

《信息系统安全实验》

指导 手册

华中科技大学网络空间安全学院

二零二二年五月

实验一

软件安全

目 录

第一章	实验目标和内容	3
1.1	格式化字符串漏洞实验.....	3
1.1.1	实验目的.....	3
1.1.2	实验环境.....	3
1.1.3	实验要求.....	4
1.1.4	实验内容.....	4
第二章	实验指导.....	7
2.1	通过 GDB 查看函数栈信息	7
2.2	通过 LIBC 的基址和内部函数的相对偏移获得 RET2LIBC 的 函数地址	7

第一章 实验目标和内容

1.1 格式化字符串漏洞实验

1.1.1 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 `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。

1.1.2 实验环境

Ubuntu 16.04 LTS 32 位（SEED 1604）的 VMware 虚拟机和本实验需要的辅助代码。简单起见，实验主要集中于 32bit 的系统，如果使用其它操作系统环境（64bit），需要使用 32bit 的编译选项。

1.1.3 实验要求

- ✧ 熟悉格式化字符串漏洞利用的原理。
- ✧ 根据本实验指导书完成实验内容。
- ✧ 提交实验报告。

1.1.4 实验内容

1. 一些设置

关闭和开启 ASLR 设置：

```
$ sudo sysctl -w kernel.randomize_va_space=0
```

关闭和开启 Stack Guard 保护：

```
$ gcc -fno-stack-protector example.c  
$ gcc -fstack-protector example.c
```

关闭和开启栈不可执行：

```
For executable stack:  
$ gcc -z execstack -o test test.c  
For non-executable stack:  
$ gcc -z noexecstack -o test test.c
```

查看程序的保护措施（gdb 中）：

checksec

或者在安装了 pwntools 的系统上，可以使用：

checksec -file=/file/to/check

2. 漏洞程序

Prog1: prog1.c

Prog2: prog2.c

Prog3: server.c (该程序执行 program 程序)

3. 实验任务

✓ 任务 1：针对 prog1，完成以下任务

- (1) 改变程序的内存数据：将变量 var 的值，从 0x11223344 变成 0x66887799；
- (2) 改变程序的内存数据：将变量 var 的值，从 0x11223344 变成 0xdeadbeef；
 - a) 后半部分数据小于前半部分数据；
 - b) 为避免 print 大量字符，可以将数据分成 4 个部分分别写入（使用 %hhn ）；

注意：以上任务，需要关闭 ASLR

✓ 任务 2：针对 prog2，完成以下任务

- (1) 开启 Stack Guard 保护，并开启栈不可执行保护，通过 ret2lib 进行利用，获得 shell（可以通过调用 system("/bin/sh")）；

- (2) 尝试设置 `setuid root`，观察是否可以获得 `root shell`。
- (3) **提示：**需要查找 `ret2lic` 中的 `system` 函数和 `“/bin/sh”` 地址：

注意：以上任务，需要关闭 ASLR

✓ 任务 3：针对 `prog3`，完成以下任务

- (1) 打印栈上数据；
- (2) 获得 `heap` 上的 `secret` 变量的值；
- (3) 修改 `target` 变量成 `0xc0ffee00`
- (4) 上述步骤在首先在关闭 ASLR 的情况下进行，进一步，可尝试开启 ASLR，观察程序内存地址的变化

4. 参考资料

“`Format_String_manual.pdf`” 文档

“`formatstring-1.2`” 文档

“`Format_String`” ppt

注意： 以上任务的 `exploit` 脚本，涉及到格式化字符串漏洞利用部分的，**不要直接使用 `pwntools` 的 API。**

第二章 实验指导

2.1 通过 libc 的基地址和内部函数的相对偏移获得 ret2libc 的函数地址

获得当前程序使用的 libc library 的 path，以及 libc 的基地址：

```
ldd ./vuln_program
```

从该 path 的 libc 中，获得 system 函数的相对偏移：

```
readelf -a /usr/lib32/libc.so.6 | grep " system"
```

也可从该 path 的 libc 中，获得"/bin/sh"的偏移(安装 ropper 工具)：

```
ropper --file /usr/lib32/libc.so.6 --string "/bin/sh"
```

其它 libc 函数的地址，可以以同样的方法获得。

注意：该方法的使用前提是 ASLR 关闭，这样，libc 的基地址不会变化；如果 ASLR 开启，那么，就需要在 exploit 中首先通过内存泄露，获得该基地址。

2.2 通过 gdb 查看函数栈信息

gdb 可以用来更加清楚的查看函数的栈信息。以下面的程序为例。

```
#include <string.h>

void overflow (char* inbuf)
{
    char buf[64];
    strcpy(buf, inbuf);
}
```



```

}

int main (int argc, char** argv)
{
    overflow(argv[1]);
    return 0;
}

```

1. 使用 gdb

\$gdb test

(1) 反汇编 `overflow` 函数，可以看到，`strcpy` 函数调用在+16的位置。那么，可以在这个函数调用的后一条指令处设置断点。

`disas overflow`

```

gdb-peda$ disas overflow
Dump of assembler code for function overflow:
   0x0804840b <+0>:    push    ebp
   0x0804840c <+1>:    mov     ebp,esp
   0x0804840e <+3>:    sub     esp,0x48
   0x08048411 <+6>:    sub     esp,0x8
   0x08048414 <+9>:    push    DWORD PTR [ebp+0x8]
   0x08048417 <+12>:   lea     eax,[ebp-0x48]
   0x0804841a <+15>:   push    eax
   0x0804841b <+16>:   call    0x80482e0 <strcpy@plt>
   0x08048420 <+21>:   add     esp,0x10
   0x08048423 <+24>:   nop
   0x08048424 <+25>:   leave
   0x08048425 <+26>:   ret
End of assembler dump.

```

(2) 设置断点。

将断点设置在 `strcpy` 函数之后，目的是及时获取 `strcpy` 函数执行之后（断点处）`overflow` 函数栈结构。

`br *overflow+21`

```
gdb-peda$ br *overflow+21
Breakpoint 1 at 0x8048420
```

(3) 执行程序。程序的输入尚没有导致溢出。

```
r $(python c 'print "A"*64')
```

```
i frame
```

```
gdb-peda$ i frame
Stack level 0, frame at 0xbffec30:
 eip = 0x8048420 in overflow; saved eip = 0x804844a
 called by frame at 0xbffec60
 Arglist at 0xbffec28, args:
 Locals at 0xbffec28, Previous frame's sp is 0xbffec30
 Saved registers:
  ebp at 0xbffec28, eip at 0xbffec2c
```

“eip at 0xbffec2c”，当前活动的是 overflow 的栈帧，也就是说 0xbffec2c 保存的是 overflow 函数的 ret 地址。

(4) 查看栈的结构

查看栈地址 0xbffec2c 附近的栈内容。

```
x/64x 0xbffec20-40
```

```
gdb-peda$ x/64x 0xbffec20-40
0xbffefbf8: 0x41 0x41 0x41 0x41 0x41 0x41 0x41 0x41
0xbffefc00: 0x41 0x41 0x41 0x41 0x41 0x41 0x41 0x41
0xbffefc08: 0x41 0x41 0x41 0x41 0x41 0x41 0x41 0x41
0xbffefc10: 0x41 0x41 0x41 0x41 0x41 0x41 0x41 0x41
0xbffefc18: 0x41 0x41 0x41 0x41 0x41 0x41 0x41 0x41
0xbffefc20: 0x00 0xc3 0xf1 0xb7 0x00 0x00 0x00 0x00
0xbffefc28: 0x48 0xec 0xff 0xbf 0x4a 0x84 0x04 0x08
0xbffefc30: 0x19 0xef 0xff 0xbf 0xf4 0xec 0xff 0xbf
```

图中，下划线处即为 0xbffec2c 的内容，也就是保存的 ret 地址。同时，0x41(即字符 A，作为程序的输入导入)一直到 0xbffec1c。那么，0xbffec2c - 0xbffec1c = 16，可以知道，ret 地址在: buf 基址+64+16 处。

通过上述方法，可以更加清楚的了解函数栈的内容。