

[\[TOC\]](#)

## 概述

本片文章描述一次完整的脱壳历程，从java层到Native层

## 流程概述

### Java层

1. java层找到库函数的入口位置
2. 过掉java层的反调试(解决方法在Native层：动态在isDebuggerConnected下断点)

### Native层

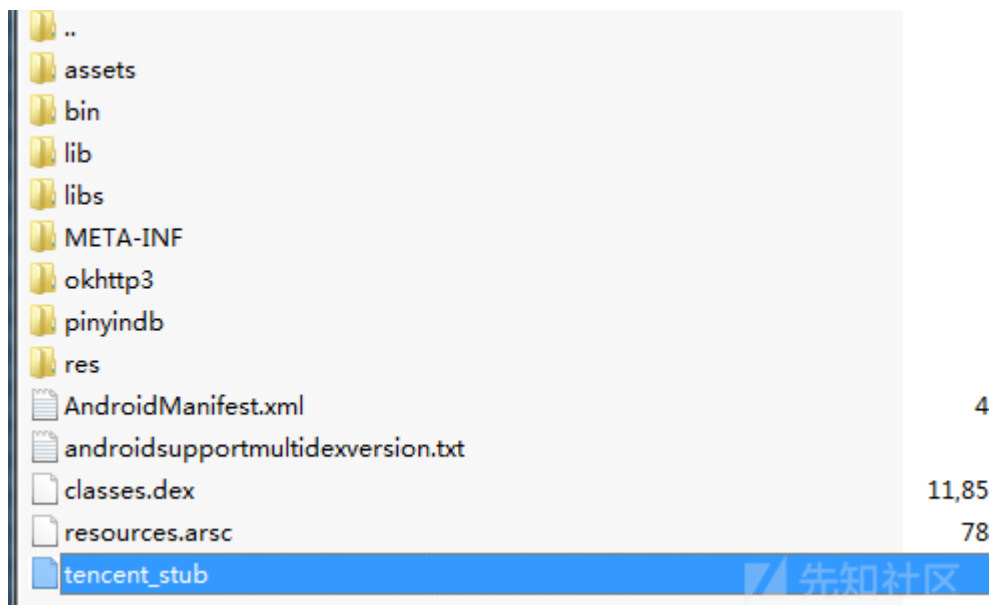
1. 绕过Anti IDA
2. 双层解密JNI\_OnLoad
3. 动态调试JNI\_OnLoad，得到注册的本地方法的具体位置
4. 分析load方法找到Dex动态解密的地方并dump

## 详细过程

这次脱壳用的测试机是Dalvik虚拟机4.4版本，所以底层用的libdvm.so库文件。

## 壳特征

有过壳经验的分析人员可以从安装包的特征文件和lib下的libshellxxx.so中看出是TX加固过的壳



### java层

#### 实锤加壳

在manifest中的入口类LoadingActivity是找不到的

```
<application android:theme="@style/AppTheme_Main" android:label="@string/app_name" android:icon="@mipmap/icon_launcher" android:allowBackup="false" android:fullBackupContent="false">
    <activity android:name="com.warmcar.nf.x.ui.activity.main.LoadingActivity" android:launchMode="singleTask">
        <intent-filter>
```

```

        <action android:name="android.intent.action.MAIN" />
        <category android:name="android.intent.category.LAUNCHER" />
    </intent-filter>
</activity>

```

## 初探attachBaseContext

既然入口类被隐藏了，我们根据调用关系找到启动入口类的地方，即Application这个类，我们主要需要关注的是attachBaseContext方法，这个在onCreate方法之前执行的

## 弃用jadx

这个方法首先调用e(context)进行了调试检查，接着在b(this)方法中进行了一些库地址的初始化操作

接着在 d(context)方法中加载不存在的库nfix、ufix，并且调用了本地方法fixNativeResource、fixUnityResource，从名称上看应该是修复操作

接下来主要是tx的SDK崩溃信息收集模块的功能，这块可以省略，主要看最后一个a((Context) this)方法，find Usage跳转过去发现调用了e()方法和load(f)方法

```

protected void attachBaseContext(Context context) {
    super.attachBaseContext(context);
    e(context);
    SystemClassLoaderInjector.fixAndroid(context, this);
    if (b(this)) {
        d(context);
        this.k = new Handler(getMainLooper());
        String str = "3.0.0.0";
        String str2 = "900015015";
        UserStrategy userStrategy = new UserStrategy(this);
        userStrategy.setAppVersion(str);
        CrashReport.setSdkExtraData(this, str2, str);
        CrashReport.initCrashReport(this, str2, false, userStrategy);
        new Thread(new d(this)).start();
        a((Context) this);
    }
}

private void d(Context context) {
    AssetManager assets = context.getAssets();
    String str = context.getApplicationInfo().sourceDir;
    try {
        System.loadLibrary("nfix");
        fixNativeResource(assets, str);
    } catch (Throwable th) {
    }
    try {
        System.loadLibrary("ufix");
        fixUnityResource(assets, str);
    } catch (Throwable th2) {
    }
}

public void a(Context context) {
    e();
    load(f);
}

```

而在jadx这里e方法并未生成相应伪代码，反汇编指令倒是没有错，为了方便分析，开启我们的jeb继续分析

```

/* JADX WARNING: Removed duplicated region for block: B:130:0x0584 */
/* JADX WARNING: Removed duplicated region for block: B:104:0x045b */
private void e() {
    /*
    r12 = this;
    r4 = r12.getApplicationInfo();
    r0 = r4.dataDir;
    r1 = new java.lang.StringBuilder;
    r1.<init>();
    r0 = r1.append(r0);
    r1 = "/tx_shell";
    r0 = r0.append(r1);
    r7 = r0.toString();
    r0 = r4.sourceDir;
    b = r0;
    r1 = android.os.Build.VERSION.SDK_INT;
    r0 = 0;
    r2 = 19;
    if (r1 >= r2) goto L_0x0151;
L_0x0024:
    r1 = 0;
    r2 = android.os.Build.VERSION.SDK_INT;
    r3 = 21;
    if (r2 >= r3) goto L_0x002d;
L_0x002b:

```

接盘侠：jeb探索首次加载so库

继续分析e();方法，根据反编译后的伪代码，可以看到这里第一次进行了so库的加载，加载shell

```

int v3 = 1;
if(v1_2.toLowerCase(Locale.US).contains("x86")) {
    v3 = 0;
}
else if(Build$VERSION.SDK_INT >= 21) {
    String[] v5 = Build.SUPPORTED_ABIS;
    if(v5 != null) {
        int v2;
        for(v2 = 0; v2 < v5.length; ++v2) {
            if(v5[v2].toLowerCase(Locale.US).contains("x86")) {
                v3 = 0;
            }
        }
    }
}
}

```

没有x86目录所以v3恒为1

```

v2 = 0;
if(v1_2.toLowerCase(Locale.US).contains("mips")) {
    v2 = 1;
}

```

```

String v4_1 = Build$VERSION.SDK_INT > 8 ? v4.nativeLibraryDir : "/data/data/" + TxAppEi
String v8 = "";
String v9 = "";
String v5_1 = "";

```

```

if(v3 != 0) { // v3恒为1
    sheila3000_so = TxAppEntry.str_sheila() + "-" + this.str_3_0_0_0();
}

```

```

else {
    sheila3000_so = "shellx-" + this.str_3_0_0_0();
    v5_1 = TxAppEntry.str_sheila() + "-" + this.str_3_0_0_0();
}

```

```

v8 = v8 + "lib" + sheila3000_so + ".so";
v5_1 = v9 + "lib" + v5_1 + ".so";
File v9_1 = new File(v4_1 + "/" + v8);

```

两种可能都是加载sheila3.0.0.so文件

```

new File(v7 + "/" + v5_1);
if(v3 == 0 && Build$VERSION.SDK_INT < 19) {
    try {

```

```

        Runtime.getRuntime().exec("chmod 700 " + v7 + "/" + v5_1);
        System.load(v7 + "/" + v5_1);
    }

```

```

    catch(IOException v0_1) {
        v0_1.printStackTrace();
    }

```

```

    return;
}

```

```

if(v9_1.exists()) {
    System.loadLibrary(sheila3000_so);
    return;
}

```

## 寥寥几句onCreate

分析完attachBaseContext，接着分析onCreate

可以看到出了一个反调试和崩溃信息收集，我们的关注重点就在本地方法runCreate

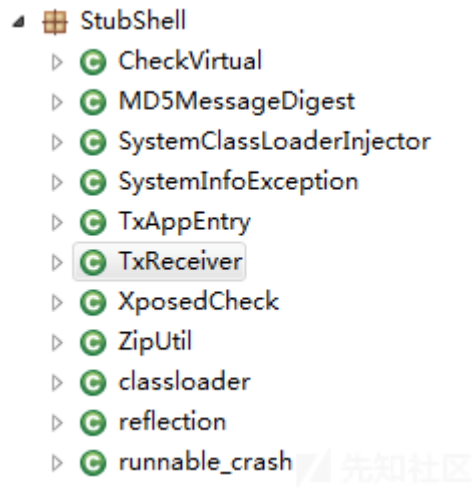
```
public void onCreate() {
    TxAppEntry.isDebugger(((Context)this));
    TxAppEntry.runCreate(((Context)this));
    this.sdkcrash(TxAppEntry.context);
}

private static native void runCreate(Context arg0) {
}
```

再度回顾加壳包目录

加固主要行为都在这里，可以从目录名称看出，多个反调试类

刨去没什么太紧要的类，只有一个TxReceiver类值得专注



通过交叉引用，并未发现有地方注册广播来执行这里，排除静态注册，剩下只有动态注册可能，都需要Native层的分析。而且他的回调方法onReceive的内部实现是通过本地方法

```
public class TxReceiver extends BroadcastReceiver {
    public static String TX_RECEIVER;

    static {
        TxReceiver.TX_RECEIVER = "com.tencent.StubShell.TxReceiver";
    }

    public TxReceiver() {
        super();
    }

    public void onReceive(Context arg1, Intent arg2) {
        TxAppEntry.receiver(arg2);
    }
}

#####TxAppEntry.java
public static void receiver(Intent arg0) {
    TxAppEntry.reciver(arg0);
}

private static native void reciver(Intent arg0) {
}
```

短暂小结，再度启程

壳的分析基本到这里暂停下来

主要分析结果：

□ 找到了唯一要加载的库shell3.0.0.0.so，根据分析流程继续分析native层的load、runCreate方法

留下的疑惑：

- 修复ufix、nfix是否得到调用
- 广播行为

Native层

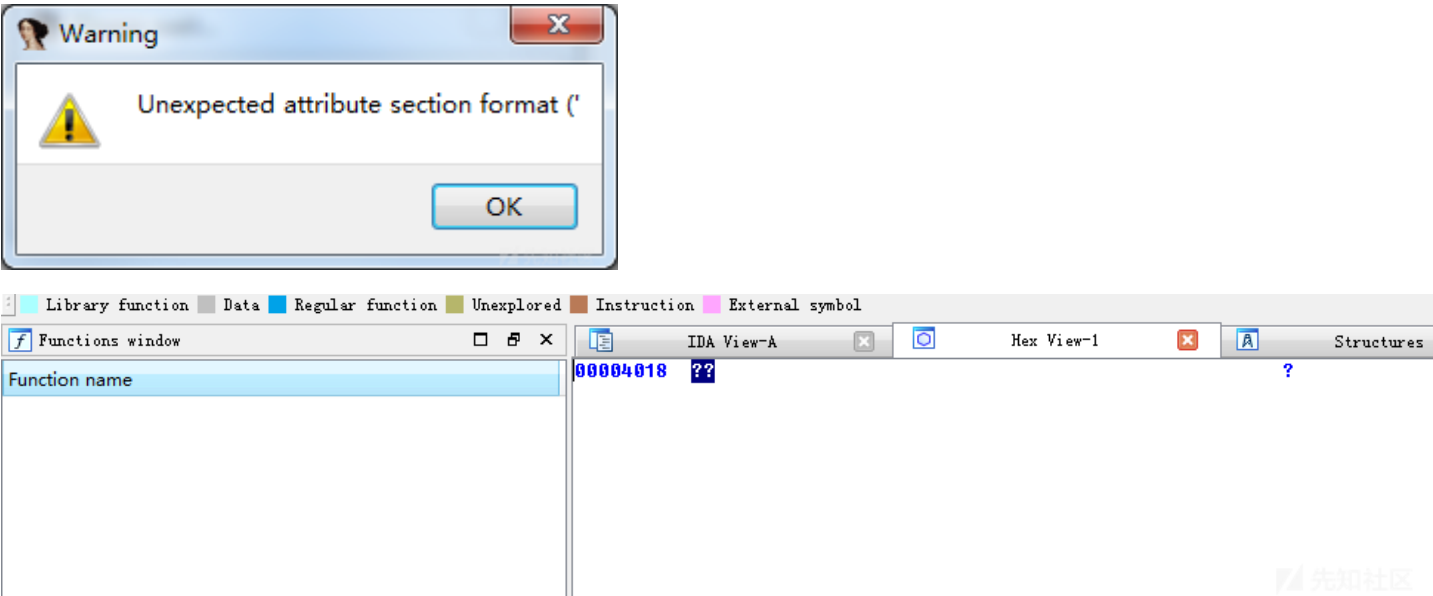
分析shella3.0.0.so，首次加载的so库

分析目标

1. 本地方法runCreate
2. java层修复ufix、nfix的fixNativeResource、fixUnityResource方法是否得到调用，做了哪些行为
3. 实锤广播注册，探索广播行为

出师未捷，对抗IDA

IDA6.8打开libshella-3.0.0.0.so弹出未识别的节格式，反编译失败，什么东西都没有！这不禁引发了我对人生的思考，是对抗反编译吗、还是对抗IDA呢？这是我需要探索的

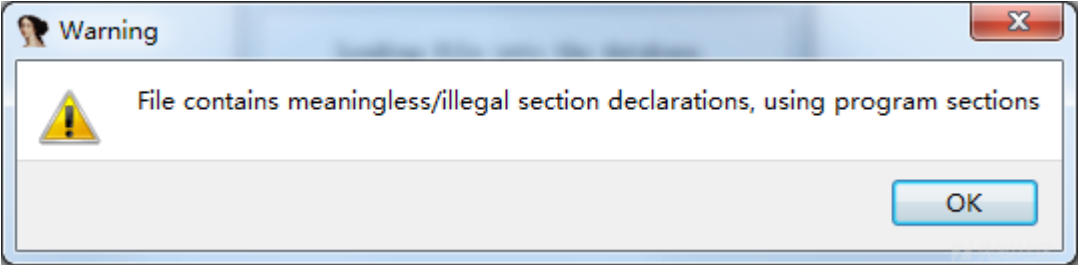


使用010edit打开so文件，可以看到解析文件是没有问题的，但是text、init等个别节头表内的数据都被抹空了，个别节头没有，如.dynstr、.dynsym

思考

- 【1】如果IDA根据节数据进行反汇编，这里数据都为空，确实会反编译失败，那么如何恢复这些节表呢？但是在看到参考【4】中文章的时候，根据之前使用经验得出一些想
- 【2】上面这种报错：检测出不识别的section格式导致终止反编译的行为很明显是对抗IDA这种反编译工具的，这也回答了上面需要探索的问题。

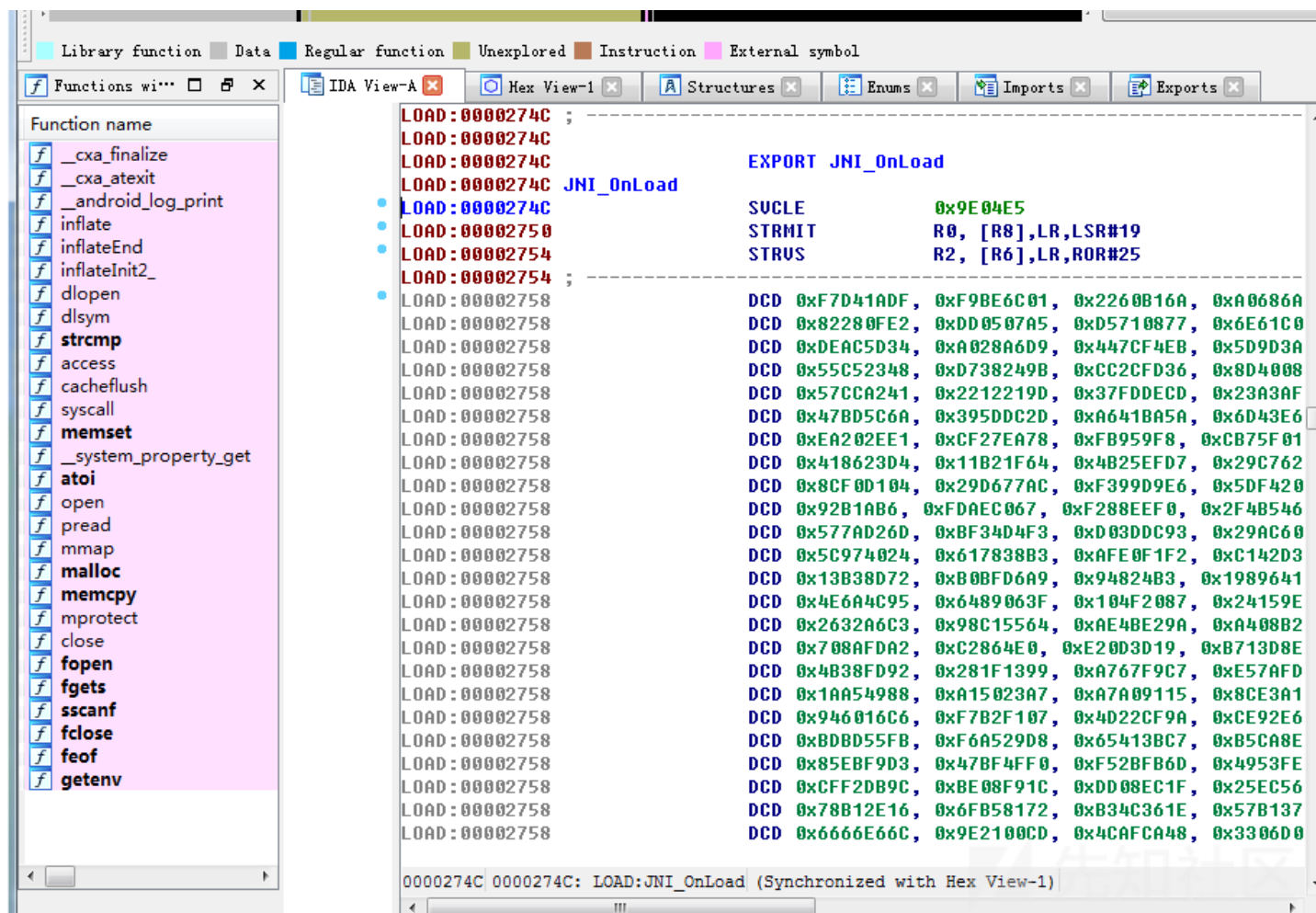
□ 为了解决其对抗IDA行为，我们这里直接将节内数据置空或者将包含字符的节数据置0，让他识别无意义或非法的节声明，接着使用程序头来进行分析即可。最终定位到.dyn



Anti不能停：JNI\_OnLoad加密

过掉AntiIDA后，再次加载so文件，可以看到导出JNI\_OnLoad函数已经被加密了（虚拟内存地址=0x274C），那么合理向上推导，只能在.init节或者.init\_array节中

接下来的目标就是找到init、init\_array节所在的地址



解决思路

【1】修复section节头

【2】动态调试so，通过在linker.so上下断点

|  |                       |
|--|-----------------------|
| struct section_table_entry32_t section_table_element[6]  | .rel.plt              |
| struct section_table_entry32_t section_table_element[7]  | .plt                  |
| struct section_table_entry32_t section_table_element[8]  | .text                 |
| struct s_name32_t s_name                                 | .text                 |
| enum s_type32_e s_type                                   | SHT_PROGBITS (1h)     |
| enum s_flags32_e s_flags                                 | SF32_Write_Alloc (6h) |
| Elf32_Addr s_addr  | 0x00000000            |
| Elf32_Off s_offset                                       | 0h                    |
| Elf32_Xword s_size                                       | 0h                    |
| Elf32_Word s_link  | 0h                    |
| Elf32_Word s_info  | 0h                    |
| Elf32_Xword s_addralign                                  | 4h                    |
| Elf32_Xword s_entsize                                    | 0h                    |
| struct section_table_entry32_t section_table_element[9]  | .code                 |
| struct section_table_entry32_t section_table_element[10] | .rodata               |
| struct section_table_entry32_t section_table_element[11] | .fini_array           |
| struct section_table_entry32_t section_table_element[12] | .init_array           |
| struct s_name32_t s_name                                 | .init_array           |
| enum s_type32_e s_type                                   | Eh                    |
| enum s_flags32_e s_flags                                 | SF32_Alloc_Exec (3h)  |
| Elf32_Addr s_addr  | 0x00000000            |
| Elf32_Off s_offset                                       | 0h                    |
| Elf32_Xword s_size                                       | 0h                    |
| Elf32_Word s_link  | 0h                    |
| Elf32_Word s_info  | 0h                    |
| Elf32_Xword s_addralign                                  | 4h                    |
| Elf32_Xword s_entsize                                    | 0h                    |
| struct section_table_entry32_t section_table_element[13] | .dynamic              |
| struct section_table_entry32_t section_table_element[14] | .got                  |

section修复，觅得init\_array

修复之前多个节都是置空的，还有个别节错误数据来Anti IDA

|  |                       |
|--|-----------------------|
| struct section_table_entry32_t section_table_element[6]  | .rel.plt              |
| struct section_table_entry32_t section_table_element[7]  | .plt                  |
| struct section_table_entry32_t section_table_element[8]  | .text                 |
| struct s_name32_t s_name                                 | .text                 |
| enum s_type32_e s_type                                   | SHT_PROGBITS (1h)     |
| enum s_flags32_e s_flags                                 | SF32_Write_Alloc (6h) |
| Elf32_Addr s_addr  | 0x00000000            |
| Elf32_Off s_offset                                       | 0h                    |
| Elf32_Xword s_size                                       | 0h                    |
| Elf32_Word s_link  | 0h                    |
| Elf32_Word s_info  | 0h                    |
| Elf32_Xword s_addralign                                  | 4h                    |
| Elf32_Xword s_entsize                                    | 0h                    |
| struct section_table_entry32_t section_table_element[9]  | .code                 |
| struct section_table_entry32_t section_table_element[10] | .rodata               |
| struct section_table_entry32_t section_table_element[11] | .fini_array           |
| struct section_table_entry32_t section_table_element[12] | .init_array           |
| struct s_name32_t s_name                                 | .init_array           |
| enum s_type32_e s_type                                   | Eh                    |
| enum s_flags32_e s_flags                                 | SF32_Alloc_Exec (3h)  |
| Elf32_Addr s_addr  | 0x00000000            |
| Elf32_Off s_offset                                       | 0h                    |
| Elf32_Xword s_size                                       | 0h                    |
| Elf32_Word s_link  | 0h                    |
| Elf32_Word s_info  | 0h                    |
| Elf32_Xword s_addralign                                  | 4h                    |
| Elf32_Xword s_entsize                                    | 0h                    |
| struct section_table_entry32_t section_table_element[13] | .dynamic              |
| struct section_table_entry32_t section_table_element[14] | .got                  |

通过[开源代码](#)对so文件进行修复后，在linux平台用readelf可以看到已经将很多节头的偏移恢复了，

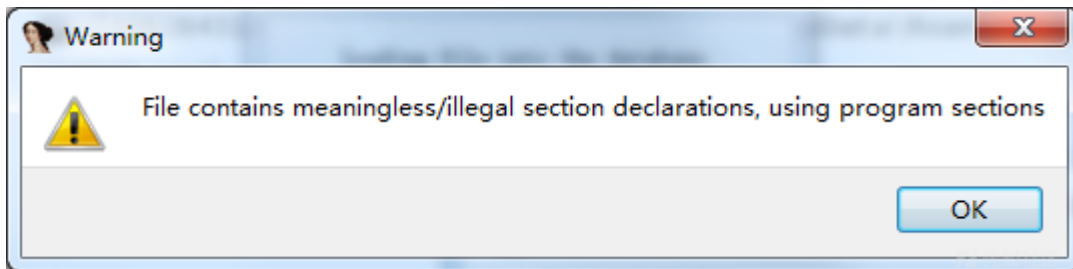
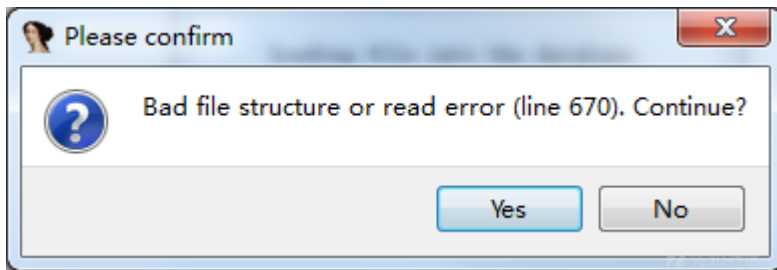
```
There are 16 section headers, starting at offset 0x69bd:
readelf: fix.so: Error: Reading 0x166a bytes extends past end of file for dynamic strings

Section Headers:
[Nr] Name                Type              Addr             Off             Size            ES Flg Lk Inf Al
[ 0]                      NULL              00000000         000000         000000         00   0  0  0
[ 1] .dynsym                 DYNSYM            0000401c         00401c         001874         10   A  2  1  4
[ 2] .dynstr                 STRTAB            00005890         005890         00166a         00   A  0  0  1
[ 3] .hash                   HASH              00006efc         006efc         d9e412b8       04   A  4  1  4
[ 4] .rel.dyn                REL               00000550         000550         000010         08   A  4  0  4
[ 5] .rel.plt                REL               00000560         000560         0000e0         08   A  1  6  4
[ 6] .plt                    PROGBITS          00000640         000640         000164         00  AX  0  0  4
[ 7] .text@.ARM.extab        PROGBITS          000007a4         0007a4         fffff85c       00  AX  0  0  0
[ 8]                      NULL              00000000         000000         000000         00   0  0  0
[ 9] .fini_array             FINI_ARRAY        00003e7c         002e7c         000008         00  WA  0  0  4
[10] .init_array             INIT_ARRAY        00003e84         002e84         000008         00  WA  0  0  4
[11] .dynamic                DYNAMIC           00003e8c         002e8c         0000f8         08  WA  2  0  4
[12] .got                    PROGBITS          00003f84         002f84         00007c         00  WA  0  0  4
[13] .data                   PROGBITS          00004000         003000         003938         00  WA  0  0  4
[14] .bss                    NOBITS            00007938         006938         000000         00  WA  0  0  1
[15] .shstrtab               STRTAB            00000000         006938         000085         00   0  0  1

Key to Flags:
W (write), A (alloc), X (execute), M (merge), S (strings), I (info),
L (link order), O (extra OS processing required), G (group), T (TLS),
C (compressed), x (unknown), o (OS specific), E (exclude),
y (nored), p (processor specific)
```

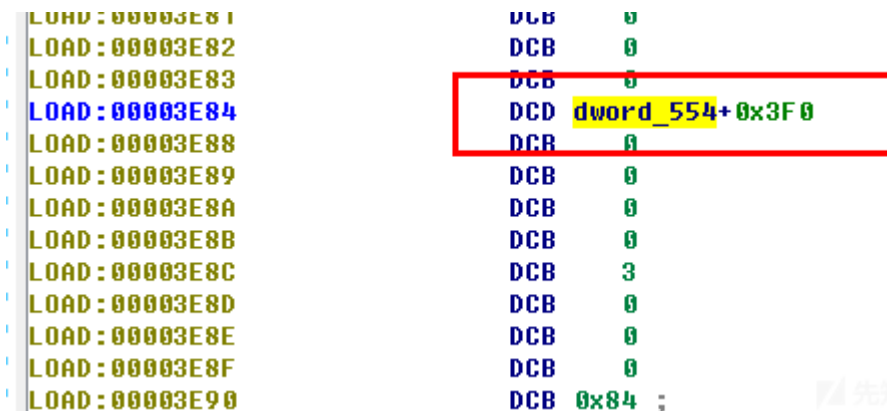
在ida6.8打开时，首先出现下面两个弹窗中的出现的错误，全部确认



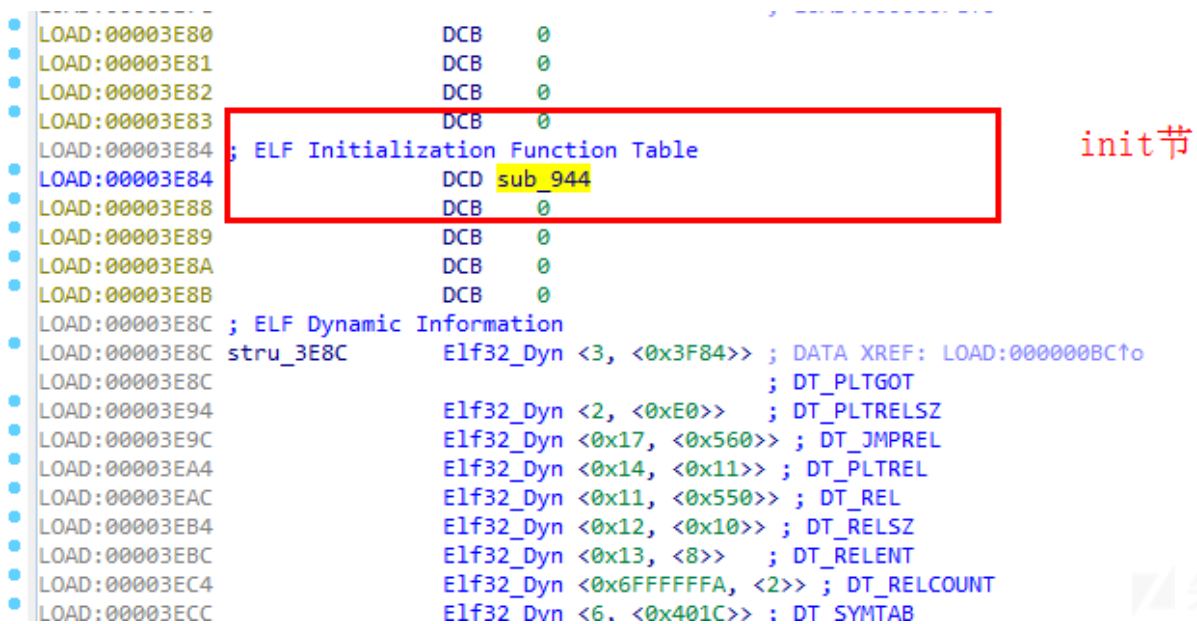


我们根据觅得的init\_array地址，抱着兴奋的情绪进行G跳转到0x3e84，这里切记别乱改数据类型，这里应该是DCD代表双字，代表的地址是0x944。

这里我犯了个错，由于不太熟吧，乱改数据类型，改成DCB字节型，结果转成代码后就懵了，在心灰意冷下我打开了IDA7.2，看到下面那个图，一度让我准备和IDA 6.8 say 拜拜。但是由于7.2 F5大法不管用（原因暂时未知），6.8还是很棒的，还是和它做好基友吧



这里要是用IDA7.2版本，他这里会识别出init节并标记（感觉棒棒哒）



通读伪代码，分析init\_array

这里主要分析出：

- 解密算法是从0x1000开始，对0x2AB4字节数据进行解密(JNI\_OnLoad地址为0x274c必然被包含在内)
- 调用JNI\_OnLoad

分析出解密算法，可以自己写脚本进行解密，这里我们选择另外一种，往下看

```

for ( i = (unsigned int)init & 0xFFFF000; *(_DWORD *)i != 0x464C457F; i -= 4096 ) // i=she11a-3.0.0.0.so在内存中的基地址
;
e_phoff = *(_DWORD *)i + 0x1C; // 程序头表在文件中的偏移
v13 = 0;
while ( *(_WORD *)i + 0x2C > v13 ) // 遍历程序头
{
    if ( *(_DWORD *)e_phoff != 1 || *(_DWORD *)e_phoff + 0x18 != 5 )
    {
        if ( *(_DWORD *)e_phoff == 1 && *(_DWORD *)e_phoff + 0x18 == 6 )
        {
            v1 = *(_DWORD *)e_phoff + 8 & 0xFFFF000;
            v2 = *(_DWORD *)e_phoff + 8 + *(_DWORD *)e_phoff + 0x10 + 4095 & 0xFFFF000;
            break;
        }
    }
    else
    {
        v0 = *(_DWORD *)e_phoff + 8 + *(_DWORD *)e_phoff + 0x16 + 0xFFF & 0xFFFF000; // p_vaddr + p_filesz + 0xFFF
    }
    ++v13;
    e_phoff += 32;
}
v12 = 0x2B;
v11 = 0x99u;
v10 = 0x20;
v9 = 0x15;
v3 = (unsigned __int64)(unsigned int)dword_4008 << 16; // 0x000010002AB40000
v7 = (unsigned __int16)dword_4008; // 0x2AB4
v6 = (unsigned __int16)dword_4008 - ((unsigned int)dword_4008 >> 16); // 0x1AB4
mprotect(i + ((unsigned int)dword_4008 >> 16), (v6 + 0xFFF) & 0xFFFF000, 3); // mprotect(ELFHeaderAddr + 0x1000, 0x2000, 3)
for ( j = HIWORD(v3); j <= v7; ++j ) // For(j=0x1000; j<0x2AB4; ++1)
{
    v4 = *(_BYTE *)i + j;
    *(_BYTE *)i + j ^= (unsigned __int8)((v11 - v10) ^ j) + v9 ^ v12;
    *(_BYTE *)i + j += v10 & v9 ^ v11;
    v12 += (v11 + v10 - v9) & v4 & j;
    v11 += (j + v12) ^ v4;
    v10 ^= (v4 - v12) ^ j;
    v9 += j - (v4 + v12);
}
mprotect(i + HIWORD(v3), (v6 + 0xFFF) & 0xFFFF000, 5);
cacheFlush(i + HIWORD(v3), v6);
dword_4008 = i;
return int_jniOnload();

```

解密算法

## 另辟蹊径，解密JNI\_OnLoad

思路：so库一经加载到内存后，要处于解密后的状态才可以正常被程序调用，所以从内存中dump出she11a-3.0.0.0.so文件，即完成对JNI\_OnLoad解密的操作

无意之举吧，：)

当时准备通过调试获取init\_array内存地址的时候没有成功，当时想着dump下so文件应该包含有解密后的节头表，后来看到一篇文章结合ELF装载知识才知道节头表并不

解密脚本，具体内存地址和加载进内存的段长度，需要自己调试的时候Ctrl+S自己看和计算

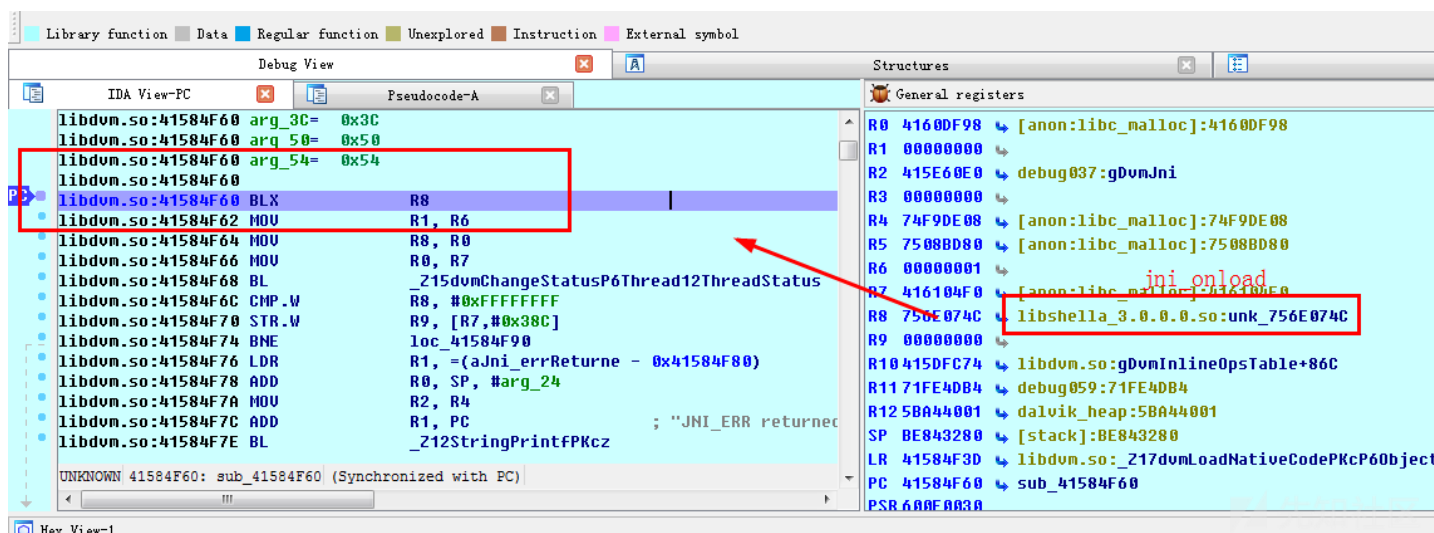
```

static main()
{
    auto i,fp;
    fp = fopen("d:\\dump","wb");
    auto start = 0x75FFD000;
    auto size = 32768;
    for(i=start;i<start+size;i++)
    {
        fputc(Byte(i),fp);
    }
}

```

## 真实调用

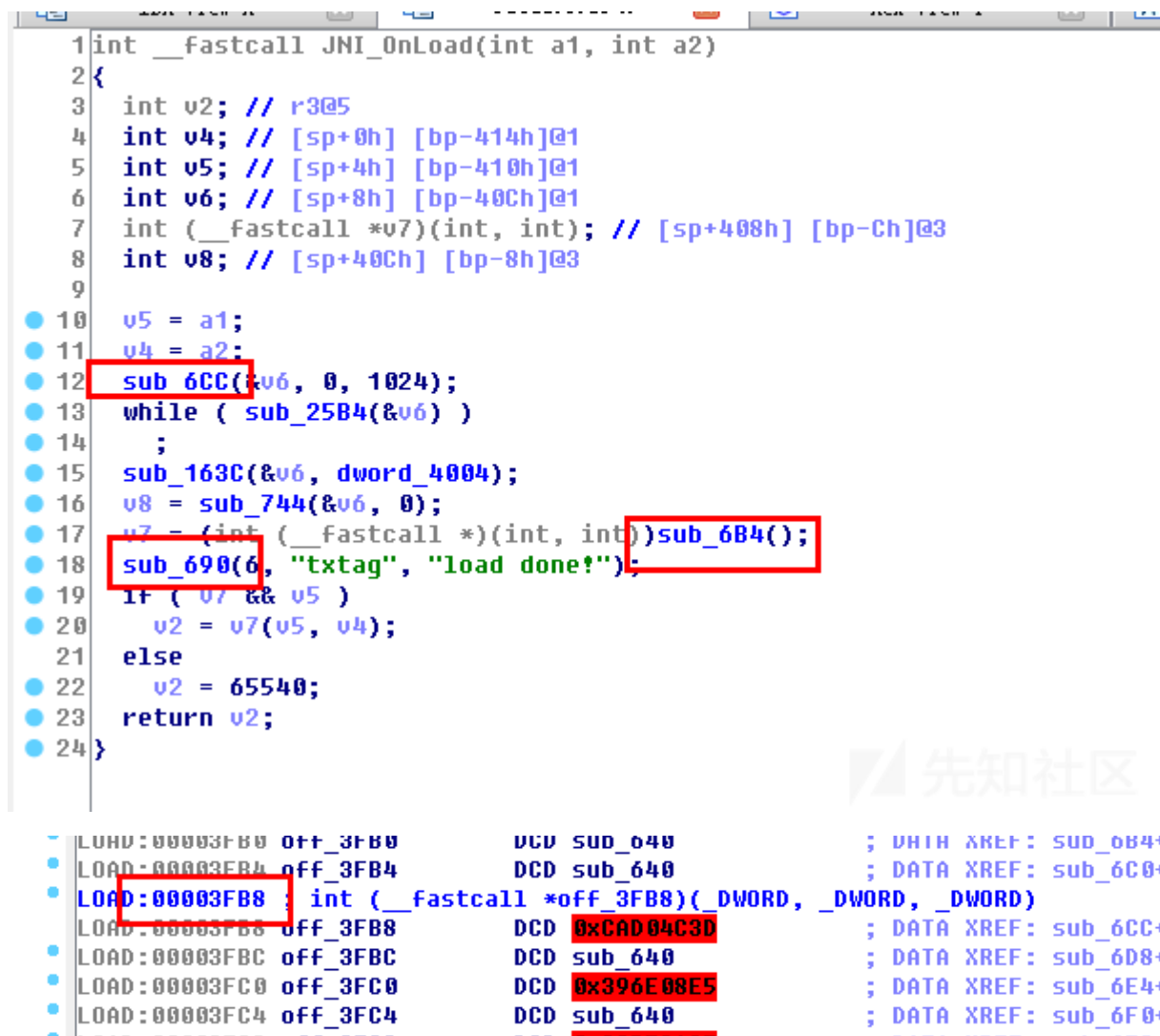
在动态调试的过程中，调用JNI\_OnLoad方法的地方不是init\_array节内，而是libdvm.so文件中的dvmLoadNativeCode方法。



分析不能停，探索JNI\_OnLoad

初遇小坑

图中圈起来的函数，最终跳到类似0x3FB8地址出的地方，为什么这个地方的函数地址是找不到的呢？



蓦然回首，原来是重定位

由于这里调用的是第三方库函数，这里就用到了PLT表，每次调用第三方库函数都会跳到PLT条目中。这个表有关第三方函数的每一个条目都指向了GOT表条目的值，第一次所以上面之所以找不到库函数地址，是因为重定位后被改写后的内存地址，在静态文件中是不能识别的。

绕过也是很简单的，因为我们解密的数据长度有限，我们将解密部分替换到原来的shella-3.0.0.0.so文件中即可，再次打开如下图所示，都是一些偏移可以被IDA识别出来

```

- 000003FB4 0+ 3FB4 DCD 100 ACCESS : DATA XREF: ACCESS+81r
LOAD:00003FB8 ; int (__fastcall *off_3FB8)(_DWORD, _DWORD, _DWORD)
LOAD:00003FB8 off_3FB8 DCD __imp_memset ; DATA XREF: memset+81r
LOAD:00003FBC off_3FBC DCD __imp__system_property_get
LOAD:00003FBC ; DATA XREF: __system_property_get+81r
LOAD:00003FC0 off_3FC0 DCD __imp_atoi ; DATA XREF: atoi+81r
LOAD:00003FC4 off_3FC4 DCD __imp_open ; DATA XREF: open+81r
LOAD:00003FC8 off_3FC8 DCD __imp_pread ; DATA XREF: pread+81r
LOAD:00003FCC off_3FCC DCD __imp_mmap ; DATA XREF: mmap+81r

```

再现加密

第一次解密中的行为，这里i本身就是libshella-3.0.0.0.so文件的内存基址，这里将地址存进dword\_4008变量中

```

40  v7 = (unsigned __int16)dword_4008;
47  v6 = (unsigned __int16)dword_4008 - ((unsigned int)dword_4008 >> 16);
48  mprotect(i + ((unsigned int)dword_4008 >> 16), (v6 + 4095) & 0xFFFF000, 3);
49  for ( j = HIWORD(v3); j <= v7; ++j )
50  {
51      v4 = *(_BYTE *)(i + j);
52      *(_BYTE *)(i + j) ^= (unsigned __int8)(((v11 - v10) ^ j) + v9) ^ v12;
53      *(_BYTE *)(i + j) += v10 & v9 ^ v11;
54      v12 += (v11 + v10 - v9) & v4 & j;
55      v11 += (j + v12) ^ v4;
56      v10 ^= (v4 - v12) ^ j;
57      v9 += j - (v4 + v12);
58  }
59  mprotect(i + HIWORD(v3), (v6 + 4095) & 0xFFFF000, 5);
60  cacheFlush(i + HIWORD(v3), v6);
61  dword_4008 = i;
62  return sub_898();
63 }

```

这里其实就是读取shella-3.0.0.0.so文件的名称到变量中

```

v3 = a1;
fp = fopen("/proc/self/maps", "r");
memset(&v7, 0, 0x400u);
memset(&v6, 0, 0x400u);
v5 = 0;
v4 = 0;
v8 = dword_4008;
while ( !feof(fp) )
{
    fgets((char *)&v7, 1024, fp);
    sscanf((const char *)&v7, "%lx-%lx %s %s %s %s %s", &v5, &v4, &v6, &v6, &v6, &v6, v3);
    if ( v5 <= v8 && v8 < v4 )
    {
        fclose(fp);
        return 0;
    }
}
fclose(fp);
return -1;
}

```

只有当找到基址时才可以返回0

接着将得到的libname和一个偏移值0x6D88（刚好指向libshella-3.0.0.0.so文件尾部附加数据开始的位置）作为参数传进函数内，执行以下操作

- 总共三次从尾部读取所有数据到内存，并进行解密运算

```

1 int_0x0088 = atoi(&v01);
2 _android_log_print(6, (int)"txntag", "version:%d", int_0x0088);
3 do
4     fp = open(addr_shella, 0x80000); // 打开/data/app-lib/repak_sign/libshella-3.0.0.0.so文件
5     while ( fp == -1 );
6     memset(&v41, 0, 0x580);
7     for ( i = 0; i <= 0x57; i = pread(fp, &v41, 0x58, int_x6D88) )// 第一次读数据: 从偏移0x6D88处(so文件尾部附加数据
8     ;
9     _android_log_print(6, (int)"txntag", "load library %s at offset %x read count %x\n", addr_shella, int_x6D88, i);
10    memset(&v23, 0, 0x1A00);
11    _android_log_print(6, (int)"txntag", "min_vaddr:%x size:%x\n", v41, int_0x20c9c);
12    int_0 = v41;
13    do
14    {
15        v10 = 0;
16        Pmem = mmap(int_0, int_0x20c9c, 0, 34); // 随机申请一块长度为0x20C9C长度的内存块
17        if ( Pmem <= 0x457FFFFFFF && Pmem > 0x40000000 - int_0x20c9c && (unsigned int)int_0x6D88 <= 0xA )
18        {
19            _android_log_print(6, (int)"txntag", "addr:%p", Pmem, -1, 0);
20            Pmem = -1;
21        }
22    }
23    while ( Pmem == -1 );
24    int_pMem_ = Pmem - int_0;
25    int_pMem_ = Pmem;
26    int_pMem_ = Pmem - int_0;
27    pMem = (void *) (int_0x0 + Pmem - int_0);
28    v26 = (void *) (int_0x19b8 + Pmem - int_0);
29    if ( int_0x148 )
30        v2 = (void ( __fastcall *) (int)) (int_0x148 + int_pMem_);
31    else
32        v2 = 0;
33    pMem_off_0x148 = v2;
34    if ( int_7da9 )
35        v3 = int_7da9 + int_pMem_;
36    else
37        v3 = 0;
38    pMem_off_0x7da9 = v3;
39    int16_0xf4f8_ = int16_0xf4f8;
40    int_0x1f4f0_ = int_0x1f4f0;
41    pMem_off_0x6 = (void *) (int_0x6 + int_pMem_);
42    int_0x50002_ = int_0x50002;
43    v30 = int_0x6 + int_pMem_ + 4 * int_0x1f4f0;
44    v38 = int_48e4;
45    v39 = int16_0x65;
46    pMem_off_0x88 = (void *) (int_0x88 + int_pMem_);
47    int_0x107_ = int_0x107;
48    pMem_off_0x302c = (void *) (int_0x302c + int_pMem_);
49    int_0x187_ = int_0x187;
50    _android_log_print(6, (int)"txntag", "load_bias:%p base:%p\n", int_pMem_, Pmem, v10);
51    v73 = malloc(24 * int16_0x2);
52    pMem2_0x48 = (unsigned int)v73;
53    for ( i = 0; 24 * (unsigned int)int16_0x2 > i; _android_log_print(6, (int)"txntag", "read count:%x", i) )
54        i = pread(fp, pMem2_0x48, 24 * int16_0x2, int16_0x58 + int_x6D88);// 第二次读数据: 继续接着从尾部读取0x30字节
55    v81 = 0;
56    while ( int16_0x2 > v81 )
57    {
58        addr_pMemAndpMem2 = *(_DWORD *) pMem2_0x48 + int_pMem_;
59        v71 = *(_DWORD *) (pMem2_0x48 + 4) + addr_pMemAndpMem2;
60        pMem_ = addr_pMemAndpMem2 & 0xFFFFF000;
61        v69 = (v71 + 0xFFF) & 0xFFFFF000;
62        v68 = v69 - (addr_pMemAndpMem2 & 0xFFFFF000);
63        i = 0;
64        v80 = 0;
65        if ( *(_DWORD *) (pMem2_0x48 + 12) )
66        {
67            v9 = -1;
68            v11 = 0;
69            mmap(pMem_, v68, 3, 50);
70            v18 = 0;
71            v19 = 0;
72            v20 = 0;
73            v15 = 0;
74            addr_decryptData = 0;
75            while ( inflateInit2((int)&addr_decryptData, -15, (int)"1.2.3", 56) )
76            ;
77            while ( *(_DWORD *) (pMem2_0x48 + 12) > i )// 循环读取数据, 每次读0x1000
78            {
79                if ( i + 4096 <= *(_DWORD *) (pMem2_0x48 + 12) )
80                    v4 = 0x1000;
81                else
82                    v4 = *(_DWORD *) (pMem2_0x48 + 12) - i;
83                v67 = v4;
84                if ( i + 4096 <= *(_DWORD *) (pMem2_0x48 + 20) )
85                    v5 = 4096;
86                else
87                    v5 = *(_DWORD *) (pMem2_0x48 + 20) - i;
88                v66 = v5;
89                v67 = pread(fp, &v21, v67, *(_DWORD *) (pMem2_0x48 + 8) + int_x6D88 + i);// 第三次读数据: 继续从尾部数据的
90                decrypt((int)"Tx:12345Tx:12345", (int)&v21, v66, 16);// 针对读取的数据进行运算, 很明显是解密操作
91                v15 = v67;
92                addr_decryptData = &v21;
93                _android_log_print(6, (int)"txntag", "read count:%x", v67, v9, v11);
94                v17 = 0x100000;
95                v16 = v80 + addr_pMemAndpMem2;
96                v65 = inflate((int)&addr_decryptData, 0);

```

这里调用了dlsym来在so文件中找到JNI\_OnLoad符号地址并进行调用。

分析到这里其实除了之前的解密操作，我们并没有看到任何动态注册本地方法的地方，那么结合这里出现符号调用可以大胆猜想，这里可能会是二次解密后得到的JNI\_OnLoad

```
#
0  v5 = a1;
1  v4 = a2;
2  memset(&libName_shella, 0, 0x400u);
3  while ( getLibName((int)&libName_shella) )
4      ;
5  sub_103C((int)&libName_shella, int_0x0088);
6  pcurrent = dlopen((int)&libName_shella, 0);
7  JNI_OnLoad = (int (__fastcall *) (int, int))dlsym(pcurrent, (int)"JNI_OnLoad");
8  _android_log_print(6, (int)"txtag", "load done!");
9  if ( JNI_OnLoad && v5 )
10     v2 = JNI_OnLoad(v5, v4);
11 else
12     v2 = 0x10004;
13 return v2;
14 }
```

先知社区

动态调试跟进解密后的JNI\_OnLoad方法

这里将壳入口类名作为参数传进函数，下面判断如果返回结果为0则打印出注册本地方法失败这样的字符串

```
EDCF8
EDCF8 sub_756EDCF8                                ; CODE XREF: sub_756EDD34+5A↑p
EDCF8 LDR                                         ; 壳入口
EDCFA LDR                                         R2, =(unk_75706004 - 0x756EDD04)
EDCFC PUSH                                       {R4,LR}
EDCFE ADD                                         R1, PC
EDD00 ADD                                         R2, PC ; unk_75706004
EDD02 MOVS                                       R3, #5
EDD04 BL                                         unk_756EDCB4
EDD08 SUBS                                       R4, R0, #0
EDD0A BNE                                        loc_756EDD1C
EDD0C LDR                                         R1, =(aSecshell - 0x756EDD16)
EDD0E LDR                                         R2, =(aRegisterNative - 0x756EDD18)
EDD10 MOVS                                       R0, #3
EDD12 ADD                                         R1, PC
EDD14 ADD                                         R2, PC
EDD16 BL                                         sub_756FDC30
EDD1A B                                           loc_756EDD1E
EDD1C ;
EDD1C
EDD1C loc_756EDD1C                                ; CODE XREF: sub_756EDCF8+12↑j
EDD1C MOVS                                       R4, #1
```

打印出注册失败  
字符

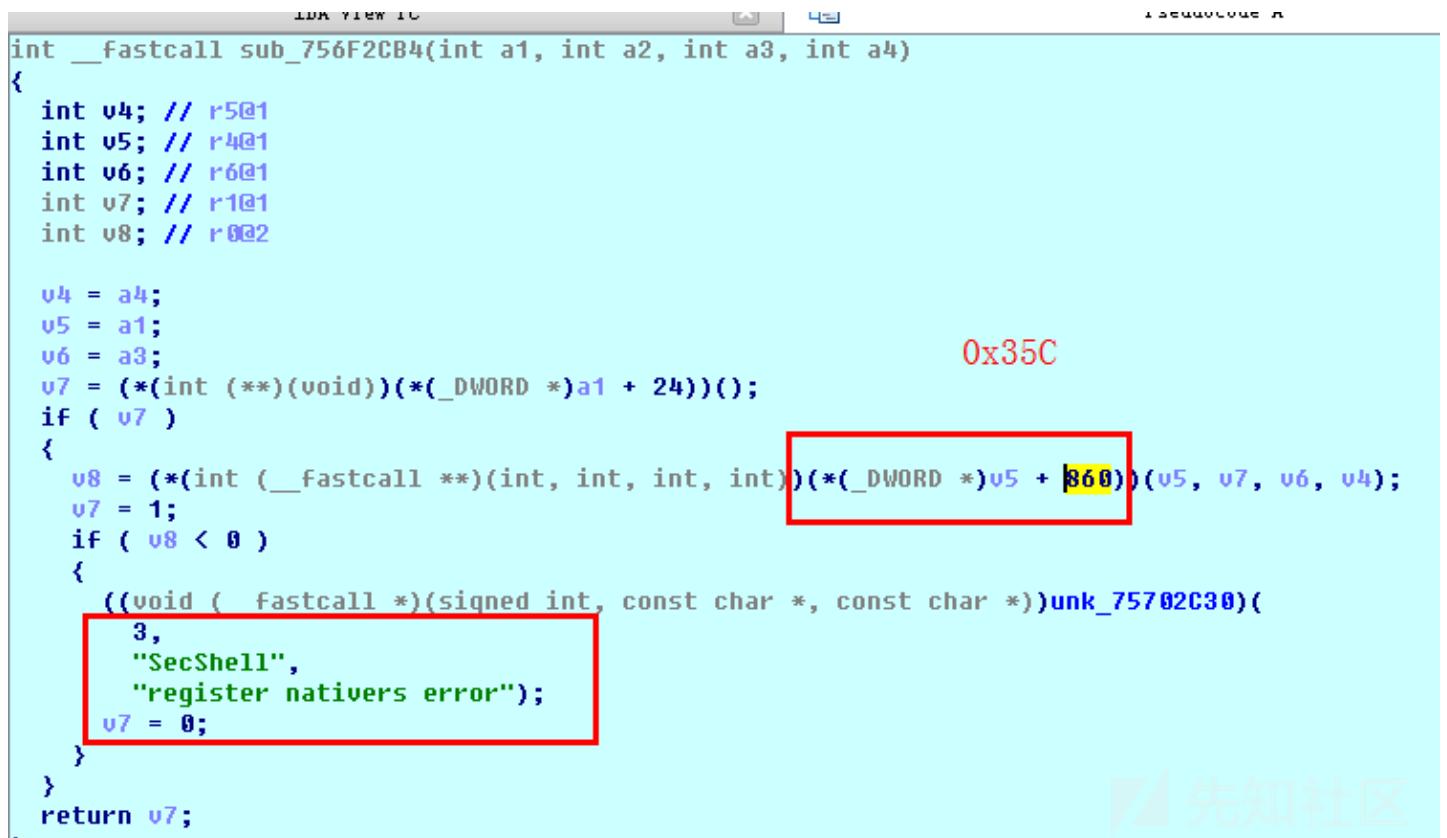
先知社区

根据传入壳的入口类名作为参数进行类定位和注册本地方法

```
7  char *v7; // [sp+10h] [bp-28h]@1
8  int v8; // [sp+14h] [bp-24h]@1
9  char v9; // [sp+18h] [bp-20h]@1
10
11 v2 = a1;
12 v3 = (const char *)entryClassName;
13 v6 = a1;
14 v7 = "FindClass";
15 v8 = 0;
16 v9 = 1;
17 sub_4156DD70((int)&v6, 0);
18 sub_4156E714((int)&v6, 1, (int)&unk_415C491B, v2);
19 sub_4156D9D4((int)&v6, v3);
20 v4 = (*(int (__fastcall **)(int, const char *))((_DWORD *) (v2 + 4) + 24))(v2, v3);
21 sub_4156E714((int)&v6, 0, (int)&loc_415C49E4, v4);
22 return v4;
23 }
```

先知社区

惊现：0x35C



```
int __fastcall sub_756F2CB4(int a1, int a2, int a3, int a4)
{
    int v4; // r5@1
    int v5; // r4@1
    int v6; // r6@1
    int v7; // r1@1
    int v8; // r0@2

    v4 = a4;
    v5 = a1;
    v6 = a3;
    v7 = (*(int (**)(void))(*(_DWORD *)a1 + 24))();
    if ( v7 )
    {
        v8 = (*(int (__fastcall **)(int, int, int, int))(*(_DWORD *)v5 + 0x35C))(v5, v7, v6, v4);
        v7 = 1;
        if ( v8 < 0 )
        {
            ((void ( __fastcall *)(signed int, const char *, const char *))unk_75702C30)(
                3,
                "SecShell",
                "register natives error");
            v7 = 0;
        }
    }
    return v7;
}
```

发现偏移0x35C，这正是registerNatives相对于JNINativeInterface的偏移。他的第三个参数是JNINativeMethod结构体数组，第四个参数就是结构体数组的长度，注册方

```
typedef struct {
    const char* name;
    const char* signature;
    void* fnPtr;
} JNINativeMethod;
```

解析本地方法

注册方法数量为5。

本地方法对应内存地址

load 0x75700B1D

runCreate 0x756fc469

changeEnv 0x756FB37D

receiver 0x756f7621

txEntries 0x756FB0F9

```
0B 9E 70 75 10 9E 70 75 1D 0B 70 75 2D 9E 70 75
10 9E 70 75 69 C4 6F 75 37 9E 70 75 10 9E 70 75
7D B3 6F 75 41 9E 70 75 49 9E 70 75 21 76 6F 75
65 9E 70 75 6F 9E 70 75 F9 B0 6F 75
```

骤现异常

要分析上面本地方法，就需要配合动态调试综合来进行。但是当我们在load方法上下断点后，程序并不能执行到这里，从日志中反馈一个signal 11的错误，并且程序也不能正常跑起来，弹出应用已经停止的窗口。

思考：我这里为了调试，第一：只是将AndroidManifest.xml文件添加了一个可调试属性。第二注释掉了几个public.xml中的几个无关属性防止反编译失败。第三就是签上了11



```

I/DEBUG(18905): pid: 22161, tid: 22161, name: com.warmcar.hf.x >>> com.warmcar.hf.x <<<
I/DEBUG(18905): [signal] 11 (SIGSEGV), code 1 (SEGV_MAPERR), fault addr 00000020
I/DEBUG(18905):    r0 00000020  r1 00000000  r2 00000010  r3 00000013
I/DEBUG(18905):    r4 00000000  r5 00000000  r6 74f14028  r7 759966f0
I/DEBUG(18905):    r8 be843200  r9 6d4c5b60  s1 41610500  fp be843214
I/DEBUG(18905):    ip 75995f3c  sp be8429c4  lr 75980fab  pc 40122108  cpsr 000d0010
I/DEBUG(18905):    d0 0000000000000000  d1 0000000000000000
I/DEBUG(18905):    d2 0000000000000000  d3 0000000000000000
I/DEBUG(18905):    d4 0000d82a584a5a90  d5 5a66be9100008c18
I/DEBUG(18905):    d6 000092b80000e2c2  d7 0000e4f74d3d1f93
I/DEBUG(18905):    d8 0000000000000000  d9 0000000000000000
I/DEBUG(18905):    d10 0000000000000000  d11 0000000000000000
I/DEBUG(18905):    d12 0000000000000000  d13 0000000000000000
I/DEBUG(18905):    d14 0000000000000000  d15 0000000000000000
I/DEBUG(18905):    d16 0000000000000000  d17 0000000000000000
I/DEBUG(18905):    d18 0000000000000004  d19 412e848000000000
I/DEBUG(18905):    d20 4022000000000000  d21 4008000000000000
I/DEBUG(18905):    d22 3ff0000000000000  d23 4008000000000000
I/DEBUG(18905):    d24 4024000000000000  d25 4000000000000000
I/DEBUG(18905):    d26 4000000000000000  d27 414e848000000000
I/DEBUG(18905):    d28 0800000009000000  d29 0001000000010000
I/DEBUG(18905):    d30 010b400001088000  d31 01108000010e0000
I/DEBUG(18905):    scr 20000010
I/DEBUG(18905): backtrace:
I/DEBUG(18905):    #00 pc 00022108 /system/lib/libc.so (memset+44)
I/DEBUG(18905):    #01 pc 0000afa7 <unknown>
I/DEBUG(18905): stack:
I/DEBUG(18905):    be842984 4e213ab8 /dev/ashmem/dalvik-heap (deleted)

```



在网上找到一些类似的解决方法，先用addr2line命令定位出错的地方在库文件的什么地方，根据栈回溯backtrace打印出的内容来定位：arm-none-linux-gnueabi-addr2line -e libc.so，返回结果为???

这里我们卡在了脱壳的过程中，该解密的区段都已经解密成功了，就在即将要开始调用java层的native方法的时候，这里出现signal 11的错误，怎么办呢？

退一步海阔天空，注入大法好

虽然暂时无法确定出问题的细节，但是大致方向是可以把握的：因为重打包后，程序出现崩溃。

为什么要重打包？因为要修改AndroidManifest.xml文件增加可调试属性，否则jdb无法启动应用。

那么有办法替代修改调试属性的操作吗？有，参考【9】，init注入或者xposed。这里直接用写好的工具mprop，执行./mprop ro.debuggable 1即可。

绕过反调试，手动绕过isDebuggerConnected

当我们开始调试的时候，其实java层有一个反调试，就在壳代码中，最开始是通过反编译smali代码，删除相应代码来对抗它的，但是因为反编译后会出现程序异常，我们这

思路：

```

ice static boolean com.tencent.studiohell.txappentry.e(
    android.content.Context v3)
    const/4                v0, 1
    # CODE XREF: TxAppEntry_attachBaseContext@UL+64j
    # TxAppEntry_onCreate@ULj
    const/4                v1, 0
    invoke-static           {}, <boolean Debug.isDebuggerConnected() imp. @ _def_Debug_isDebuggerConnec
    move-result            v2
    if-ne                   v2, v0, loc_1FA3C
    invoke-static           {}, <int Process.myPid() imp. @ _def_Process_myPid@I>
    move-result            v1
    invoke-static           {v1}, <void Process.killProcess(int) imp. @ _def_Process_killProcess@UI>
    # CODE XREF: TxAppEntry_e@2L+224j

```



【1】patch掉该处代码，重新修改dex文件头的signature和checksum

【2】动态修改isDebuggerConnected的返回值，参考【10】

这里我用的第二种方法

load：核心逻辑

顺利到在load函数中下上断点。



```

v4 = a3;
((void (*)(void))hook)();
(*(void (__fastcall *)(int, void *)))(*(DWORD *)v3 + 876)(v3, &unk_7605CBAC);
v5 = ((int (__fastcall *)(int, const char *))sub_760435FA)(v3, "com.tencent/StubShell/TxAppEntry");
v6 = v5;
v7 = ((int (__fastcall *)(int, int, const char *, const char *))unk_76045EA4)(
    v3,
    v5,
    "mSrcPath",
    "Ljava/lang/String;");
v8 = ((int (__fastcall *)(int, int, int))unk_76045EB2)(v3, v6, v7);
v9 = ((int (__fastcall *)(int, int))sub_76044388)(v3, v8);
v10 = ((int (__fastcall *)(int))unk_76053C40)(v9);
((void (*)(signed int, const char *, const char *, ...))log_print)(3, "SecShell", "Start load %d", v10);
result = ((int (__fastcall *)(int))unk_76046544)(v3);
if ( result )
{
    v12 = ((int (__fastcall *)(int))isART)(v3);
    ((void (__fastcall *)(int, int))sub_760436C8)(v3, v4);
    if ( v12 )
    {
        ((void (__fastcall *)(signed int, const char *, const char *))log_print)(3, "SecShell", "fix-1");
        ((void (__fastcall *)(int, int))unk_76049388)(v3, v4);
        v13 = "SecShell";
        v14 = "fix-1-1";
    }
    else
    {
        ((void (__fastcall *)(signed int, const char *, const char *))log_print)(3, "SecShell", "fix-2");
        ((void (__fastcall *)(int, int, DWORD))sub_7604B8E0)(v3, v4, 0);
        v13 = "SecShell";
        v14 = "fix-2-1";
    }
}

```

下面主要分析核心内容。

- 获取odex基址，0x750DD000

The screenshot displays the IDA View-PC interface. The main window shows pseudocode for a function. A red box highlights the call to `sub_76148EFC(v10, "classes.dex", 0);`. To the right, the 'General registers' window is visible, showing the value of register R0 as 750DD000, with a comment indicating it points to `data@app@com.warmcar.nf.x_1.apk@classes.dex:750DD000`. A red arrow points from the highlighted call in the pseudocode to the R0 register value.

- 获取dex文件偏移、地址，并且解密dex头部数据到内存中

```

t char *)sub_76046C30)(3, "SecShell", "k 23");

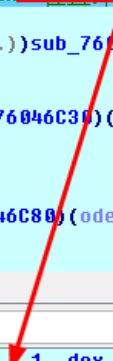
ut_orgDexOffset)(odex_baseAddr + 0x28);// 根据odex文件内的dex_header结构中的data_size和data_off计算出真实Dex文件偏移
..)sub_76046C30)(3, "SecShell", "orgDexOffset:%d", orgDexOffset);// 偏移值为:278528
d int))j_j_memset_0>(&v128, 0, 224);
fset + 0x28;
nt))memcpy>(&v128, DEX_VirtualAddr, 224);// 将DEX文件前224字节内容写入内存
d int, signed int))decrypt(&unk_7604F294, &v128, 224, 32);
// DEX头偏移0x28位置即文件大小

const char *, ...))sub_76046C30)(3, "SecShell", "FileSize:%d", v130);

nst char *)sub_76046C30)(3, "SecShell", "k 24");

ned int))unk_76046C80)(odex_baseAddr, v31, 3);

```



```

65 78 0A .....1..dex.
A6 F2 C8 035.r.....z&...
CB 8B 00 .....g.u.#.....
00 00 00 p...xU4.....
29 00 00 0.....p.....
AF 00 00 .....8..h?.....

```

- 根据解密后的DEX头部0xE0字节数据+DEX偏移指向的剩余部分数据，结合起来就是原始DEX文件

dump解密后的头部0xE0字节数据

```

static main(void)
{
    auto fp, begin, end, ptr;
    fp = fopen("d:\\header.dex", "wb");
    begin = 0x74fd7000;
    len = 0xE0;
    for ( ptr = begin; ptr < begin+len; ptr ++ )
        fputc(Byte(ptr), fp);
}

```

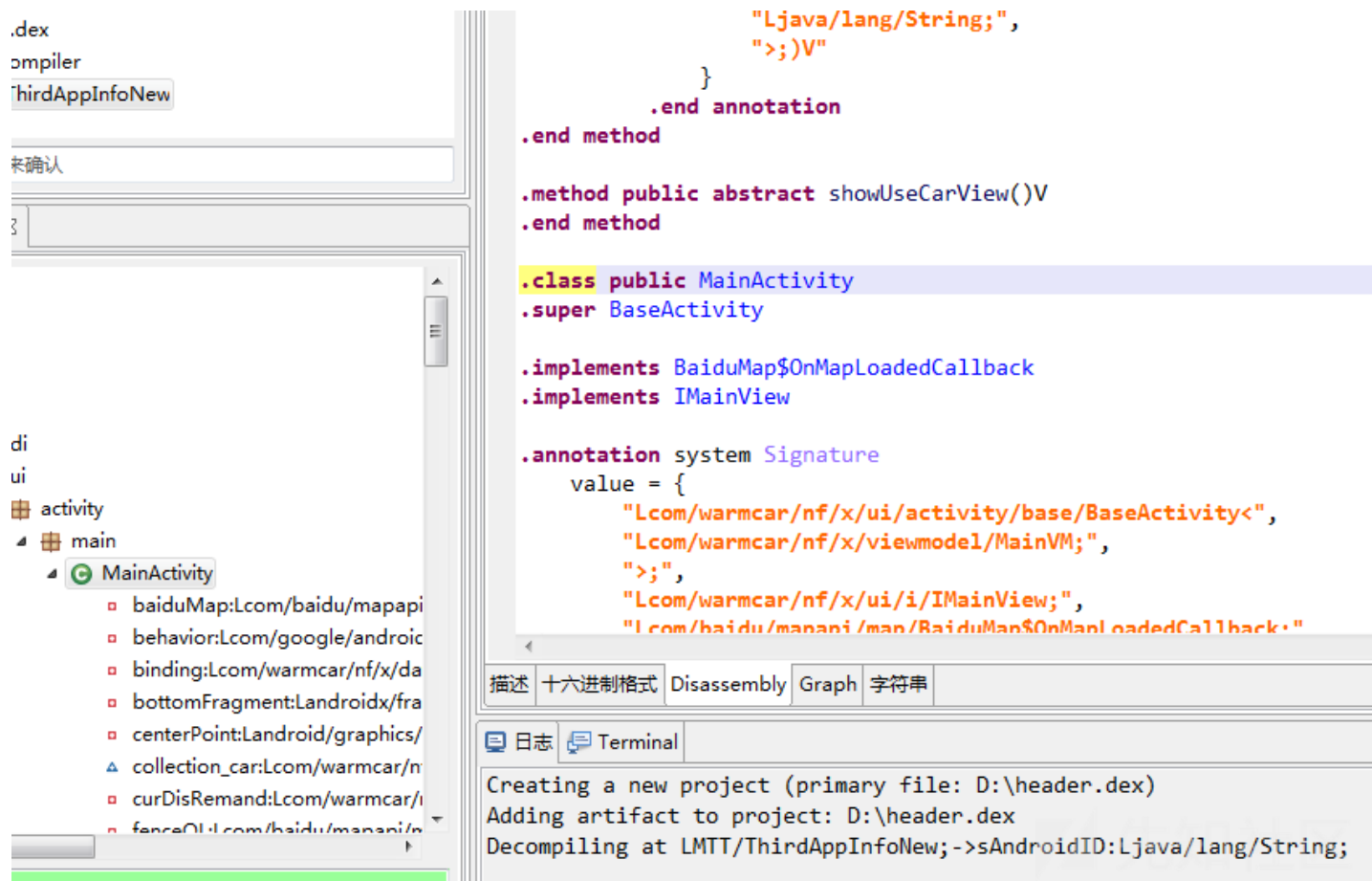
ida脚本打印ODEX文件在内存中的所有数据

```

static main(void)
{
    auto fp, begin, end, ptr;
    fp = fopen("d:\\dump.odex", "wb");
    begin = 0x74fd7000;
    end = 0x75b2f000;
    for ( ptr = begin; ptr < end; ptr ++ )
        fputc(Byte(ptr), fp);
}

```

- dump出Dex header和整个ODEX文件数据后，然后根据Dex Header中的file\_size字段dump出Dex文件，接着用正确的Dex Header头替换错误的头部即可(010edit: ctrl+shift+a, 使用select Range即可)



## 脱壳思路

搜索DEX文件的magic字符64 65 78 0a 30 33 35，截取前0xE0长度的字符并dump到classes.dex本地文件中。获取偏移0x20处的文件大小长度。

接着搜索/proc/<pid>/maps获取odex的内存基址，根据下面计算，得到dex文件偏移地址。a1+0x6C=data\_off，a1+0x68=data\_size</pid>

1. dex偏移 + ODex基址 + 0x28即Dex文件内存地址。结合文件大小dump出dex文件数据，接着去除前0xE0字节数据，将剩余内容写入classes.dex文件中

## 小结

- 【1】IDA在识别节头出错的情况下，会去识别程序头继续分析
- 【2】ELF基础：ELF节头表不能被装载进内存。由于ELF程序装载过程中只用到了程序头表
- 【3】#define HIDWORD(l) (((DWORD)((DWORDLONG)(l) >> 32) & 0xFFFFFFFF))
- 【4】Alt+S：修改段属性，将需要保存的段内存勾上loader选项，TakeMemorySnapshot(1)；：IDC语句，直接打下内存快照
- 【5】0x28为odex文件格式中dex\_header的相对偏移地址，所以(odexAddr + 0x28)为该odex文件格式中dex header的绝对地址

总的来说，是一次马马虎虎的脱壳路程，但是从结果看还是成功的。中途出现很多问题，耐心是必须的。不足也是很多的：

- JNI本地方法注册调用逻辑不熟悉，过程中的很多地方是参考其他文章学习到的。
- 伪代码也不是完全看懂了，很多代码细节是模糊的

## 参考

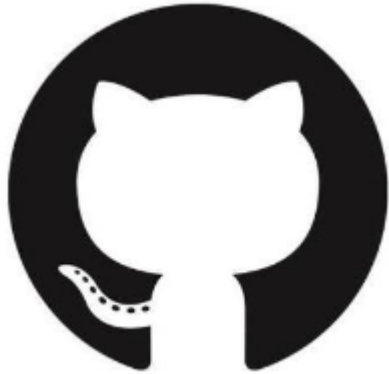
- 【1】国内众多加固厂商存在有各自标志性的加固文件分析的时候可以快速识别
- 【2】[ELF的dump及修复思路](#)
- 【3】section开源修复代码 <https://github.com/WangYinuo/FixElfSection>
- 【4】[乐固壳分析] <https://www.cnblogs.com/goodhacker/p/8666217.html>
- 【5】[原创]乐固libshella 2.10.1分析笔记 <https://bbs.pediy.com/thread-218782.htm>
- 【6】Dalvik虚拟机JNI方法的注册过程分析 <https://blog.csdn.net/Luoshengyang/article/details/8923483>

- 【7】乐固2.8 <https://my.oschina.net/jalen1991/blog/1870774>
- 【8】Fatal signal 11问题的解决方法 <https://blog.csdn.net/tankai19880619/article/details/9004619>
- 【9】Android 「动态分析」 打开调试开关的三种方法 <https://blog.csdn.net/hp910315/article/details/82769506>
- 【10】手动绕过百度加固Debug.isDebuggerConnected反调试的方法 <https://blog.csdn.net/QQ1084283172/article/details/78237571>

点击收藏 | 0 关注 | 1

[上一篇：2018铁人三项赛总决赛PWN](#) [下一篇：Redis 4.x RCE分析](#)

1. 4 条回复



[chybeta](#) 2019-07-10 09:12:23

炒鸡详细，感谢分享

0 回复Ta



[1590307279601171](#) 2019-07-16 10:04:52

《国内众多加固厂商存在有各自标志性的加固文件分析的时候可以快速识别》这篇文章为什么没有连接呢？

0 回复Ta



[星落\\_z](#) 2019-08-07 17:31:24

感谢大佬，我想问一下脱壳了之后怎样得知原始APP的入口Application啊？  
脱壳了之后不知道如何反编译  
希望大佬给一些思路

0 回复Ta



[yong夜](#) 2019-09-06 09:44:50

@[星落\\_z](#) 原始App的Application的name应该有俩种查看方法：

- 1.壳加载原始Application的时候会将原始application的name值赋给mBoundApplication里appInfo的className字段，可以跟一下
- 2.就是用JEB或者Jadx直接打开DEX文件，看继承自Application的类名，一般只有一个吧，就是原始DEX的application了

脱壳之后直接把DEX拖进工具即可

0 回复Ta

---

[登录](#) 后跟帖

先知社区

---

[现在登录](#)

热门节点

---

[技术文章](#)

[社区小黑板](#)

目录

[RSS](#) [关于社区](#) [友情链接](#) [社区小黑板](#)