bufLab实验报告

软件52班 张迁瑞 2015013226

实验环境

实验使用操作系统为腾讯云服务器，操作系统为 Ubuntu 14.04。

所用Cookie: 0x1fb3d168。

实验准备

实验前的准备工作包括写脚本和反汇编

脚本的作用主要是简化输入流程，使得我们可以通过一条命令执行多个测试：

```
bufLab/answer.sh
#bin/bash
./hex2raw < answer1.txt > answer1-raw.txt
./bufbomb -u 2015013226 < answer1-raw.txt
......
```

我还将bufbomb反汇编得到了bufbomb.s便于查看地址：

```
objdump -d bufbomb > bufbomb.s
```

解题过程

Level 0:Candle（烛)

本关要求通过缓冲区溢出，使函数**getbuf()**执行后直接执行smoke()函数，而不执行后面的代码。

为了研究**getbuf()**的行为,我首先查看了它的汇编代码

```
8049284:    55                      push   %ebp
8049285:    89 e5                   mov    %esp,%ebp
8049287:    83 ec 38                sub    $0x38,%esp
804928a:    8d 45 d8                lea    -0x28(%ebp),%eax
804928d:    89 04 24                mov    %eax,(%esp)
8049290:    e8 d1 fa ff ff          call   8048d66 <Gets>
8049295:    b8 01 00 00 00          mov    $0x1,%eax
804929a:    c9                      leave
804929b:    c3                      ret
```

重点是4，5行，尽管先前为栈提供了0x38个字节的空间，但执行过这两行的指令之后，栈上的空间只剩下了0x28(我不会说开始我把这个当成了28/4 = 7个字节)的空间，也就是40个字节。所以，只要输入超过40个字节，正常的栈结构就会被破坏。

被破坏掉的地方，开始是保存的ebp的值(4个字节)，之后就是返回地址。也就是说，只要随便输入44个字节，最后四个字节再改成smoke()的地址，这一关就可以成功了。

于是，再去找到smoke()的地址

```
08048b04   <smoke>:
```

将08048b04按照小端模式"04 8b 04 08"放在最后四个字节，实验顺利完成！Smoke()!

值得一提的是，尽管test中有用来保护栈的canary，但因为出事的是getbuf,它连刷存在感的机会都没有。

Level 1: Sparkler(闪烁)

和上一关类似，同样是要返回一个函数fizz(int val)，唯一不同的是这个函数是带参数的，需要在覆盖的时候把参数也放进去。

同样先找到fizz的位置

```
08048b2e   <fizz>:
```

用这个地址替换掉上一关中smoke的地址。

接下来再通过汇编代码确定参数的位置（也可以通过课上讲的知识确定），这里的参数为

```
 8048b34:   8b 55 08                mov    0x8(%ebp),%edx
```

也就是在ebp + 8的位置，因此，我们需要先放四个字节的任意数据，再把自己的userid放上去。

如此一试，果然成功破关~

Level 2: Firecracker(花火)

本关的要求是，通过输入的字符串，使得程序getbuf后不返回test，而是执行bang中的代码，同时还要修改global_value的值为自己的cookie值。

与前两关相比，这一关的操作bang了许多，因为我们不仅要改变函数的返回地址，还改变了程序中的全局变量，开始有点黑客的味道了！

这一关的总体实现思路是这样的：首先通过栈溢出，覆盖掉getbuf原来的返回值，并把它改成我们想要插入的代码的地址，这样，getbuf就会在我们想让其返回的地方返回，并执行预先设置在那里的代码（即改变global_value)。在执行完这段代码之后，再通过设置正确的返回值，让程序返回bang函数。为了方便寻址，我把插入代码的位置设置为buf的首地址。

为了达到前述要求，显然需要得到global_value的地址。这里，我第一次使用gdb（前两关都是通过反汇编得到的代码找地址）进行调试。开始以为没有图形界面的调试很难受，但用起来之后却觉得还好，至少命令很直观。

```
p &global_value
$2 = (<data variable, no debug info> *) 0x804e10c <global_value>   # 返回值
```

然后，按照书上的提示，我开始用汇编写改变这个全局变量的代码(3.S):

```
movl $0x1fb3d168, %eax  ;前者是我的id
movl %eax, 0x804e10c    ;改global_value的值
```

该怎么让这些代码执行完之后返回bang呢？直接ret显然是不行的，因为我们的代码不是走正常渠道来的，显然不能用正常的方法返回。但是考虑到ret的实质是popl %eip,而%eip里面是下一条执行指令的位置，我们可以先把我们希望它回到的地址压栈，这样ret之后自然就到那里了。

于是，找到我们想让函数返回的位置，并将其压栈后返回。

```
pushl $0x08048b82
ret
```

再把这四条语句转为指令形式:

```
gcc -m32 -c 3.S
objdump -d 3.o > 3.d
```

查看3.d，将生成的文本作为这次答案的前半段。

最后，就要找buf的地址了。这个找到的地址将被放在攻击字符串的最后，以使得getbuf之后直接跳转到这个地址。

找buf的地址颇费了一番周折，本想直接 `p &buf` 拿到地址，然而并没有找到。查找资料发现，应该是局部变量被编译器用寄存器优化掉了。看来，只能用寄存器中的值推buf了。在文档的前面，我们曾提到过，buf的地址在%ebp中的值的上面0x28处，故

```
p ($ebp - 0x28)     ;越往上值越小
```

得到buf地址，放在字符串的最后，成功破关。

Level 3: Dynamite(炸药)

这一关的要求是在修改getbuf返回值的基础上，不改变原来的栈结构，仍然让其返回test,这样，程序会以为一切正常，照常执行test中在getbuf之后的代码。然而，炸弹已经悄悄被注入到了程序之中。

实验的思路和上一关很相似，不同的是，我们要恢复被破坏掉的栈状态。也就是说，我们在执行完自己的代码之后，应该把原来ebp的值复原，再返回test函数中getbuf下一条指令的地址。

首先我们尝试通过gdb获取ebp的值，值得注意的是，获得的ebp一定是test函数运行时的ebp，而不是getbuf中的。如果断点打在了getbuf中，要用*(int*)($ebp)来获取ebp指向位置的值。

```
b *0x8048bf3
r ...  #执行函数代码
p/x ($ebp)
```

拿到这个值之后，将其放在倒数第二个字节的位置，这样，我们就巧妙地打造了一个栈状态的修复器。

下面就开始着手修改函数的返回值了，首先查看test的反汇编代码，找到getbuf之后的那句指令。

```
 8048bf3:   89 45 f4                mov    %eax,-0xc(%ebp)
```

比对前面给出的test的 C语言代码，不难了解到，这里面的%eax存放的就是getbuf的返回值。为了改变这个值，我们需要在插入的代码中加入改变%eax的语句,并将上面指令的地址压入栈中，之后再返回。

```
movl $0x1fb3d168, %eax
pushl $0x8048bf3
ret
```

用同样的方法获取机器指令，放入字符串开头。

最后，再用与上一关相同的方法获取到buf的地址放在字符串最后，再次破关成功！

Level 4: Nitroglycerin(硝化甘油)

魔高一尺，道高一丈。在这一关，程序启用了“栈随机化”之盾，试图阻碍我们的侵入。然而，道高一尺，魔高一丈，我们采用了“空操作雪橇”之矛来攻破它的防御。

所谓“栈随机化”，是指在程序开始时，在栈上用随机函数分配一定大小的空间。也就是说，每一次函数执行时，栈的具体地址都是变化的，这样，我们就没有办法通过一次调试确定栈的具体位置了。

我们的任务和上一关相同：使用栈溢出的方法，改变getbufn的返回值，并使之回到testn函数中继续运行。

既然要改变返回值，那么和上一关相似的流程也是少不了的。首先通过查反汇编代码获知，需要改变的寄存器是eax。然后，在testn中有这样一段代码：

```
8048c55:    89 e5                   mov     %esp,%ebp
8048c57:    83 ec 28                sub     $0x28, %esp
```

这句话说明，在testn调用getbufn之前，esp和ebp之间是满足相差0x28的大小关系的。所以在我们的函数中，也可以利用这一点来恢复ebp。这样就不需要在字符串的结尾加入恢复ebp的代码了。

最后当然还要将返回后第一条命令地址进栈，具体代码如下：

```
mov $0x1fb3d168, %eax
lea 0x28(%esp), %ebp
push $0x8048c67
ret
```

指令共十五字节。

到这里，一切还和上一关差不太多，但是现在问题来了：这段代码放在那里呢？换句话说，攻击字符串的最后四个字节应该放什么呢?

由于栈随机化的原因，已经不会再有像上一关一样的buf固定地址了，我们的指令也不能再放在最开头。好在这一次字符串的长度给的足够多，我们可以使用“空操作雪橇”来进行攻击。

空操作，即nop(90),顾名思义，就是什么也不做的指令，它只会使程序计数器加1，使之指向下一条指令。这个特性为我们的攻击提供了方便：我们可以把注入的代码放在一大串的nop之后，这样，只要函数返回到前面的任意一个nop上，都会不停地向后执行，直到执行完我们的攻击代码。就像滑雪一样。这样，尽管地址是随机的，我们仍然可以保证攻击代码被完整执行。

那么，现在需要的就是找到buf可能出现的最大地址（也就是栈底），并把返回地址设置的比这个最大地址还大。

buf的位置是($ebp - 0x208),在用gdb调试的过程中，对于这个位置的查询返回了这些结果：

```
0x55683168, 0x55683178, 0x556830e8, 0x55683170, 0x556831c8
```

查询过程截图,这样的过程进行了五次。

```
(gdb) b getbufn
Breakpoint 1 at 0x80492a5
(gdb) r -n -u 2015013226 < answer5.txt
Starting program: /home/owen/bufLab/bufbomb -n -u 2015013226 < answer5.txt
Userid: 2015013226
Cookie: 0x1fb3d168

Breakpoint 1, 0x080492a5 in getbufn ()
(gdb) p ($ebp - 0x208)
$1 = (void *) 0x55683168 <_reserved+1036648>
(gdb) c
Continuing.
```

基于这些值，我尝试着取了比其中最大值稍大一点的0x556831e8。这样一试，这一次的攻击也成功完成了，游戏通关，Congratulations, general!

实验感想

收获

通过这次实验，我深刻理解了缓冲区溢出的原理，也明白原来hack并不是那么高深莫测的事情。只要掌握了原理，再辅以时间慢慢分析，很多程序的漏洞都会被找寻出来。

体验

五星好评。这个实验真的很精彩，名字叫bufbomb，几个任务的名称Candle,Sparkler...也和"bomb"紧密相关。成功和失败的函数提示也很好玩，像什么smoke(),fizz(嘶嘶声), dub(哑弹)...，让人在写作业的时候感觉就像在玩一个游戏，没有觉得麻烦，不知不觉就完成了实验，甚至还有些意犹未尽。

更重要的是，这个实验的负担不大，也没有让人绞尽脑汁的“自由发挥”内容。如果所有的大作业都像这样该多好啊。

最后，非常感谢老师为我们选择这个实验！