Lab9 实验报告

1.flag

flag{y0u_ha4e_g0t_1t}
flag{f14g_fr03_D3bu9}

2.flag1 解题过程

首先用 jadx 查看 java 代码，结果代码反编译错误，请参考，这个反逆向有点狠，就看 smali 代码。

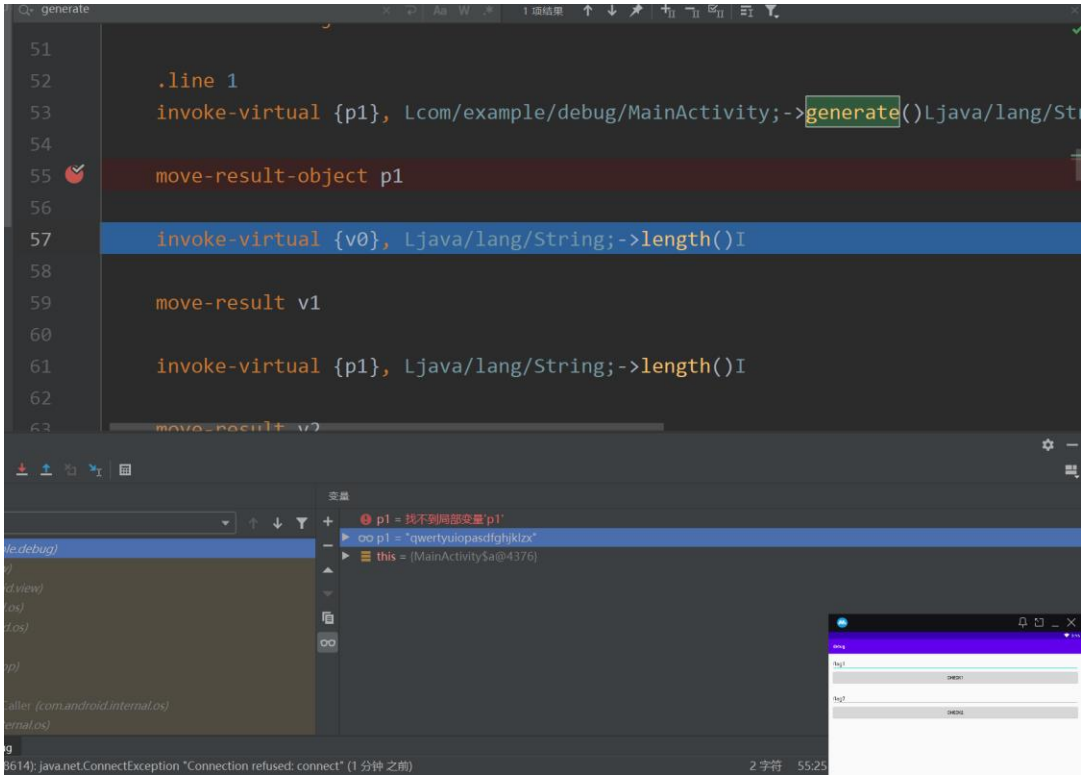
```
public void onCreate(android.os.Bundle r10) {
    /*
    // Method dump skipped, instructions count: 469
    */
    throw new UnsupportedOperationException("Method not decompiled: com.example.debug.MainActivity.onCreate(android.os.Bundle):void");
}
```

onCreate 函数会对程序的运行环境进行检测，判断有误就退出程序，尝试修改 smali 代码重打包后将这些检测去除，然后再进行下一步。

flag1 的代码同样反编译错误。

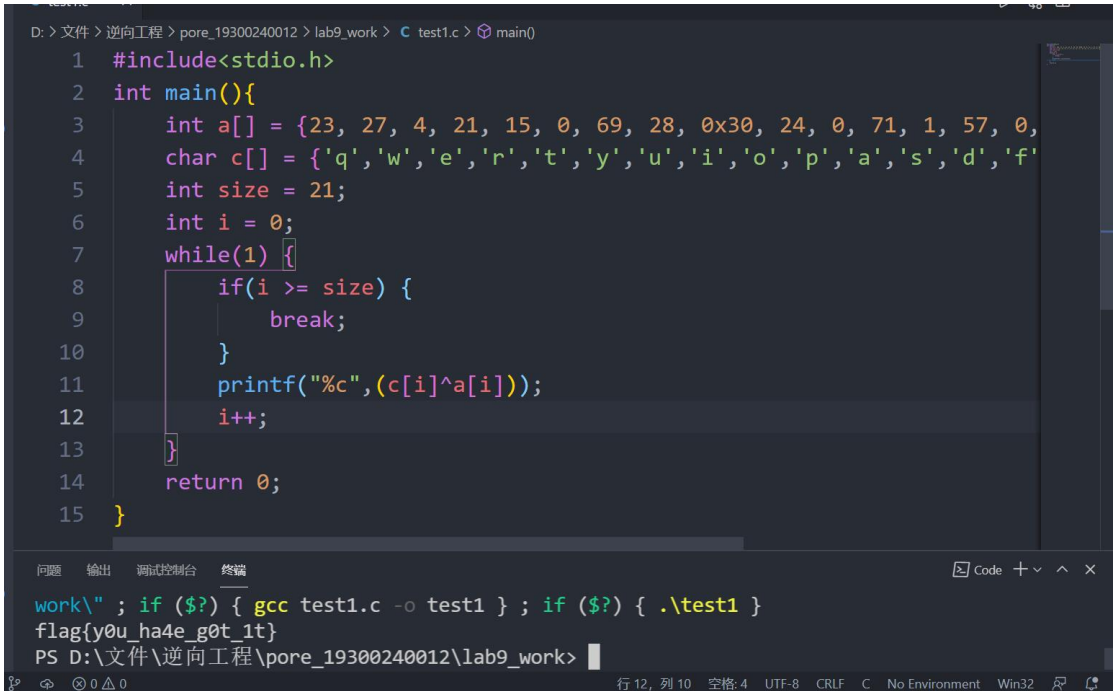
```
public class a implements View.OnClickListener {
    public a() {
    }

    /* JADX WARNING: Removed duplicated region for block: B:10:0x003f  */
    /* JADX WARNING: Removed duplicated region for block: B:11:0x0048  */
    /* Code decompiled incorrectly, please refer to instructions dump. */
    public void onClick(android.view.View r8) {
        /*
        // Method dump skipped, instructions count: 134
        */
        throw new UnsupportedOperationException("Method not decompiled: com.example.debug.MainActivity.a.onClick(android.view.View):void");
    }
}
```

通过阅读 smali 代码发现，flag1 主要是通过 native 方法获得一个字符串，让后利用异或加密对比输入的 flag，所以就可以使用 smali debug，在 generate 函数后设置断点，获得了对比字符串。



qwertyuiopasdfghjklzx

编写脚本，使用异或来输出字符串



获得 flag{y0u_ha4e_g0t_1t}

3.flag2 的解题过程

Flag2 的部分异常恶心，.so 中根本找不到 java 对应的 native 函数，后来发现这部分是动态加载，不遵循 jni 静态函数的命名规则。

通过阅读资料得知，jni 动态加载的方式，通过在 JNI_OnLoad 中设置断点获得动态加载的函数：

```
o:C21976D4 off_C21976D4 dd offset off_C21976D4     ; DATA XREF: sub_C216B550+10↑o
o:C21976D4                                         ; .data.rel.ro:off_C21976D4↓o
o:C21976D8 off_C21976D8 dq offset aFlag2           ; DATA XREF: JNI_OnLoad+114↑r
o:C21976E0 off_C21976E0 dq offset _Z15O0OOOOOO00OOP7_JNIEnvP8_jobjectP8_jstring
o:C21976E0                                         ; DATA XREF: JNI_OnLoad+106↑r
o:C21976E8 off_C21976E8 dq offset aZ_0             ; DATA XREF: JNI_OnLoad+F8↑r
o:C21976F0 public _ZTVSt9bad_alloc
o:C21976F0 ; `vtable for'std::bad_alloc
o:C21976F0 _ZTVSt9bad_alloc dd 0                   ; DATA XREF: .got:_ZTVSt9bad_alloc_ptr↓o
o:C21976F0                                         ; offset to this
o:C21976F0                                         ; offset to this
o:C21976F0                                         ; offset to this
o:C21976F4 dd offset _ZTISt9bad_alloc              ; `typeinfo for'std::bad_alloc
o:C21976F8 dd offset _ZNSt9exceptionD2Ev           ; std::exception::~exception()
o:C21976FC dd offset _ZNSt9bad_allocD0Ev           ; std::bad_alloc::~bad_alloc()
o:C2197700 dd offset _ZNKSt9bad_alloc4whatEv       ; std::bad_alloc::what(void)
o:C2197704 public _ZTVSt20bad_array_new_length
o:C2197704 ; `vtable for'std::bad_array_new_length
o:C2197704 _ZTVSt20bad_array_new_length_dd 0       ; DATA XREF: .got:_ZTVSt20bad_array_new_length_ptr↓o
```

_Z15O0OOOOOO00OOP7_JNIEnvP8

根据函数名去找到对应代码：

```
v43 = __readgsdword(0x14u);
result = 0;
if ( !(trick_time | (unsigned __int8)(trick_port | trick_ptraceid)) )
{
  v17 = -968232633;
  v16 = 712063750;
  v25 = 0;
  v27 = -585762285;
  v28 = -999170335;
  v29 = 533396744;
  v30 = 1915952719;
  v31 = -1772197344;
  v32 = -119;
  v33 = 1431590987;
  v34 = 1113867588;
  v35 = 1199192435;
  v36 = 930707052;
  v37 = 0;
  s = 1684234849;
  v39 = 1634166373;
  v40 = 1701077858;
  v41 = 842098534;
  v42 = 0;
  v26 = &v16;
  v24 = 8;
  memcpy(&dest, &unk_C214E3CB, 0x2Cu);
  v4 = 0;
  v22 = 0;
  do
  {
    *((_BYTE *)&v21 + v4) = v4;
    v4 = v22 + 1;
    v22 = v4;
  }
  while ( v4 != 256 );
  v23 = 0;
  v22 = 0;
  v5 = 0;
  v6 = 0;
  while ( 1 )
  {
    v23 = (unsigned __int8)(v5 + *((_BYTE *)v26 + v6 % v24) + *((_BYTE *)&v21 + v6));
```

这里的代码比较友善，如果通过判断，最后可以直接获得对应的 flag：

```
    }
    v13 = (const char *)l1111l1111111l1(&dest, (char *)&s);
    v14 = (const char *)jstringToChar(a1, a3);
    result = strcmp(v14, v13) == 0;
  }
  if ( __readgsdword(0x14u) != v43 )
    JUMPOUT(*(_DWORD *)algn_C212CEF4);
  return result;
}
```

这里有反逆向的检查，检查了运行的时间和端口的情况，在使用 ida 的情况下无法避免

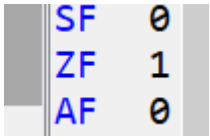
```
result = 0;
if ( !(trick_time | (unsigned __int8)(trick_port | trick_ptraceid)) )
{
  v17 = -968232633;
```

但是我们可以修改寄存器：

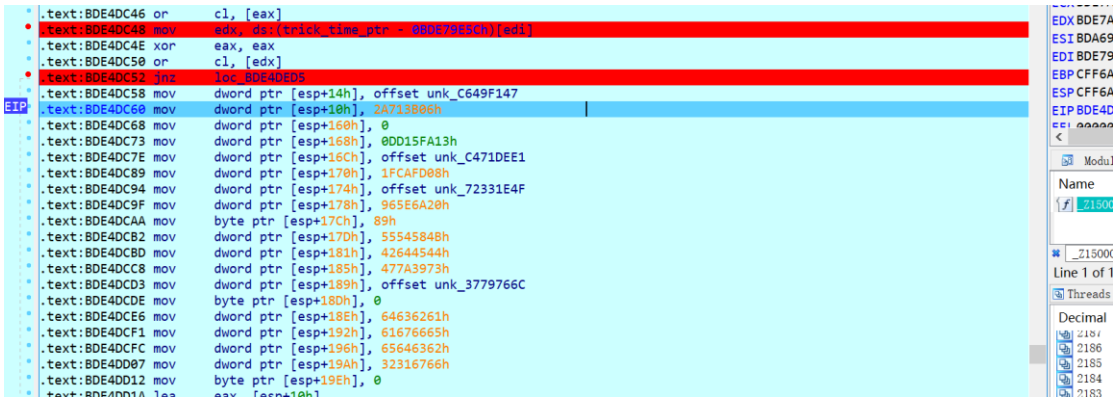
运行到 if 的语句：

```
.text:BDE4DC4E xor     eax, eax
.text:BDE4DC50 or      cl, [edx]
.text:BDE4DC52 jnz     loc_BDE4DED5
.text:BDE4DC58 mov     dword ptr [esp+14h], offset unk_C649F147
.text:BDE4DC60 mov     dword ptr [esp+10h], 2A713B06h
.text:BDE4DC68 mov     dword ptr [esp+160h], 0
.text:BDE4DC73 mov     dword ptr [esp+168h], 0DD15FA13h
.text:BDE4DC7E mov     dword ptr [esp+16Ch], offset unk_C471DEE1
.text:BDE4DC89 mov     dword ptr [esp+170h], 1FCAFD08h
.text:BDE4DC94 mov     dword ptr [esp+174h], offset unk_72331E4F
.text:BDE4DC9F mov     dword ptr [esp+178h], 965E6A20h
.text:BDE4DCAA mov     byte ptr [esp+17Ch], 89h
.text:BDE4DCB2 mov     dword ptr [esp+17Dh], 5554584Bh
```

将 ZF 寄存器修改为 1



成功转入 if 内部：



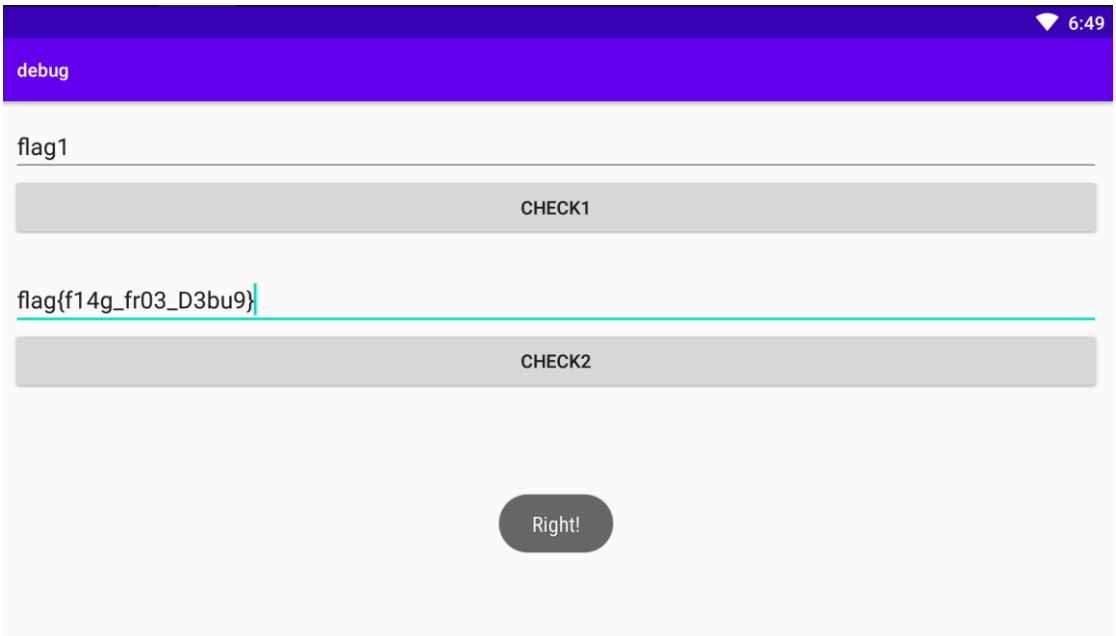
直接运行到 strcmp 函数前，进入内存察看：

```
debug285:B21BFCBD db    0
debug285:B21BFCBE db    0
debug285:B21BFCBF db    0
EAX debug285:B21BFCC0 db  66h ; f
debug285:B21BFCC1 db  6Ch ; l
debug285:B21BFCC2 db  61h ; a
debug285:B21BFCC3 db  67h ; g
debug285:B21BFCC4 db  7Bh ; {
debug285:B21BFCC5 db  66h ; f
debug285:B21BFCC6 db  31h ; 1
debug285:B21BFCC7 db  34h ; 4
debug285:B21BFCC8 db  67h ; g
debug285:B21BFCC9 db  5Fh ; _
debug285:B21BFCCA db  66h ; f
debug285:B21BFCCB db  72h ; r
debug285:B21BFCCC db  30h ; 0
debug285:B21BFCCD db  33h ; 3
debug285:B21BFCCE db  5Fh ; _
debug285:B21BFCCF db  44h ; D
debug285:B21BFCD0 db  33h ; 3
debug285:B21BFCD1 db  62h ; b
debug285:B21BFCD2 db  75h ; u
debug285:B21BFCD3 db  39h ; 9
debug285:B21BFCD4 db  7Dh ; }
debug285:B21BFCD5 db    0
debug285:B21BFCD6 db    0
debug285:B21BFCD7 db    0
```

获得了 flag2

flag{f14g_fr03_D3bu9}

关闭所有调试，重启程序：



4.反逆向技巧小结

（1）app onCreate 函数中增加对程序运行环境的判断，发现可能调试的环境，就直接退出。处理方式，直接修改 smali 代码，之后重打包。

（2）jadx 对 smali 反编译错误，不清楚是不是在 smali 代码的插入错误且不会执行的代码。处理方式，使用 jeb 反编译。

（3）在关键代码上新建线程，检查时间，如果时间过长，说明程序正在被单步调试。

（4）检查系统端口的状态，特殊端口被打开，说明程序很有可能被调试。处理方式，到具体退出逻辑前直接修改寄存器的值，避免退出

（5）native 代码使用动态加载，让重要逻辑代码不会被轻易发现。处理，在 JNI_OnLoad 中设置断点，去检查被加载的代码

（6）native 函数名混淆，放置大量垃圾代码。