

```
struct socket *head, *so;

int fd;

u_int fflag;

mtx_lock(&Giant);

fdp = td->td_proc->p_fd;

if (uap->name) {

[4] error = copyin(uap->anamelen, &namelen, sizeof (namelen));

if(error)

goto done2;

}

[deleted]

error = soaccept(so, &sa);

[deleted]

if (uap->name) {

/* check sa_len before it is destroyed */

[5] if (namelen > sa->sa_len)

namelen = sa->sa_len;

[deleted]

[6] error = copyout(sa, uap->name, (u_int)namelen);

[deleted]

}
```

Fakt, że FreeBSD akceptuje podatność wywołań systemowych, jest problemem z podpisem, który prowadzi do stanu wycieku informacji z pamięci jądra. Wywołanie systemowe accept() jest bezpośrednio kierowane do funkcji accept1() [1] tylko z dodatkowym argumentem zerowym. Argumenty z ziemi użytkownika są pakowane do struktury accept_args [2], która zawiera:

■■ Liczba całkowita reprezentująca gniazdo

■■ Wskaźnik do struktury sockaddr

■■ Wskaźnik do liczby całkowitej ze znakiem, która reprezentuje rozmiar struktury sockaddr

Początkowo [4] funkcja accept1() kopiuje wartość argumentu rozmiaru podanego przez użytkownika do zmiennej o nazwie namelen [3]. Należy zauważyć, że jest to liczba całkowita ze znakiem i może reprezentować wartości ujemne. Następnie funkcja accept1() wykonuje dużą liczbę operacji związanych z gniazdem, aby ustawić właściwy stan gniazda. To stawia gniazdo w oczekiwaniu na nowy stan połączeń. Na koniec funkcja soaccept() wypełnia nową strukturę sockaddr adresem łączącej się encji [5], która ostatecznie zostanie skopiowana do obszaru użytkownika. Rozmiar nowej struktury

sockaddr jest porównywany z rozmiarem argumentu rozmiaru użytkownika [5], zapewniając, że w buforze ziemi użytkownika jest wystarczająco dużo miejsca, aby pomieścić strukturę. Niestety, ta kontrola jest omijana, a atakujący mogą podać ujemną wartość dla namelen integer i ominąć to większe niż porównanie. To uchylanie się od sprawdzenia rozmiaru prowadzi do skopiowania dużej części pamięci jądra do bufora w przestrzeni użytkownika. Tę lukę można przetłumaczyć na następujący pseudokod, aby zilustrować potencjalny wzorzec luki w różnych jądrach:

```
struct userdata {

int len; /* signed! */

char *data;

};

vuln_func(struct userdata *uptr) {

struct kerneldata *kptr;

internal_func(kptr); /* fill-in kptr */

if( uptr->len > kptr->len )

uptr->len = kptr->len;

copyout(kptr, uptr->data, uptr->len);

}
```

Solaris priocntl() directory traversal

```
/*

* The priocntl system call.

*/

long

priocntlsys(int pc_version, procset_t *psp, int cmd, caddr_t arg)

{

[deleted]

switch (cmd) {

[1] case PC_GETCID:

...

[2] if (copyin(arg, (caddr_t)&pcinfo, sizeof (pcinfo)))

...

error =

[3] scheduler_load(pcinfo.pc_clname,

&sclass[pcinfo.pc_cid]);
```

```
[deleted]

}

int

scheduler_load(char *clname, sclass_t *clp)

{

[deleted]

[4] if (modload("sched", clname) == -1)

return (EINVAL);

rw_enter(clp->cl_lock, RW_READER);

[deleted]

}
```

Luka Solaris priocntl() jest doskonałym przykładem gatunku luk w zabezpieczeniach związanych z błędami projektowymi. Nie wdając się w niepotrzebne szczegóły, przyjrzyjmy się, jak ta luka jest możliwa. priocntl to wywołanie systemowe, które daje użytkownikom kontrolę nad planowaniem lekkich procesów (LWP), co może oznaczać albo pojedynczy LWP procesu, albo sam proces. W typowej instalacji Solarisa dostępnych jest kilka obsługiwanych klas planowania:

■■ Zajęcia w czasie rzeczywistym

■■ Zajęcia z podziałem czasu

■■ Klasa sprawiedliwego udziału

■■ Klasa o stałym priorytecie

Wszystkie te klasy planowania są zaimplementowane jako dynamicznie ładowane moduły jądra. Są one ładowane przez wywołanie systemowe prionctl na podstawie żądań z obszaru użytkownika. To wywołanie systemowe zwykle pobiera dwa argumenty z polecenia cmd user-land i wskaźnik do argumentu struktury. Luka tkwi w cmd typu PC_GETCID, który jest obsługiwany przez instrukcję case [1]. Po przesunięciu argumentu cmd następuje skopiowanie podanego przez użytkownika wskaźnika arg do odpowiedniej struktury związanej z klasą planowania [2]. Nowo skopiowana struktura zawiera wszystkie informacje dotyczące klasy planującej, jak widać z tego fragmentu kodu:

```
typedef struct pcinfo {

id_t pc_cid; /* class id */

char pc_clname[PC_CLNMSZ]; /* class name */

int pc_clinfo[PC_CLINFOSZ]; /* class information */

} pcinfo_t;
```

Interesującym elementem tej konkretnej struktury jest argument pc_clname. Jest to nazwa klasy planującej, a także jej względna ścieżka. Jeśli chcemy użyć nazwy klasy planowania myclass, wywołanie systemowe prionctl przeszuka katalogi /kernel/sched/ i /usr/kernel/sched/ w poszukiwaniu modułu jądra mycall. Jeśli go znajdzie, moduł zostanie załadowany. Wszystkie te kroki są aranżowane przez

funkcje [3] scheduler_load i [4] modload. Jak wspomniano wcześniej, nazwa klasy programu planującego jest względną nazwą ścieżki; jest dołączany do predefiniowanej ścieżki, w której znajdują się wszystkie moduły jądra. Gdy to zachowanie dołączania istnieje bez sprawdzenia warunków przechodzenia katalogów, możliwe jest podanie nazwy klasy z ../ w jej nazwie. Teraz możemy wykorzystać tę lukę i załadować dowolne moduły jądra z różnych lokalizacji w systemie plików. Na przykład argument pc_clname, taki jak ../../tmp/mojmod, zostanie przetłumaczony na /kernel/sched/../../tmp/mojmod, co pozwoli na załadowanie do pamięci złośliwego modułu jądra. Chociaż w różnych jądrach zidentyfikowano kilka innych interesujących błędów projektowych (ptrace, vfork itd.), uważamy, że ta konkretna usterka jest doskonałym przykładem podatności jądra. W chwili pisania tego tekstu luka ta mogła być zlokalizowana i wykorzystana w podobny sposób we wszystkich aktualnych wersjach systemu operacyjnego Solaris. Błąd priocntl jest ważnym odkryciem. Prowadzi nas do przyjrzenia się interfejsowi modload, który pozwala nam odkryć dodatkowe możliwe do wykorzystania słabości na poziomie jądra. Zalecamy przyjrzenie się wcześniej znalezionym lukom w jądrze i spróbowanie przetłumaczenia ich na pseudokod lub jakiś prymityw błędu, który ostatecznie pomoże ci zidentyfikować i wykorzystać własne 0day.

0-dniowe luki w zabezpieczeniach jądra

Przedstawimy teraz kilka nowych luk na poziomie jądra w głównych systemach operacyjnych, które istniały w czasie pisania tej książki. Te luki reprezentują kilka nowych technik wykrywania i wykorzystywania luk, które nigdy wcześniej nie były publikowane. OpenBSD exec_ibcs2_coff_prep_zmagic() Przepełnienie stosu Zacznijmy od przyjrzenia się interfejsowi, który umknął tylu oczom audytu:

int

vn_rdwr(rw, vp, base, len, offset, segflg, ioflg, cred, aresid, p)

[1] enum uio_rw rw;

[2] struct vnode *vp;

[3] caddr_t base;

[4] int len;

off_t offset;

enum uio_seg segflg;

int ioflg;

struct ucred *cred;

size_t *aresid;

struct proc *p;

{

...

Funkcja vn_rdwr() odczytuje i zapisuje dane do lub z obiektu reprezentowanego przez węzeł v. Węzeł v reprezentuje dostęp do obiektu w wirtualnym systemie plików. Jest tworzony lub używany do odwoływania się do pliku według nazwy ścieżki. Być może zastanawiasz się, po co zagłębiać się w ten kod systemu plików, szukając luk w jądrze? Po pierwsze, luka ta wymaga odczytania z pliku i

przechowywania go w buforze stosu jądra. Popełnia drobny błąd polegający na zaufaniu argumentowi rozmiaru podanemu przez użytkownika. Ta luka nie została zidentyfikowana w żadnym z systematycznych audytów przeprowadzonych na systemie operacyjnym OpenBSD, prawdopodobnie dlatego, że audytorzy nie byli świadomi ewentualnych problemów z interfejsem vn_rdwr(). Zachęcamy do przyjrzenia się interfejsowi API jądra i spróbowania określenia, co może być następną dużą klasą luk w jądrze, zamiast wielokrotnie szukać znanych problemów copyin/malloc. vn_rdwr() ma cztery istotne argumenty, o których musimy wiedzieć; pozostałe możemy spokojnie zignorować. Pierwszy to rw enum. Argumenty rw reprezentują tryb działania. Będzie odczytywał lub odwrotnie zapisywał do wirtualnego węzła (v-node). Dalej jest wskaźnik vp. Wskazuje na węzeł v pliku do odczytu lub zapisu. Trzecim argumentem, o którym należy pamiętać, jest wskaźnik bazowy, który jest wskaźnikiem do pamięci jądra (stos, sterta itd.). Na koniec mamy liczbę całkowitą len, czyli rozmiar pamięci jądra wskazywanej przez argument base. Argument rw UIO_READ oznacza, że vn_rdwr jest używany do odczytywania len bajtów pliku i przechowywania go w bazie pamięci jądra. UIO_WRITE zapisuje długość bajtów do pliku z bazy bufora jądra. Jak wynika z operacji, UIO_READ może być wygodnym źródłem przepełnień, ponieważ jest podobna do operacji copyin(). Z drugiej strony UIO_WRITE może prowadzić do wycieku informacji, który jest podobny do różnych problemów z copyout(). Jak zawsze, po zidentyfikowaniu potencjalnego problemu i możliwej nowej klasy błędu bezpieczeństwa na poziomie jądra, powinieneś użyć Cscope (przeglądarki kodu źródłowego) na całym drzewie źródeł jądra. Alternatywnie, jeśli kod źródłowy nie jest dostępny, możesz rozpocząć przeprowadzanie audytów binarnych za pomocą IDA Pro. Po krótkim przejściu przez funkcję vn_rdwr w jądrze OpenBSD, znaleźliśmy humorystyczny błąd jądra, który istniał we wszystkich wersjach OpenBSD w czasie pisania tej książki. Jedynym możliwym obejściem jest niestandardowa kompilacja jądra, pomijając pewne opcje kompatybilności. W terenie większość ludzi pozostawia włączone opcje kompatybilności, nawet jeśli kompilują niestandardowe jądra. Powinniśmy również przypomnieć, że opcje zgodności istnieją w bezpiecznej instalacji domyślnej.

Podatność

Luka występuje w funkcji exec_ibcs2_coff_prep_zmagic(). Oczywiście, aby zrozumieć podatność, należy najpierw zapoznać się z kodem:

```
/*
* exec_ibcs2_coff_prep_zmagic(): Prepare a COFF ZMAGIC binary's exec package
*
* First, set the various offsets/lengths in the exec package.
*
* Then, mark the text image busy (so it can be demand paged) or error
* out if this is not possible. Finally, set up vmcmds for the
* text, data, bss, and stack segments.
*/
int
exec_ibcs2_coff_prep_zmagic(p, epp, fp, ap)
struct proc *p;
```

```
struct exec_package *epp;
struct coff_filehdr *fp;
struct coff_aouthdr *ap;
{
int error;
u_long offset;
long dsize, baddr, bsize;
[1] struct coff_scnhdr sh;
/* set up command for text segment */
[2a] error = coff_find_section(p, epp->ep_vp, fp, &sh,
COFF_STYP_TEXT);
[deleted]
NEW_VMCMD(&epp->ep_vmcmds, vmcmd_map_readvn, epp->ep_tsize,
epp->ep_taddr, epp->ep_vp, offset,
VM_PROT_READ|VM_PROT_EXECUTE);
/* set up command for data segment */
[2b] error = coff_find_section(p, epp->ep_vp, fp, &sh,
COFF_STYP_DATA);
[deleted]
NEW_VMCMD(&epp->ep_vmcmds, vmcmd_map_readvn,
dsize, epp->ep_daddr, epp->ep_vp, offset,
VM_PROT_READ|VM_PROT_WRITE|VM_PROT_EXECUTE);
/* set up command for bss segment */
[deleted]
/* load any shared libraries */
[2c] error = coff_find_section(p, epp->ep_vp, fp, &sh, COFF_STYP_SHLIB);
if (!error) {
size_t resid;
struct coff_slhdr *slhdr;
[3] char buf[128], *bufp; /* FIXME */
[4] int len = sh.s_size, path_index, entry_len;
```

/* DPRINTF((“COFF shlib size %d offset %d\n”,

sh.s_size, sh.s_scnptr)); */

[5] error = vn_rdwr(UIO_READ, epp->ep_vp, (caddr_t) buf,

len, sh.s_scnptr,

UIO_SYSSPACE, IO_NODELOCKED, p->p_ucred,

&resid, p);

Funkcja exec_ibcs2_coff_prep_zmagic() jest odpowiedzialna za tworzenie środowiska wykonywania plików binarnych typu COFF ZMAGIC. Jest wywoływany przez funkcję exec_ibcs2_coff_makecmds(), która sprawdza, czy dany plik jest plikiem wykonywalnym w formacie COFF. Sprawdza również magiczny numer. Ta magiczna liczba będzie dalej używana do identyfikacji konkretnego programu obsługi odpowiedzialnego za ustawienie układu pamięci wirtualnej dla procesu. W plikach binarnych typu ZMAGIC, tym modułem obsługi będzie funkcja exec_ibcs2_coff_prep_zmagic(). Powinniśmy przypomnieć, że punktem wejścia do tych funkcji jest wywołanie systemowe execve, które obsługuje i emuluje wiele typów plików wykonywalnych, takich jak ELF, COFF i inne natywne pliki wykonywalne z różnych systemów operacyjnych opartych na systemie Unix. Funkcję exec_ibcs2_coff_prep_zmagic() można uzyskać i wykonać, tworząc plik wykonywalny COFF (typ ZMAGIC). W kolejnych sekcjach utworzymy tego typu plik wykonywalny, osadzając nasz wektor przepełnienia w złośliwym pliku binarnym. Tutaj jednak wyprzedzamy samych siebie; najpierw porozmawiajmy o podatności. Ścieżka kodu do luki jest następująca:

user mode:

0x32a54 <execve>: mov $0x3b,%eax

0x32a59 <execve+5>: int $0x80

|

|

V

kernel mode:

[ISR and initial syscall handler skipped]

int

sys_execve(p, v, retval)

register struct proc *p;

void *v;

register_t *retval;

{

[deleted]

if ((error = check_exec(p, &pack)) != 0) {

goto freehdr;

}

[deleted]

}

Porozmawiajmy o ważnych strukturach w tym fragmencie kodu. Tablica execsw przechowuje wiele struktur execsw, które reprezentują różne typy plików wykonywalnych. Funkcja check_exec() przechodzi przez tę tablicę i wywołuje funkcje odpowiedzialne za identyfikację określonych formatów wykonywalnych. es_check jest wskaźnikiem funkcji, który jest wypełniony adresem weryfikatora formatu wykonywalnego w każdym wykonywalnym programie obsługi formatu.

struct execsw {

u_int es_hdrsz; /* size of header for this format */

exec_makecmds_fcn es_check; /* function to check exec format */

};

...

struct execsw execsw[] = {

[deleted]

#ifdef _KERN_DO_ELF

{ sizeof(Elf32_Ehdr), exec_elf32_makecmds, }, /* elf binaries */

#endif

[deleted]

#ifdef COMPAT_IBCS2

{ COFF_HDR_SIZE, exec_ibcs2_coff_makecmds, }, /* coff binaries */

[deleted]

check_exec(p, epp)

struct proc *p;

struct exec_package *epp;

{

[deleted]

newerror = (*execsw[i].es_check)(p, epp);

Ponownie, ważne jest, abyś podążał za tym, co robi ten kod. Typ binarny COFF będzie identyfikowany przez element COMPAT_IBCS2 struktury execsw, a ta funkcja (es_check=exec_ibcs2_coff_makecmds) będzie stopniowo wysyłać pliki binarne typu ZMAGIC do funkcji exec_ibcs2_coff_prep_zmagic().

}

|

|

V

```
int
exec_ibcs2_coff_makecmds(p, epp)
struct proc *p;
struct exec_package *epp;
{
[deleted]
if (COFF_BADMAG(fp))
return ENOEXEC;
```

To makro sprawdza, czy format binarny to COFF, a jeśli tak, wykonanie jest kontynuowane.

```
[deleted]
switch (ap->a_magic) {
[deleted]
case COFF_ZMAGIC:
error = exec_ibcs2_coff_prep_zmagic(p, epp, fp, ap);
break;
[deleted]
}
```

|

|

V

```
int
exec_ibcs2_coff_prep_zmagic(p, epp, fp, ap)
struct proc *p;
struct exec_package *epp;
struct coff_filehdr *fp;
struct coff_aouthdr *ap;
```

Przejdźmy przez tę funkcję, abyśmy zrozumieli, co ostatecznie doprowadzi nas do przepełnienia bufora opartego na stosie. W [1] widzimy, że coff_scnhdr definiuje informacje dotyczące sekcji dla pliku binarnego COFF (nazywanego nagłówkiem sekcji), a ta struktura jest wypełniana przez funkcję

coff_find_section() [2a, 2b 2c] opartą na pytanym typie sekcji . Pliki binarne ZMAGIC COFF są analizowane odpowiednio dla nagłówków sekcji COFF_STYP_TEXT (.text), COFF_STYP_DATA (.data) i COFF_STYP_SHLIB (biblioteka współdzielona). Podczas przepływu wykonania, coff_find_section() jest wywoływana kilka razy. Struktura coff_scnhdr jest wypełniana nagłówkiem sekcji z pliku binarnego, a dane sekcji są mapowane do wirtualnej przestrzeni adresowej procesu przez makro NEW_VMCMD. Teraz nagłówek dotyczący sekcji segmentu .text jest wczytywany do sh (coff_scnhdr) [2a]. Wykonywane są różne kontrole i obliczenia, a następnie makro NEW_VMCMD, aby faktycznie zmapować sekcję do pamięci. Precyzyjne kroki zostały podjęte dla segmentu .data [2b], który utworzy kolejny obszar pamięci. Trzeci krok odczytuje nagłówek sekcji [2c], reprezentujący wszystkie połączone biblioteki współdzielone, a następnie mapuje je pojedynczo na przestrzeń adresową pliku wykonywalnego. Po odczytaniu nagłówka sekcji reprezentującego .shlib w [2c], dane sekcji są odczytywane z węzła v pliku wykonywalnego. Następnie wywoływana jest vn_rdwr() z rozmiarem zebranym z nagłówka sekcji [4] do statycznego bufora stosu, który ma tylko 128 bajtów [4]. Może to spowodować typowe przepełnienie bufora. To, co naprawdę się tutaj dzieje, to to, że dane są wczytywane do statycznego bufora stosu na podstawie rozmiaru dostarczonego przez użytkownika i danych dostarczonych przez użytkownika. Ponieważ możemy skonstruować fałszywy plik binarny COFF ze wszystkimi niezbędnymi nagłówkami sekcji i, co najważniejsze, nagłówkiem sekcji a.shlib, możemy przepełnić ten bufor. Potrzebujemy pola rozmiaru większego niż 128 bajtów, co doprowadzi nas do zniszczenia stosu OpenBSD i uzyskania pełnego wykonania kodu pierścienia 0 (tryb jądra) dowolnego ładunku dostarczonego przez użytkownika. Pamiętasz, jak powiedzieliśmy, że z tą luką wiąże się trochę humoru? Humor kryjący się za tą usterką jest ukryty w [3], gdzie zadeklarowany jest lokalny kernel storage char buf[128]:

/* FIXME */

Nie do końca koktajlowy żart, ale mimo wszystko zabawny. Mamy nadzieję, że deweloperzy OpenBSD w końcu zrobią to, co zamierzali zrobić dawno temu. Teraz, gdy dobrze rozumiesz lukę, przejdziemy do luki w systemie operacyjnym o zamkniętym kodzie źródłowym. Zademonstrujemy również kilka ogólnych technik eksploatacji jądra i kod powłoki.

Solaris vfs_getvfssw().

Luka w zabezpieczeniach przechodzenia modułu jądra ładowalnego

Jeszcze raz przyjrzyjmy się bezpośrednio zagrożonemu kodowi i wyjaśnijmy, co on robi, zanim zagłębimy się w szczegóły luki:

/*

* Find a vfssw entry given a file system type name.

* Try to autoload the filesystem if it’s not found.

* If it’s installed, return the vfssw locked to prevent unloading.

*/

struct vfssw *

vfs_getvfssw(char *type)

{

struct vfssw *vswp;

```
char *modname;
int rval;
RLOCK_VFSSW();
if ((vswp = vfs_getvfsswbyname(type)) == NULL) {
RUNLOCK_VFSSW();
WLOCK_VFSSW();
if ((vswp = vfs_getvfsswbyname(type)) == NULL) {
[1] if ((vswp = allocate_vfssw(type)) == NULL) {
WUNLOCK_VFSSW();
return (NULL);
}
}
WUNLOCK_VFSSW();
RLOCK_VFSSW();
}
[2] modname = vfs_to_modname(type);
/*
* Try to load the filesystem. Before calling modload(), we drop
* our lock on the VFS switch table, and pick it up after the
* module is loaded. However, there is a potential race: the
* module could be unloaded after the call to modload() completes
* but before we pick up the lock and drive on. Therefore,
* we keep reloading the module until we've loaded the module
* _and_ we have the lock on the VFS switch table.
*/
while (!VFS_INSTALLED(vswp)) {
RUNLOCK_VFSSW();
if (rootdir != NULL)
[3] rval = modload("fs", modname);
[deleted]
}
```

System operacyjny Solaris ma większość funkcji związanych z jądrem zaimplementowanych jako moduły jądra, które są ładowane na żądanie. Poza podstawową funkcjonalnością jądra, większość usług jądra jest zaimplementowanych jako dynamiczne moduły jądra, w tym różne typy systemów plików. Kiedy jądro otrzymuje żądanie usługi dla systemu plików, który nie został wcześniej załadowany do przestrzeni jądra, szuka możliwego dynamicznego modułu jądra dla tego systemu plików. Ładuje moduł z jednego z wcześniej wspomnianych katalogów modułów, zyskując w ten sposób możliwość obsługi żądania. Ta konkretna luka, podobnie jak luka priocntl, polega na nakłonieniu systemu operacyjnego do załadowania modułu jądra dostarczonego przez użytkownika (w tym konkretnym przypadku modułu reprezentującego system plików), uzyskując w ten sposób pełne prawa do wykonania jądra. Jądro Solaris śledzi załadowane systemy plików za pomocą tabeli przełączników plików Solaris. Zasadniczo ta tabela jest tablicą struktur vfssw_t.

```
typedef struct vfssw {

char *vsw_name; /* type name string */

int (*vsw_init)(struct vfssw *, int);

/* init routine */

struct vfsops *vsw_vfsops; /* filesystem operations vector

*/

int vsw_flag; /* flags */

} vfssw_t;
```

Funkcja vfs_getvfssw() przeszukuje tablicę vfssw[] w poszukiwaniu pasującego wpisu na podstawie vsw_name (który jest typem ciągu znaków przekazanym do funkcji). Jeśli nie zostanie znaleziony pasujący wpis, funkcja vfs_getvfssw() najpierw alokuje nowy punkt wejścia w tablicy vfssw[] [1], a następnie wywołuje funkcję translacji [2], która zasadniczo nie robi nic więcej niż parsowanie argumentu typu dla pewnych ciągów. Takie zachowanie nie ma większego znaczenia przy wykorzystywaniu luki. Na koniec automatycznie ładuje system plików, wywołując niesławną funkcję modload [3]. Podczas naszego audytu jądra stwierdziliśmy, że dwa wywołania systemu Solaris używają funkcji vfs_getvfssw() z typem podanym w przestrzeni użytkownika. Zostanie on przetłumaczony na nazwę modułu do załadowania z katalogu /kernel/fs/ lub /usr/kernel/fs/. Po raz kolejny interfejs modload może zostać zaatakowany prostymi sztuczkami z przechodzeniem przez katalogi, które umożliwią nam wykonanie jądra. Wywołania systemowe mount i sysfs zostały zidentyfikowane i skutecznie wykorzystane podczas naszych audytów. Przyjrzyjmy się teraz dwóm możliwym ścieżkom kodu, które prowadzą do vfs_getvfssw() z danymi wejściowymi kontrolowanymi przez użytkownika.

Wywołanie systemowe sysfs()

Wywołanie systemowe sysfs() jest jednym z przykładów ścieżki kodu do vfs_getvfssw(), która umożliwia wprowadzanie danych kontrolowanych przez użytkownika.

```
int

sysfs(int opcode, long a1, long a2)

{

int error;
```

```
switch (opcode) {

case GETFSIND:

error = sysfsind((char *)a1);

[deleted]

|

|

V

static int

sysfsind(char *fsname)

{

/*

* Translate fs identifier to an index into the vfssw structure.

*/

struct vfssw *vswp;

char fsbuf[FSTYPSZ];

int retval;

size_t len = 0;

retval = copyinstr(fsname, fsbuf, FSTYPSZ, &len);

[deleted]

/*

* Search the vfssw table for the fs identifier

* and return the index.

*/

if ((vswp = vfs_getvfssw(fsbuf)) != NULL) {

[deleted]
```

Wywołanie systemowe mount()

Wywołanie systemowe mount() jest kolejnym przykładem ścieżki kodu do vfs_getvfssw(), która umożliwia wprowadzanie danych kontrolowanych przez użytkownika.

```
int

mount(char *spec, char *dir, int flags,

char *fstype, char *dataptr, int datalen)
```

```
{
[deleted]
ua.spec = spec;
ua.dir = dir;
ua.flags = flags;
ua.fstype = fstype;
ua.dataptr = dataptr;
ua.datalen = datalen;
[deleted]
error = domount(NULL, &ua, vp, CRED(), &vfsp);
[deleted]
|
|
V
int
domount(char *fsname, struct mounta *uap, vnode_t *vp, struct cred
*credp,
struct vfs **vfspp)
{
[deleted]
error = copyinstr(uap->fstype, name,
FSTYPSZ, &n);
[deleted]
if ((vswp = vfs_getvfssw(name)) == NULL) {
vn_vfsunlock(vp);
[deleted]
}
```

Trzeba przyznać, że nie sprawdziliśmy wszystkich możliwych interfejsów jądra korzystających z funkcji vfs_getvfssw(), ale najprawdopodobniej to wszystko. Zachęcamy do przyjrzenia się problemom związanym z modload(), które mogą ujawnić jeszcze bardziej użyteczne interfejsy.

Podsumowanie

W tym rozdziale przedstawiliśmy metody odkrywania nowych luk w dwóch systemach operacyjnych, OpenBSD i Solaris. Zrozumienie luk w zabezpieczeniach jądra jest trudne; dlatego zachowaliśmy faktyczną eksploatację na następny rozdział. Kontynuuj tylko wtedy, gdy w pełni zrozumiesz pojęcia i luki w zabezpieczeniach opisane w tym rozdziale. Mamy nadzieję, że czytając ten rozdział, wyrobisz sobie wyczucie pewnych rodzajów luk na poziomie jądra. Konstruowanie exploitów dla luk w Solaris i OpenBSD zostawimy teraz w rozdziale 26. Przewróć stronę, aby zobaczyć trochę poważniejsza zabawa!