wargame behemoth writeup

转载

weixin_34101784 于 2018-03-08 10:56:30 发布 70 收藏 1
文章标签： shell 网络 运维
原文链接：https://juejin.im/post/5aa116de51882577b45e93c1

> litao3rd · 2015/04/28 10:34

> 又一期的wargame来了，这一期的*wargame*主要侧重于逆向，基本上在*gdb*下把程序的思路弄清楚了，再利用一些简单的渗透溢出技巧，就可以成功了。 Let's go!

依然是老方法，在游戏behemoth首页可以找到登陆的服务器的账号和密码。ssh登陆上去，开始我们的wargame之旅。

level 0

登陆服务器，在游戏文件夹/behemoth下可以看到全部的可执行程序，首先执行./behemoth0，这是这一关待解决的程序。

程序让我们输入一个密码，估计是将我们输入的密码和某个固定的密码做匹配，可能是加密后最匹配，谁知道它怎么做。gdb走起。

从反汇编的代码可能看到两个令人激动的函数，一个是memfrob()，通过找男人(*man*)知道，这个函数是将输入的指定长度的字符串中的字符与数字42做异或，既然是异或操作，就是可逆的。另一个函数是strcmp()，这个函数才是最令人激动的，从整个程序流程大概看到，程序通过scanf()获取用户输入，然后通过memfrob()做异或处理，然后再送入strcmp()做匹配，所以，我们只要在strcmp()函数调用处下断点，然后查看栈内容就可以得到真正的密码了。

程序需要输入的密码是eatmyshorts，执行程序，这一关就过去了。

level 1

这一关程序也是要求输入一个密码，无法得到准备密码，还是要gdb走起。

好吧，从逆向出来的汇编代码看，程序很简单，使用gets()得到用户输入，然后puts()输出"Authentication failure.\nSorry."提示结束就可以了，没有匹配，也就是没有正确的密码。不过从gets()这是一个不安全的函数，这里也没有边界检查，说明存在缓冲区溢出漏洞，这是可以利用的。

通过验证，确实存在缓冲区溢出漏洞，下面就是如何利用这个漏洞了，这个溢出利用前面的wargame已经玩得很多了。

level 2

这里程序执行似乎是要创建某个文件，多次执行发现，每次创建的文件名似乎都不相同，应该是跟pid相关。

还是看汇编代码比较容易理解程序的执行意图。

对反汇编代码做个大概解释，执行的流程如下

```
#!c
id = getpid()
s = lstat("touch " + str(id))
if(s & 0xf000 == 0x8000){
    unlink(str(id);
    system("touch " + ID);
}
sleep(0x7d0);
system("cat  " + str(id));
复制代码
```

程序在当前目录下尝试查找以自己pid为名的文件，如果不存在的话，就建立该文件，然后执行一个很长时间的sleep()，然后再打开文件。这里没有输入，不存在溢出，也没有其它很明显的漏洞。sleep(a_long_time)这个函数调用似乎没有办法越过，也没法通过修改.got.plt来尝试将sleep()替换成其它的函数。后来再参考了网上的类似writeup，发现这里在调用system()的时候使用的是相对路径，既然是相对路径，那么这个路径就是我们可以控制的，通过修改PATH环境变量，这样就可以使程序在路径搜索的时候，先搜索我们指定的路径，这样就可以将touch程序替换成我们想要执行的程序，比如执行生成一个*shell*。这样就顺利通过这一关了。

这里需要注意路径和伪 touch 程序的权限问题，开始我一直失败就是因为权限配置的不对，浪费了不少时间按。

level 3

这里还是有一个输入，既然有输入就有可能有溢出点。

从程序执行的结果来看，程序打印了我们的输入，猜测可能有缓冲区溢出，或者是格式化字符串漏洞，经过验证，确实有格式化字符串漏洞，这样就很容易了，基本上不需要看汇编代码就可以搞定了。

从上面的测试我们发现，字符串存储地址是在当前栈的第六个偏移的地方，即0x18(%esp)，这个值在gdb中也是可以看到的。接下来就是构造攻击程序了。我将shellcode放在环境变量中。

如上图所示，环境变量即shellcode保存在*0xffffd78c*中，可能存在一点偏移，不过我们有Nop sled，Return Address处保存着程序原来的返回地址，我们需要将它修改为shellcode

这里执行的cat是为了防止管道关闭，在前面的wargame也使用了这个方法。其中需要注意的是，管道的右边，一开始我使用的是相对路径，导致总是没法得到正确的结果，也不知道问题出现在哪里，后来无意中使用了绝对路径，才搞定的。这里不知道为什么，等我之后看看管道的具体原理再做记录。

level 4

这里执行的结果就给了一个提示，PID not found!，看样子程序又是跟PID相关了。

从反汇编出来的代码可以大概了解到程序的执行流程。

程序首先打开/tmp/pid文件，然后sleep(1)一秒，然后将文件内容输出，这样我们只要将文件软链接到密码文件，就可以让程序打开密码文件，同时输出文件内容了。难点在于，我们如何知道程序的pid，虽然*linux 下 pid 是递增的，但是这也无法保证每次增加的就是 1 个单位*。于是，我想到了一个*不优雅*的方法，我们先建立大量的可能的pid软链接文件，然后一直执行程序，执行程序的pid会落到我们建立的文件范围内的。

behemoth4.py

```
#!python
#!/usr/bin/env python
#coding=utf-8

import sys, os
passwd_file = "/etc/behemoth_pass/behemoth5"

if len(sys.argv) < 2:
        print "usage %s [start pid num]"
        sys.exit(-1)
try:
        start_pid = int(sys.argv[1])
except ValueError:
        print "usage %s [start pid num]"
        sys.exit(-1)

# 建立 50 个符号链接文件
for i in range(50):
        os.popen("ln -s " + passwd_file + " /tmp/" + str(i+start_pid))

# 执行 1000 次程序
for i in range(1000):
        ret = os.popen("/behemoth/behemoth4")
        ret = ret.read()
        if not "not" in ret:
                print ret
                break

# 删除所有建立的文件
for i in range(50):
        os.popen("rm /tmp/" + str(i+start_pid))
复制代码
```

主要是python下获取子进程的pid太麻烦了，要不然这个爆破的代码可以写得更优雅一点。不过不影响结果，依然爆破出来了。这里需要注意的是，start pid即程序的参数应该要选得大一点，因为程序里面建立符号链接文件启动了不少子进程，我这里在原来的基础上增加了500个数，否则容易越过。

level 5

这一关执行程序没有任何输出，没有提示，还是直接看反汇编出来的代码吧。代码

可以看到，程序首先打开了密码文件/etc/behemoth_pass/behemoth6，然后建立了localhost:1337的socket，再用sendto函数将文件内容发送出去。程序的流程很明显了，接下来我们只要监听本地端口1337就可以收到密码了。需要注意的是sendto是用UDP协议发送的，需要监听该端口的UDP数据包。

使用瑞士军刀nc进行监听，然后在另外一个shell中执行程序，nc就会输出收到的UDP数据包内容了。

shell 1

shell 2

shell 1

level 6

这一关有两个可执行程序，执行程序都没有得到任何有意义的结果，还是直接看反汇编出来的代码吧。

第一个主程序与第二个程序/behemoth/behemoth6_reader建立一个管道，然后通过管道读取，如果读到的内容等于HelloKitty，这样就会执行execl()，建立一个shell。我们再看看第二个程序。第二个程序，执行就会输出Couldn't open shellcode.txt!，看样子是要建立一个名为shellcode.txt的文件，具体还是看看汇编代码吧。

程序首先打开一个名为shellcode.txt的文件，然后将文件内容读取到动态申请的存储区，最后跳转到动态存储区，执行读取到的内容。这样就很容易理解了，我们在shellcode.txt文件中存放一段shellcode，这段shellcode只执行一个任务，就是向标准输出stdout打印一段字符串HelloKitty就可以了。

```
section .text
global _start
_start:
        mov ax, 0x7974          ; ty
        movzx eax, ax           ; zero-extend ax to 32bits
        push eax
        push 0x74694b6f         ; oKit
        push 0x6c6c6548         ; Hell
        mov ecx, esp
        xor ebx, ebx
        inc ebx
        xor edx, edx
        mov dl, 10
        xor eax, eax
        mov al, 4
        int 80h

        ; exit(0)
        xor ebx, ebx
        xor eax, eax
        mov al, 1
        int 80h
复制代码
```

上面就是我写的输出HelloKitty的shellcode程序，汇编之后就可以得到可用的shellcode程序了。需要注意的是，behemoth6_reader程序使用的也是相对路径，既然是相对路径，就是我们可以控制的。
在/tmp下建立文件夹behemoth6，将当前文件夹设置为/tmp/behemoth6，在该目录下操作就可以了。

level 7

这一关的程序也是执行没有任何输出，所以说这次的wargame主要还是看逆向能力，基本上能逆向出来程序流程，后面的问题都很容易就可以解决了。汇编出来的程序很长，就不贴了。

唯一可能有点难度的是，在汇编代码中有两次调用call 0x8048420 <>这一个函数，男人(man)告诉说，这个函数一般是<ctype.h>中的函数如isspace()、isalpha()等调用的，也就是在程序里执行的是类似的检测，再加上从其它的代码总体来看，得到结论。该程序检测argv[1]中是否有Non-alpha字符存在，如果有的话那么就有可能是shellcode，这样就提示错误。如图所示，

如上图所示，第一次执行时，argv[1]中有字符,存在，于是程序报错，提示可能有shellcode存在。

不过程序只是检测argv[1]中前256个字符，这样我们只需要用alpha字符填充前面256个位置就可以了，后面可以进行缓冲区溢出利用。不过程序中将全部的环境变量都清空了，这就意味着我们不能将shellcode放置在环境变量中，需要找其它的地方存放，同时一开始，我将shellcode放在argv[1]字符串中的尾部，加上Nop sled，执行的时候提示Illegal Struction，猜测可能栈不可执行。最终我使用return-to-libc，将返回地址修改成system()函数的地址，同时将参数/bin/sh放置在argv[2]中，加上一定的猜测，最终搞定了。

end of the game

又完成一期wargame了，现在的难度不是很大，都是一些很基础的逆向、溢出的知识，不过作为一个新手，我深深地知道打好基础才是最重要的，后面依然有很多好玩的。尽请期待～

想了解更多关于wargame的内容，请参考这里！