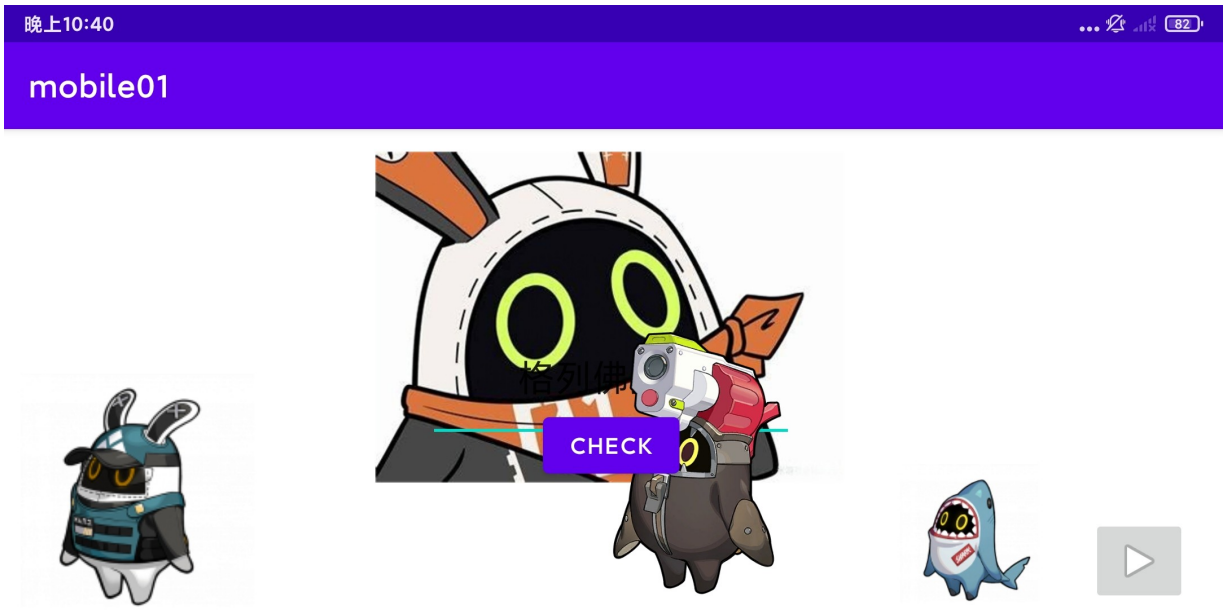
ISCC 练武初赛re+mobile wp-先知社区

返回文档

mobile

ISCC mobile 邦布出击

安装apk



点击右下角的按钮，进入图鉴界面，百度各种邦布的种类，一个一个试，可以得到三段base64加密的文本 [邦布图鉴 - 绝区零WIKI_BWIKI_哔哩](#)

名字： 鲨牙布

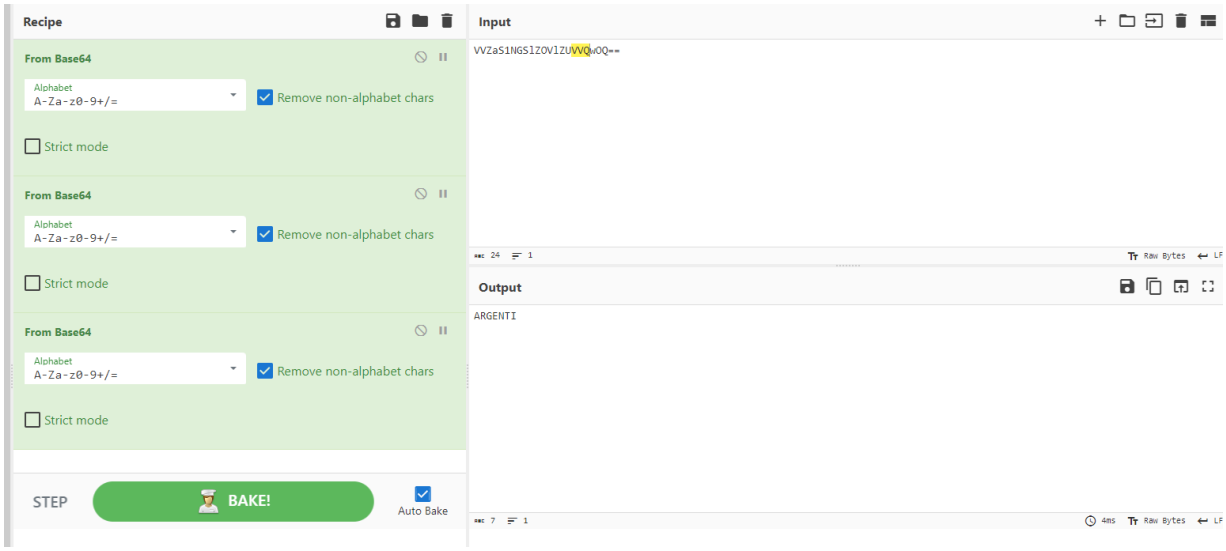
级别： S

SUBMIT

VVZaS1NGS

[哔哩]

然后将三段base64拼接起来，循环解码三次base64



得到一串明文 尝试打开解压得到的db文件，提示非数据库文件，经查询是经过sqlcipher加密，那么此前得到的明文应该就是解密的key

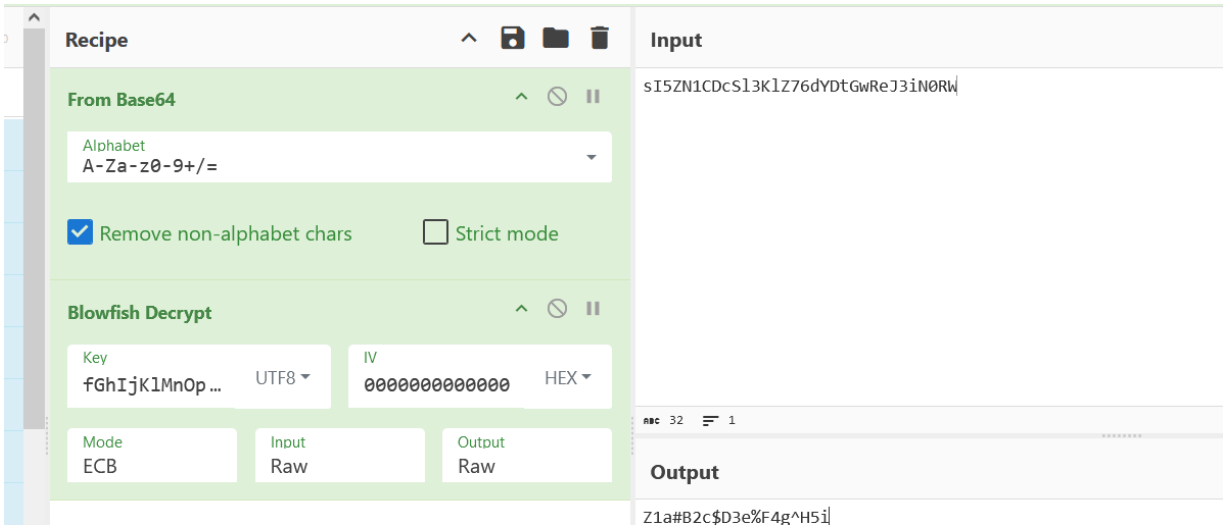
	id	name	value	info
1	1	flag!	102;108;97;103;123;121;111;11...	Congratulationo.0◆◆&#◆◆
2	2	key?	CdEfGhIjKlMnOpQr	!blowfish!
3	3	():flag?KEY	\u0074\u0068\u0065\u0020\u...	something crucial

flag是假的，实际应该留意的是key以及info中的blowfish（一种加密方式） 使用jadx打开apk

```
public boolean Jformat(String str) {
    if (str.length() < 7 || !str.substring(0, 5).equals("ISCC{") || str.charAt(str.length() - 1) != '}') {
        return false;
    }
    try {
        String a = a.a();
        Log.d("str1", "des加密明文: " + a);
        try {
            String encrypt = new DESHelper().encrypt(a, "WhItenet", getiv());
            Log.d("DEBUG_RES", "加密结果 res: " + encrypt);
            return str.substring(5, str.length() - 1).equals(encrypt);
        } catch (Exception e) {
            throw new RuntimeException(e);
        }
    } catch (Exception e2) {
        throw new RuntimeException(e2);
    }
}
```

```
public class b {
    private static String hiddenString = "sI5ZN1CDcSl3KlZ76dYDtGwReJ3iN0RW";

    public static String b() {
        try {
```

将上图中的密文通过blowfish解密之后得到的内容就是DES的明文

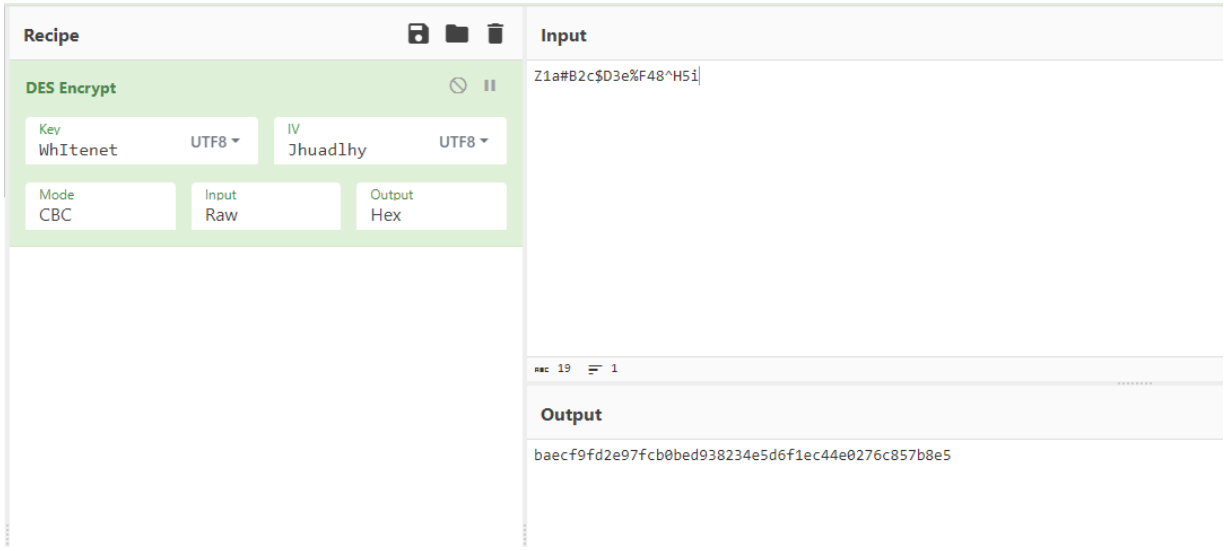


根据apk的逻辑，只有当该明文DES加密的结果和输入内容去掉flag格式后的内容相同才正确 已知明文、key、加密方式，那么对于DES加密，还需要具备的就是iv，但是iv是通过native函数生成的

```
public native String getiv();
```

方法一：分析so文件iv的生成逻辑 -- 生成逻辑比较复杂，放弃 方法二：hook native function，在调用getiv时输出iv 这里使用frida hook（要在手机上先运行frida-server）

```
[*] MainActivity.getiv() called, returned: Jhuadlhykvdutfpssbzpvu
Jhuadlhykvdutfpssbzpvu
```



ISCC mobile detective

附件是一个apk文件，用jadx打开

```
ride // androidx.fragment.app.FragmentActivity, androidx.activity.ComponentActivity, androidx.core.app.ComponentActivity, android.app.Activity
cted void onCreate(Bundle bundle) {
uper.onCreate(bundle);
ctivityMainBinding inflate = ActivityMainBinding.inflate(getLayoutInflater());
his.binding = inflate;
etContentView(inflate.getRoot());
his.flagEditText = this.binding.editTextFlag;
utton button = this.binding.button;
his.submitButton = button;
utton.setOnClickListener(new View.OnClickListener() { // from class: com.example.detective.MainActivity.1
    @Override // android.view.View.OnClickListener
    public void onClick(View view) {
        if (MainActivity.this.Jformat(MainActivity.this.flagEditText.getText().toString())) {
            Toast.makeText(MainActivity.this, "Congratulations, you are right!", 1).show();
        } else {
            Toast.makeText(MainActivity.this, "PITY", 0).show();
        }
    }
});


DX INFO: Access modifiers changed from: private */
c boolean Jformat(String str) {
eturn str.length() >= 8 && (str.length() + 1) % 2 != 0 && str.substring(0, 5).equals("ISCC{") && str.charAt(str.length() - 1) == '}' && stringFromJNI(
```

可以看到关键是这个stringFromJNI函数，跟进之后发现是native函数，因此用IDA打开so文件

```
  intext2 = (char *)v22 + 1;
  LOBYTE(v22[0]) = 2 * len;
  if ( len )
LABEL_5:
    memmove(intext2, intext, len2);
  intext2[len2] = 0;
  (*a1)->ReleaseStringUTFChars(a1, (jstring)input, intext);
  v20 = 16;
  strcpy(v21, "Sherlock");
  xorEncrypt((__int64)v22, &v20, (__int64)&v17);
  v15[0] = 0LL;
  v15[1] = 0LL;
  if ( (v17 & 1) != 0 )
    v10 = v19;
  else
    v10 = v18;
  v16 = 0LL;
```

关键是这个xorEncrypt函数

```
}
v9 = *a2;
v10 = v3 >> 1;
if ( (v9 & 1) != 0 )
  v11 = *((_QWORD *)a2 + 1);
else
  v11 = v9 >> 1;
if ( !(_DWORD)v8 )
  v10 = *((_QWORD *)v5 + 1);
if ( v10 )
{
  v12 = 0LL;
  do
  {
    v13 = (unsigned __int8 *)*((_QWORD *)v5 + 2);
    v14 = (v8 & 1) == 0;
    v15 = (unsigned __int8 *)*((_QWORD *)a2 + 2);
    if ( !v14 )
      v13 = v5 + 1;
    if ( (*a2 & 1) == 0 )
      v15 = a2 + 1;
    v16 = v15[v12 % v11];
    v17 = v13[v12];
    if ( (*(_BYTE *)a3 & 1) != 0 )
      v18 = *(_QWORD *)(a3 + 16);
    else
      v18 = a3 + 1;
    *(_BYTE *)(v18 + v12++) = v16 ^ v17;
    v8 = *v5;
    v19 = *((_QWORD *)v5 + 1);
    v20 = v8 >> 1;
    LOBYTE(v8) = (v8 & 1) == 0;
    if ( (_BYTE)v8 )
      v19 = v20;
  }
  while ( v12 < v19 );
}
return result;
}
```

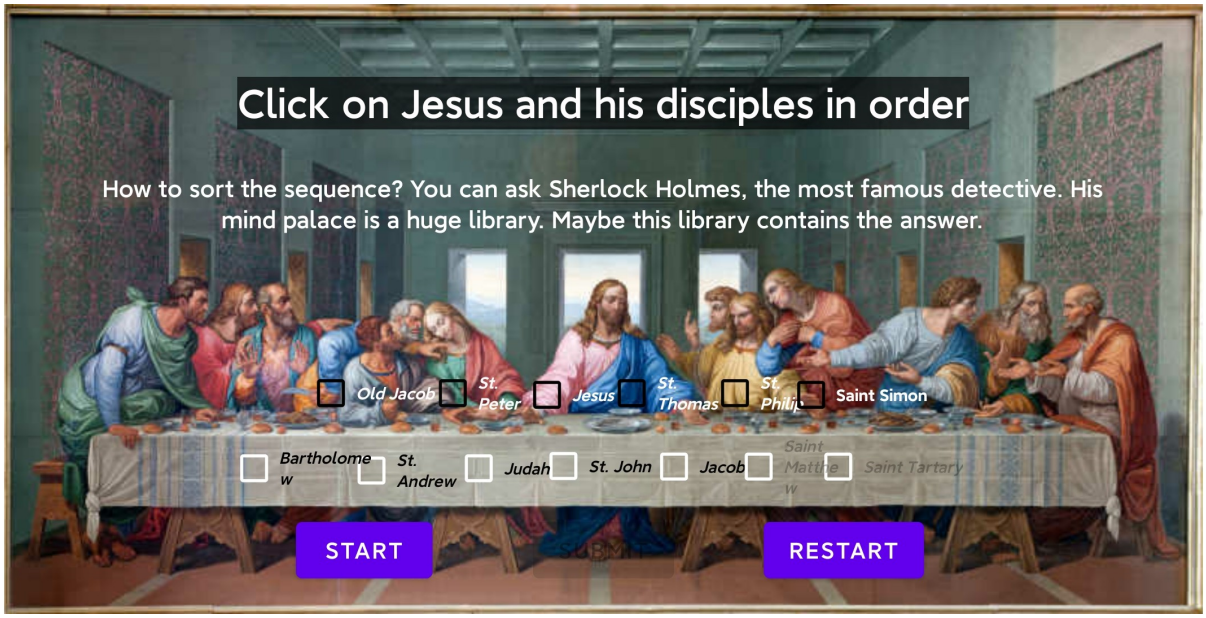
```
__int64 __usercall xorEncrypt@<X0>(__int64 result@<X0>, unsigned __int8 *a2@<X1>, __int64 a3@
{
  unsigned __int8 v3; // w9
  unsigned __int8 *v5; // x20
  __int64 v7; // x8
  unsigned __int64 v8; // x11
  unsigned __int64 v9; // x8
  __int64 v10; // x9
  unsigned __int64 v11; // x8
  unsigned __int64 v12; // x9
  unsigned __int8 *v13; // x15
  bool v14; // zf
  unsigned __int8 *v15; // x11
  unsigned __int8 v16; // w11
  unsigned __int8 v17; // w14
  __int64 v18; // x15
  unsigned __int64 v19; // x14
  unsigned __int64 v20; // x15

  v3 = *(_BYTE *)result;
  v5 = (unsigned __int8 *)result;
  if ( (*(_BYTE *)result & 1) != 0 )
  {
    result = sub_62B70(a3, *(_QWORD *)(result + 16), *(_QWORD *)(result + 8));
    v3 = *v5;
    LODWORD(v8) = (*v5 & 1) == 0;
  }
  else
  {
    v7 = *(_QWORD *)(result + 16);
    LODWORD(v8) = 1;
    *(_QWORD *)a3 = *(_QWORD *)result;
    *(_QWORD *)(a3 + 16) = v7;
  }
  v9 = *a2;
  v10 = v3 >> 1;
  if ( (v9 & 1) != 0 )
    v11 = *((_QWORD *)a2 + 1);
  else
```

00062754 _Z10xorEncryptRKNSt6__ndk112basic_stringIcNS_11char_traitsIcEENS_9allocatorIcEEEES7_:60 (62754)

通过分析代码可知，该函数先将字符串转换为十六进制，再将输入与key异或之后转为字符串，然后从每4个字符中提取前2个字符，然后再根据一定规律打乱字符串的位置信息，最后替换特定位置的字符

HolyGrail

附件为apk安装包



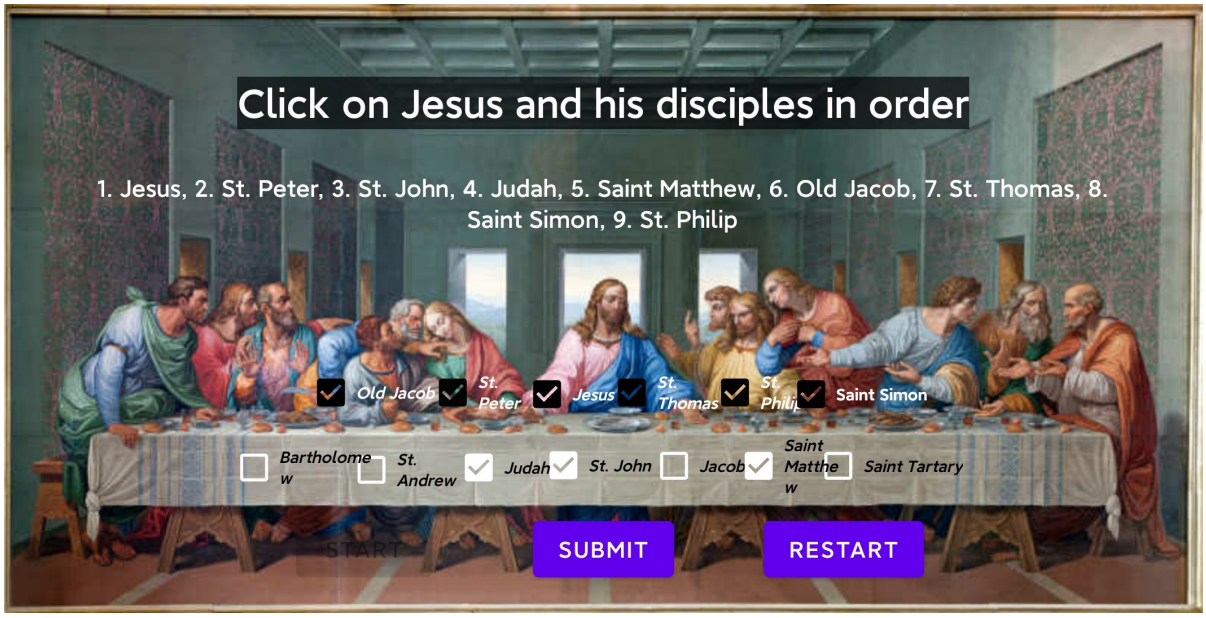
使用jadx打开apk，发现其中有许多checkbox，点击checkbox的响应如下

```
    /* JADX INFO: Access modifiers changed from: private */
    /* renamed from: onCheckBoxClicked, reason: merged with bridge method [inline-methods] */
119 public void m51lambda$enableCheckBoxes$3$comexampleholygrailMainActivity(CheckBox checkBox) {
120     String resourceEntryName = getResources().getResourceEntryName(checkBox.getId());
121     if (checkBox.isChecked()) {
123         this.userSequence.add(resourceEntryName);
        } else {
126         this.userSequence.remove(resourceEntryName);
        }
129     updateSelectedOrderTextView();
    }
```

每点击一个checkbox就会在userSequence末尾添加当前checkbox的资源名称

```
private void submitSequence() {
    CipherDataHandler.saveCipherText(this, CipherDataHandler.getCipherText(this.userSequence));
    GameData.userSequence.clear();
    GameData.userSequence.addAll(this.userSequence);
    goToNextPage();
}
```

而根据app的提示，需要按照特定顺序点击checkbox，才能进入验证flag的页面，并且返回在native层加密后的密文 关于顺序，可以自行百度，也可以问ai，最终顺序如下



如何获得密文：通过frida hook，手动传入特定顺序的参数（每个checkbox的参数也需要通过frida hook得到），然后输出返回的密文

然后分析验证flag的页面

```
private void submitSequence() {
    String trim = this.flagInput.getText().toString().trim();
    if (!isCorrectFormat(trim)) {
        Toast.makeText(this, "Wrong flag format", 0).show();
        return;
    }
    String substring = trim.substring(5, trim.length() - 1);
    String string = this.sharedPreferences.getString("cipherText", "");
    String validateFlag = a.validateFlag(this, substring);
    if (validateFlag != null && validateFlag.equals(string)) {
        if ("af5c66387436b0c8cfa537be5751c4629c9e288966315c41ec07bf91658a32f4".equalsIgnoreCase(sha256(substring))) {
            Toast.makeText(this, "Success", 0).show();
            return;
        } else {
            Toast.makeText(this, "Correctly matched but in the wrong order.", 0).show();
            return;
        }
    }
    Toast.makeText(this, "Wrong flag", 0).show();
}

private boolean isCorrectFormat(String str) {
    return str.startsWith("ISCC{") && str.endsWith("}");
}

private String sha256(String str) {
    try {
        byte[] digest = MessageDigest.getInstance("SHA-256").digest(str.getBytes("UTF-8"));
        StringBuilder sb = new StringBuilder();
        for (byte b : digest) {
            String hexString = Integer.toHexString(b & UByte.MAX_VALUE);
            if (hexString.length() == 1) {
                sb.append('0');
            }
            sb.append(hexString);
        }
        return sb.toString();
    } catch (Exception unused) {
        return "";
    }
}
```

首先检查flag格式，然后调用a类的validateFlag方法

```
public class a {
    public static native String processWithNative(String str);

    static {
        System.loadLibrary("holygrail");
    }

    public static String validateFlag(Context context, String str) {
        return b.a(processWithNative(b(context, str)));
    }

    private static String b(Context context, String str) {
        return vigenereEncrypt(str, getEncryptionKey(context));
    }

    private static String getEncryptionKey(Context context) {
        String str = "";
        try {
            BufferedReader bufferedReader = new BufferedReader(new InputStreamReader(context.getResources().openRawResource(R.raw.key)));
            str = bufferedReader.readLine();
            bufferedReader.close();
            return str;
        } catch (Exception e) {
            e.printStackTrace();
            return str;
        }
    }

    private static String vigenereEncrypt(String str, String str2) {
        StringBuilder sb = new StringBuilder();
        int length = str2.length();
        int i = 0;
        for (int i2 = 0; i2 < str.length(); i2++) {
            char charAt = str.charAt(i2);
            if (Character.isLetter(charAt)) {
                char c = Character.isLowerCase(charAt) ? 'a' : 'A';
                char charAt2 = str2.charAt(i % length);
                charAt = (char) ((((charAt - c) + (charAt2 - (Character.isUpperCase(charAt2) ? 'A' : 'a'))) % 26) + c);
                i++;
            }
            sb.append(charAt);
        }
        return sb.toString();
    }
}
```

大概流程

●getEncryptionKey

●vigenereEncrypt

●processWithNative

● b.a

```
public class b {
    public static String a(String str) {
        StringBuilder sb = new StringBuilder();
        if (str.length() % 2 != 0) {
            Log.e("b", "Invalid input: Length must be even");
            return "Invalid input: Length must be even";
        }
        int i = 0;
        while (i < str.length()) {
            int i2 = i + 2;
            sb.append((char) Integer.parseInt(str.substring(i, i2), 16));
            i = i2;
        }
        return sb.toString();
    }
}
```

由于processWithNative是JNI函数，因此尝试frida hook该函数，尝试传入不同的值，发现每个字符对应的加密结果和顺序无关，因此可以直接生成所有字符加密的结果，再对目标字符串进行匹配

解密思路

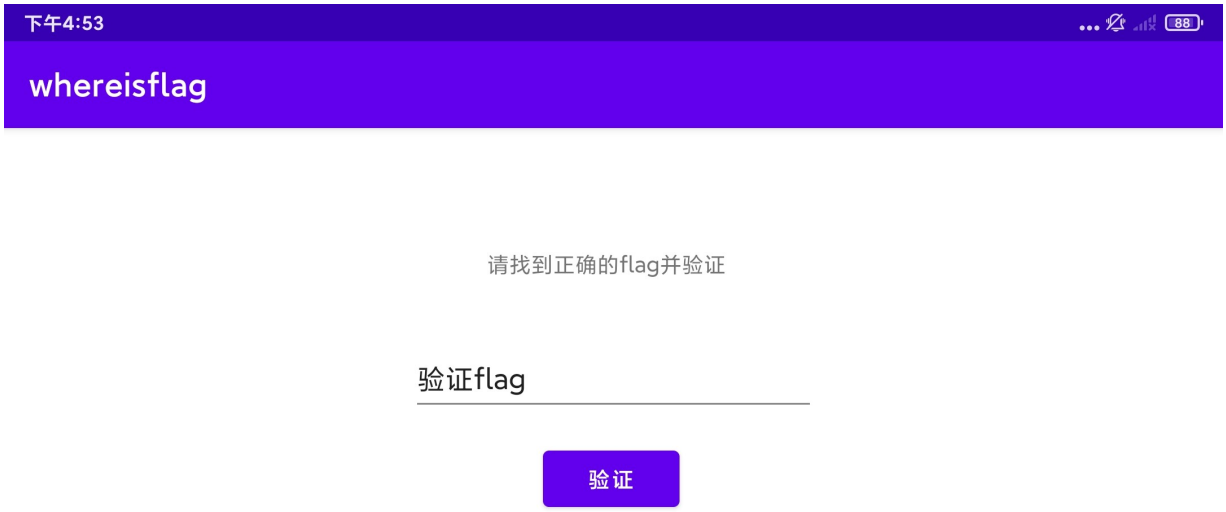
●转十六进制

●字符替换

●字符偏移

exp

whereisflag



jadx打开apk可以看到具体逻辑

```
public void onClick(View view) {
    String obj = MainActivity.this.flagEditText.getText().toString();
    if (obj.length() != 16) {
        Toast.makeText(MainActivity.this, "flag长度错误，请继续寻找", 0).show();
    } else if (new a().b(obj)) {
        Toast.makeText(MainActivity.this, "恭喜你找到了正确的flag", 1).show();
    } else {
        Toast.makeText(MainActivity.this, "flag错误，请继续寻找", 0).show();
    }
}
```

分析之后发现核心函数是native函数 Native 函数基本介绍

● 定义：Native 函数通过 native 关键字在 Java 中声明，实际代码编译在 .so 动态库（ELF 格式）中。

● JNI 桥梁：Java 层通过 JNI（Java Native Interface）调用 Native 函数，函数名和参数需遵循 JNI 规范。

```java
public class a {
    public native String compute(String str);

    public boolean b(String str) {
        if (str.startsWith("ISCC{") && str.endsWith("}")) {
            return compute(str.substring(5, 15)).equals("iB3A7kSISR");
        }
        return false;
    }
}
```

用解压软件直接解压apk文件，然后进入\lib\arm64-v8a目录找到so文件，使用IDA64打开so文件，在其中找到Java_开头的函数便是native导出函数 在加密函数中首先将输入倒序

```
if ( v11 )
{
  v13 = &v12[v11 - 1];
  if ( v13 > v12 )
  {
    v14 = v12 + 1;
    do
    {
      v15 = *(v14 - 1);
      *(v14 - 1) = *v13;
      *v13-- = v15;
    }
    while ( v14++ < v13 );
    v7 = *((_QWORD *)&v20 + 1);
    v8 = (unsigned __int8 *)v21;
    v9 = (unsigned __int64)(unsigned __int8)v20 >> 1;
    v10 = v20 & 1;
  }
}
```

然后根据字符表查找输入的字符

```
  *(_QWORD *)v21 = v16 + 2;
  goto LABEL_5;
}
v15 = (char *)v21 + 1;
LOBYTE(v21[0]) = 2 * v6;
if ( v6 )
LABEL_5:
  memmove(v15, v5, (size_t)v14);
*(( BYTE *)v14 + ( QWORD)v15) = 0;
encrypt((int)v21, v7, v8, v9, v10, v11, v12, v13, v20, v21[0], v22);
if ( (v21[0] & 1) != 0 )
  operator delete(v23);
(*(void (__fastcall **)(__int64, __int64, const char *))(*(_QWORD *)a1 + 1360LL))(a1, a3, v5);
if ( (v24 & 1) != 0 )
  v17 = v26;
else
```

字符表需要动态调试得到

```
__int64 __fastcall charToIndex(unsigned int a1)
{
  __int64 v1; // x0

  v1 = std::string::find(&xmmword_52DF0, a1, 0LL);
  if ( v1 == -1 )
    return 0xFFFFFFFFLL;
  else
    return (unsigned int)(v1 + 1);
}
```

```
__int64 __fastcall indexToChar(int a1)
{
  unsigned __int64 v1; // x9
  char *v3; // x8

  if ( a1 < 1 )
    return 0LL;
  v1 = *((_QWORD *)&xmmword_52DF0 + 1);
  if ( (xmmword_52DF0 & 1) == 0 )
    v1 = (unsigned __int64)(unsigned __int8)xmmword_52DF0 >> 1;
  if ( v1 < (unsigned int)a1 )
    return 0LL;
  if ( (xmmword_52DF0 & 1) != 0 )
    v3 = (char *)qword_52E00;
  else
    v3 = (char *)&xmmword_52DF0 + 1;
  return (unsigned __int8)v3[a1 - 1];
}
```

而根据encrypt、charToIndex、indexToChar函数的逻辑，可以看到在索引转换时有固定偏移，为2 从jadx反编译的结果得到目标密文iB3A7kSlSR，解密

exp

RE

打出flag

从可执行程序的图标判断为pyinstaller编译的程序，使用pyinstxtractor反编译

然后打开反编译的文件夹，打开同名pyc文件，反编译（uncompyle6或者在线） [python反编译 - 在线工具]

可以将decompress之后的内容写入文件（以下为部分）

叫AI写个脚本去混淆

有趣的小游戏

附件是一个exe和两个txt，其中txt内容为非打印字符 main函数中定义了许多常量

```
int __cdecl main(int argc, const char **argv, const char **envp)
{
  _QWORD *v3; // rax
  _QWORD *v4; // rax
  _QWORD *v5; // rax
  __int64 v7[3]; // [rsp+20h] [rbp-60h] BYREF
  char v8; // [rsp+3Fh] [rbp-41h] BYREF
  int v9[24]; // [rsp+40h] [rbp-40h] BYREF
  char v10[32]; // [rsp+A0h] [rbp+20h] BYREF
  int v11[31]; // [rsp+C0h] [rbp+40h] BYREF
  char v12; // [rsp+13Fh] [rbp+BFh] BYREF
  char v13[48]; // [rsp+140h] [rbp+C0h] BYREF

  sub_40B270(argc, argv, envp);
  v11[0] = 0x18A550A;
  v11[1] = 0x840630DB;
  v11[2] = 0x3EC0C129;
  v11[3] = 0x175BDB99;
  v11[4] = 0x7FD5E3DB;
  v11[5] = 0xF99F6912;
  v11[6] = 0x199B32C1;
  v11[7] = 0x836C22BB;
  v11[8] = 0x440E4880;
  v11[9] = 0xE4EC8310;
  v11[10] = 0x2F00227A;
  v11[11] = 0xAB294A2A;
  v11[12] = 0x8EDB89F1;
```

通过查看附近函数，发现其他地方也定义了常数

```
  sub_48EC30(a1);
  *(_DWORD *)(a1 + 48) = 1000;
  *(_DWORD *)(a1 + 52) = 0;
  sub_48F820(a1 + 56);
  *(_DWORD *)(a1 + 80) = 0x12345678;
  *(_DWORD *)(a1 + 84) = 0x9ABCDEF0;
  *(_DWORD *)(a1 + 88) = 0xFEDCBA98;
  *(_DWORD *)(a1 + 92) = 0x76543210;
  v2 = time64(0i64);
  srand(v2);
  qmemcpy(v32, "##########", sizeof(v32));
  nullsub_4(&v33);
  v20.m128i_i64[0] = (__int64)v32;
  v20.m128i_i64[1] = 10i64;
  sub_48F050(v21, &v20, &v33);
  qmemcpy(v34, "#   #    #", sizeof(v34));
  nullsub_4(&v35);
  v20.m128i_i64[0] = (__int64)v34;
  v20.m128i_i64[1] = 10i64;
  sub_48F050(&v22, &v20, &v35);
  qmemcpy(v36, "# # #### #", sizeof(v36));
  nullsub_4(&v37);
  v20.m128i_i64[0] = (__int64)v36;
  v20.m128i_i64[1] = 10i64;
```

查看字符串表，可以在其中找到两个txt的文件名，交叉引用查看

```
.data:00…   00000005   C
.data:00…   0000005F   C
.rdata:0…   0000000A   C   file1.txt
.rdata:0…   0000000A   C   file2.txt
.rdata:0…   00000006   C   pause
.rdata:0…   0000001E   C   terminate called recursively\n
.rdata:0…   00000031   C   terminate called after throwing an instance of '
.rdata:0…   0000002E   C   terminate called without an active exception\n
.rdata:0…   0000000C   C     what():
.rdata:0…   00000024   C   __gnu_cxx::__concurrence_lock_error
.rdata:0…   00000026   C   __gnu_cxx::__concurrence_unlock_error
.rdata:0…   00000015   C   basic_string::append
.rdata:0…   00000031   C   locale::_S_normalize_category category not found
.rdata:0…   00000020   C   locale::_Impl::_M_replace_facet
.rdata:0…   00000024   C   __gnu_cxx::__concurrence_lock_error
```

```
void __fastcall process(__int64 a1, int a2, __int64 a3)
{
  __int64 v3; // [rsp+20h] [rbp-20h]
  void (__fastcall *v4)(__int64, _QWORD, __int64); // [rsp+28h] [rbp-18h]
  __int64 v5; // [rsp+30h] [rbp-10h]
  void (__fastcall *lpAddress)(__int64, _QWORD, __int64); // [rsp+38h] [rbp-8h]

  if ( a2 <= 1 )
  {
    if ( a2 < -1 )
    {
      v4 = (void (__fastcall *)(__int64, _QWORD, __int64))sub_41C090("file2.txt");
      if ( v4 )
      {
        v3 = sub_48F610(a1);
        v4(v3, (unsigned int)a2, a3);
        VirtualFree(v4, 0i64, 0x8000u);
      }
    }
  }
  else
  {
    lpAddress = (void (__fastcall *)(__int64, _QWORD, __int64))sub_41C090("file1.txt");
    if ( lpAddress )
    {
      v5 = sub_48F610(a1);
      lpAddress(v5, (unsigned int)a2, a3);
```

其中process是我重命名的结果 可以看到其中比较奇怪的一点是程序将文件的内容作为函数执行，也就是说原本内容不可见的txt其实是函数的二进制数据，要想知道该函数的具体逻辑，需要动态调试，在此处下断点，触发断点之后在汇编步进就可以看到其中逻辑

```
debug032:0000000000190000 sub     rsp, 30h
debug032:0000000000190004 mov     [rsp+28h], r8
debug032:0000000000190009 mov     [rsp+24h], edx
debug032:000000000019000D mov     [rsp+18h], rcx
debug032:0000000000190012 xor     eax, eax
debug032:0000000000190014 sub     eax, [rsp+24h]
debug032:0000000000190018 mov     [rsp+24h], eax
debug032:000000000019001C mov     eax, 34h ; '4'
debug032:0000000000190021 cdq
debug032:0000000000190022 idiv    dword ptr [rsp+24h]
debug032:0000000000190026 add     eax, 6
debug032:0000000000190029 mov     [rsp+4], eax
debug032:000000000019002D imul    eax, [rsp+4], 9E3779B9h
debug032:0000000000190035 mov     [rsp+0Ch], eax
debug032:0000000000190039 mov     rax, [rsp+18h]
debug032:000000000019003E mov     eax, [rax]
debug032:0000000000190040 mov     [rsp+14h], eax
debug032:0000000000190044
debug032:0000000000190044 loc_190044:                         ; CODE XREF: debug032:000000000019017A↓j
debug032:0000000000190044 mov     eax, [rsp+0Ch]
```

可以将汇编扔给ai判断函数逻辑 deekseek：“这段汇编代码实现的是 XXTEA（eXtended TEA）算法的解密过程……” 于是知道了加解密逻辑，并且根据xxtea的密钥格式可以判断先前的两处常量中位数较短的是key，

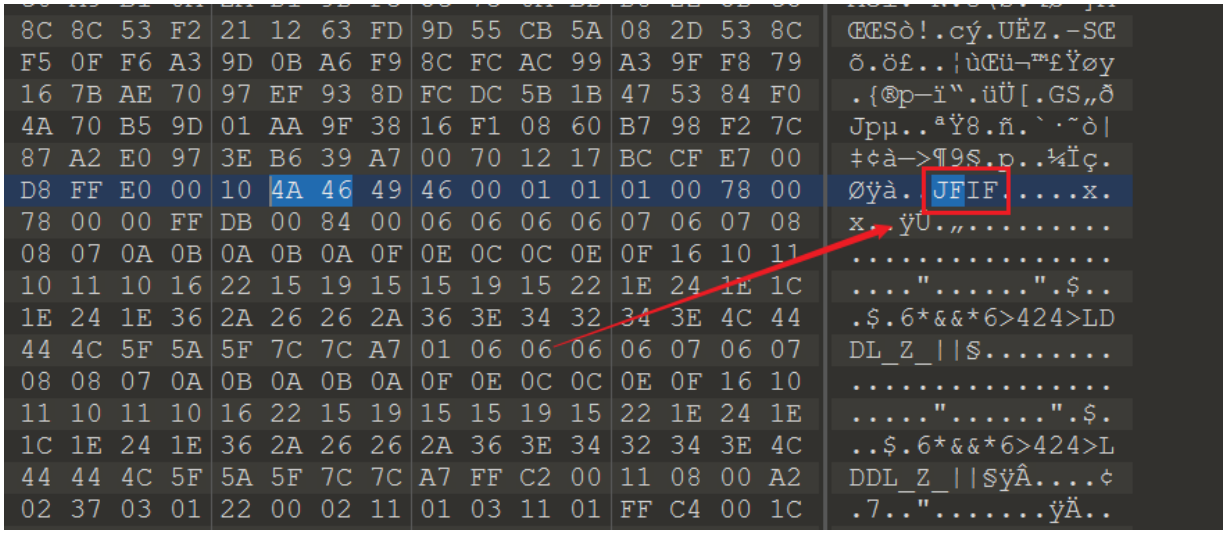
而位数较长的是密文 接下来有两种解题方式：

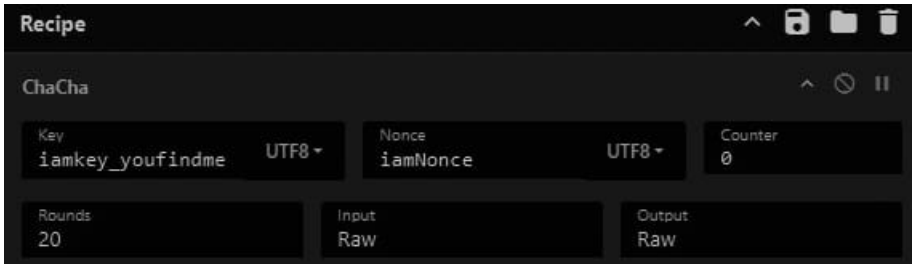
1手动分析解密逻辑，自己编写代码

2交给ai xxtea的加解密逻辑网上有很多就不细说了，直接给出解密脚本

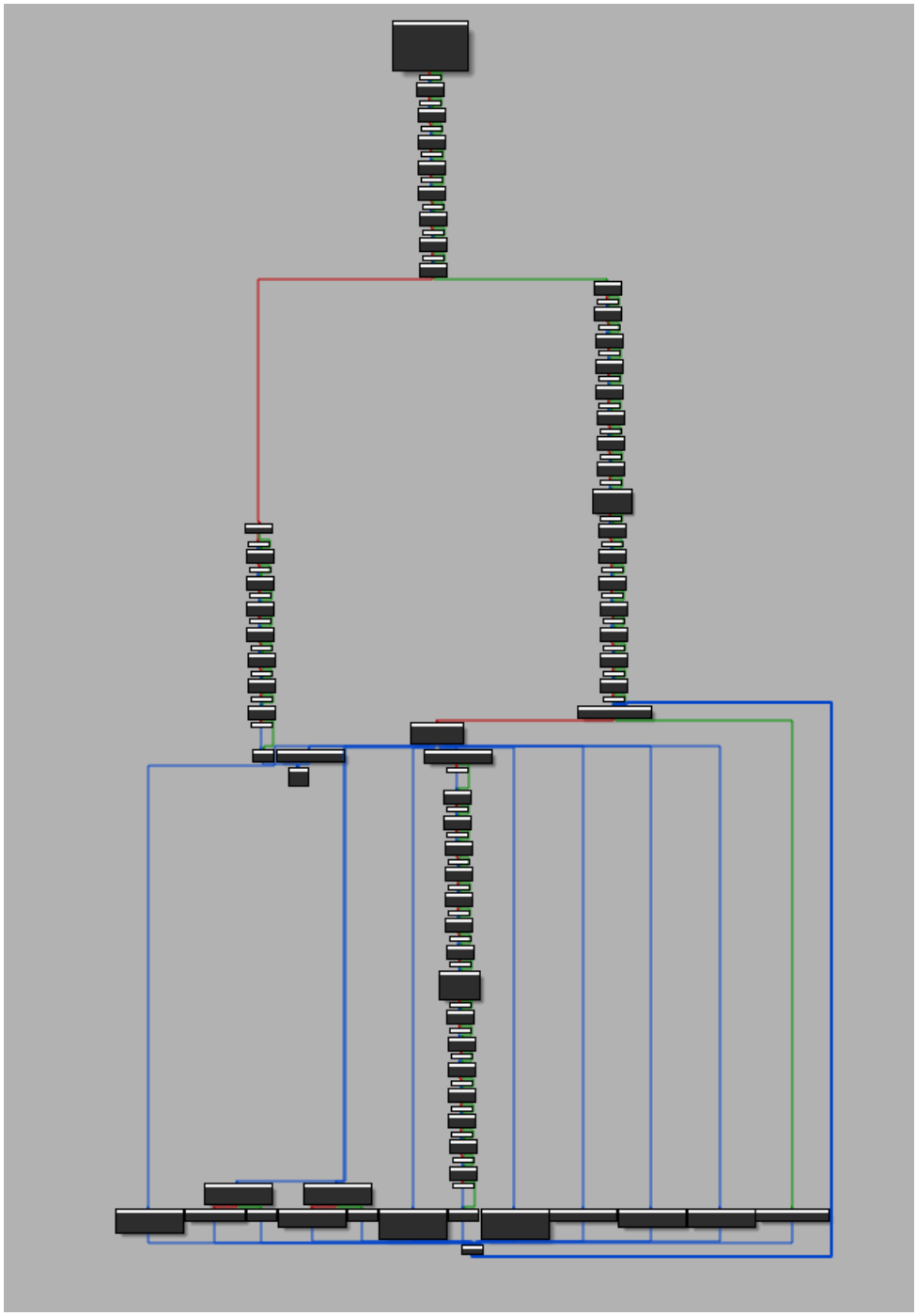
真？复杂

题目附件是一个raw文件，010editor查看发现JFIF文件头，提取图片





然后使用cyberchef解密，解密之前要先把原raw文件中附加的图片信息删除 解密之后得到压缩包一个，解密得exe文件和enc文件各一个



虽然流程图长这个样，但是是可以手动去除的

```
65      case 1:
66        if ( (unsigned int)sub_4015DE() )
67          sub_401550();
68        if ( (unsigned int)sub_401682() )
69          sub_40172A();
70        if ( (unsigned int)sub_40172A() )
71          sub_4015DE();
72        if ( (unsigned int)sub_4015DE() )
73          sub_401550();
74        if ( (unsigned int)sub_401550() )
75          sub_401682();
76        if ( (unsigned int)sub_40172A() )
77          sub_401550();
78        if ( (unsigned int)sub_401550() )
79          sub_40172A();
80        if ( (unsigned int)sub_401682() )
81          sub_40172A();
82        if ( (rand() & 1) != 0 )
83        {
84          if ( (rand() & 1) != 0 )
85          {
86            if ( (v4 & 1) != 0 )
87              v5 = 15;
88            else
89              v5 = 2;
90          }
91          else if ( (v4 & 1) != 0 )
92          {
93            v5 = 15;
94          }
95          else
96          {
97            v5 = 2;
98          }
99        }
100       else if ( (rand() & 1) != 0 )
101       {
102         if ( (v4 & 1) != 0 )
```

第一种方法：（直接忽略和输入无关的语句和函数，对于涉及到修改输入的语句统统下断点） 第二种方法：直接分析加密函数的switch逻辑，可以发现是对奇偶索引的字符做不同的变换，核心变量为v4（索引）和v5（控制跳转的case），通过v4&1的操作判断奇偶 通过分析exe文件可知原本逻辑是给定flag.txt，用exe加密得到enc文件，而现在只有enc文件，故需要逆向推解密逻辑 通过分析得到解密脚本

faze

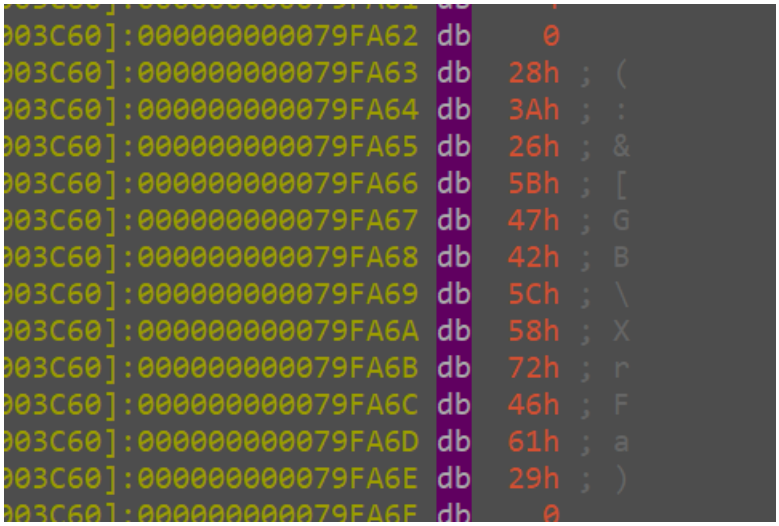
题目附件：faze.exe 使用IDA打开附件

```
text:0000000000401DBE loc_401DBE:                              ; CODE XREF: main+CD↑j
text:0000000000401DBE                 lea     rax, [rbp+390h+var_3B0]
text:0000000000401DC2                 mov     rcx, rax
text:0000000000401DC5                 call    _Z3xP9PA10_i    ; xP9(int (*)[10])
text:0000000000401DCA                 lea     rax, [rbp+390h+var_3BD]
text:0000000000401DCE                 mov     rcx, rax
text:0000000000401DD1                 call    _Z3mS2Pc        ; mS2(char *)
text:0000000000401DD6                 lea     rdx, [rbp+390h+var_3BD]
text:0000000000401DDA                 lea     rax, [rbp+390h+var_3D0]
text:0000000000401DDE                 mov     r8, rdx
text:0000000000401DE1                 lea     rdx, aIsccS     ; "ISCC{%s}"
text:0000000000401DE8                 mov     rcx, rax
text:0000000000401DEB                 call    _Z9sprintf_sILy19EEiRAT__cPKcz ; sprintf_s<19ull>(char (&)[19ull],char const*,...)
text:0000000000401DF0                 lea     rax, [rbp+390h+var_3F0]
text:0000000000401DF4                 mov     rcx, rax
text:0000000000401DF7                 call    _ZNSt7__cxx1112basic_stringIcSt11char_traitsIcESaIcEEC1Ev ; std::__cxx11::basic_string<char,std::cha
text:0000000000401DFC                 lea     rdx, aEnterFlag ; "Enter flag:"
text:0000000000401E03                 mov     rcx, cs:_refptr__ZSt4cout
text:0000000000401E0A ;   try {
text:0000000000401E0A                 call    _ZStlsISt11char_traitsIcEERSt13basic_ostreamIcT_ES5_PKc ; std::operator<<<std::char_traits<char>>(st
text:0000000000401E0F                 mov     rdx, cs:_refptr__ZSt4endlIcSt11char_traitsIcEERSt13basic_ostreamIT_T0_ES6_
text:0000000000401E16                 mov     rcx, rax
text:0000000000401E19                 call    _ZNSolsEPFRSoS_E ; std::ostream::operator<<(std::ostream & (*)(std::ostream &))
text:0000000000401E1E                 lea     rax, [rbp+390h+var_3F0]
text:0000000000401E22                 mov     rdx, rax
text:0000000000401E25                 mov     rcx, cs:_refptr__ZSt3cin
text:0000000000401E2C                 call    _ZSt7getlineIcSt11char_traitsIcESaIcEERSt13basic_istreamIT_T0_ES7_RNSt7__cxx1112basic_stringIS4_S5_T
text:0000000000401E31                 lea     rdx, [rbp+390h+var_3D0]
text:0000000000401E35                 lea     rax, [rbp+390h+var_3F0]
text:0000000000401E39                 mov     rcx, rax
text:0000000000401E3C                 call    _ZSteqIcSt11char_traitsIcESaIcEEbRKNSt7__cxx1112basic_stringIT_T0_T1_EEPKS5_ ; std::operator==<char>
text:0000000000401E41                 test    al, al
text:0000000000401E43                 jz      short loc_401E69
text:0000000000401E45                 lea     rdx, aCorrect   ; "Correct!"
text:0000000000401E4C                 mov     rcx, cs:_refptr__ZSt4cout
```

一眼C++，通过判断代码可以发现目标字符串在用户输入之前（getline）已经完成了目标字符串的初始化，所以这里有多种解法

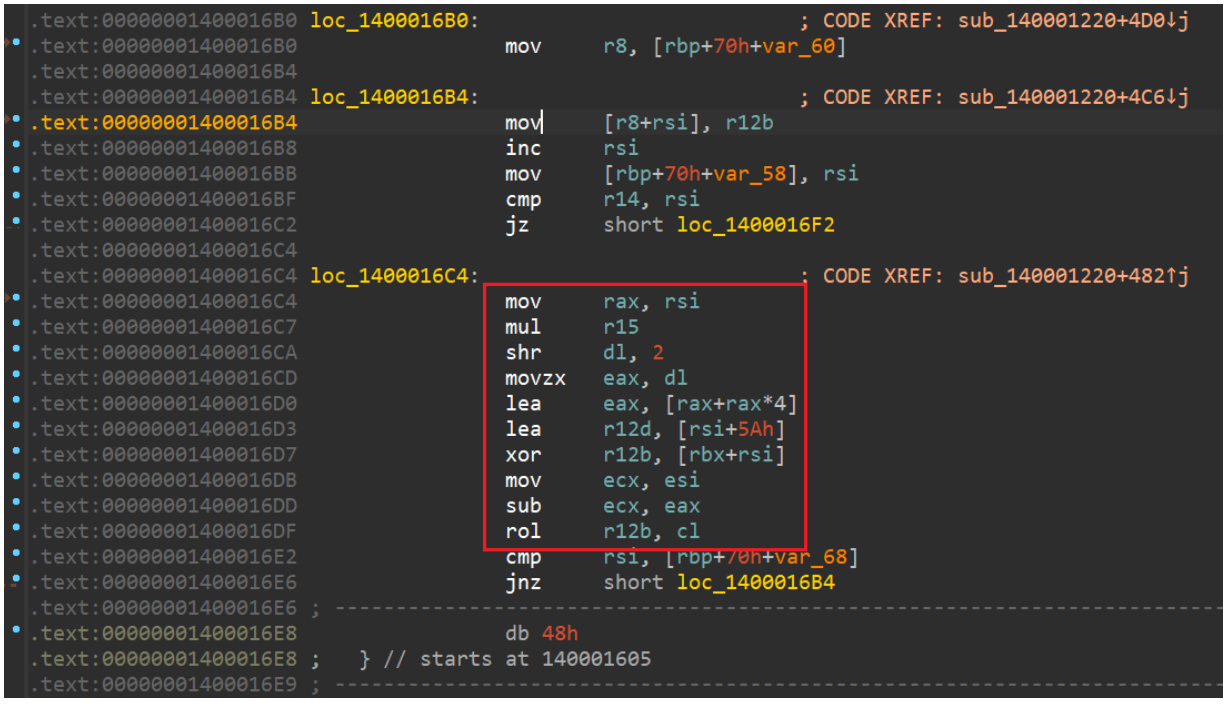
1在sprintf上下断点，直接查看写入目标字符串的内容

2 在比较的时候（operator==）下断点，查看比较的数据 这里选择前者，在程序暂停时跳转到rcx所在地址



greeting

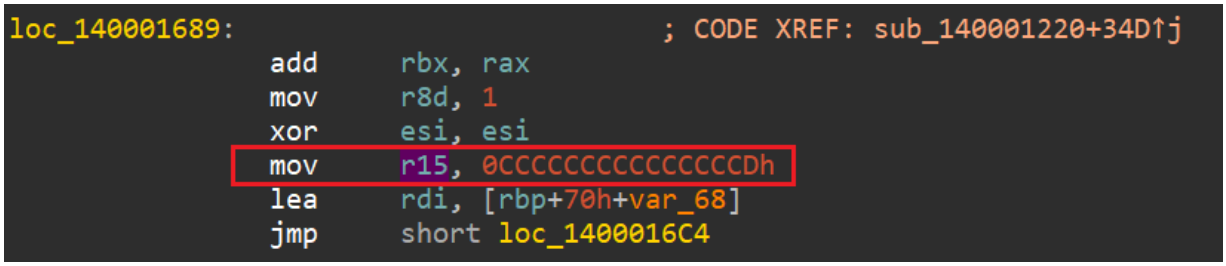
首先IDA打开可执行文件，会发现有些函数反编译的结果不正确，且提示错误，因此可以查看目标函数附近的汇编代码，找到类似加密逻辑的代码



明显的异或和循环左移操作，大概率是加密逻辑 通过分析可知，代码首先是计算一个偏移，然后将目标数据对应索引的字节在异或i+0x5a之后（esi为索引）循环左移该计算出来的偏移，因此目标可以分为两步：

1分析该偏移的计算方式

2 反推整个加密逻辑 这里的r15其实是一个固定的值



关于偏移量的计算

●通过手动分析

○ mul r15 和 shr dl, 2 的组合实际上执行的是整数除法 i / 5

○ lea eax, [rax+rax*4] 计算的是 (i/5)*5

○ sub ecx, eax 计算的是 i - (i/5)*5

○ 以上逻辑等价于i%5

●直接动态调试可以发现rol操作中cl的取值是0、1、2、3、4、0……，所以其实偏移的计算方式是索引对5取余

然后就是逆向整个加密逻辑，有了偏移的计算方式，解密的逻辑很好推，就是对每个字节先循环右移再异或(i+0x5a) 对于密文，通过交叉引用和人肉分析等方式最终可以找到位于0x014001B390

因此完整的解密脚本如下