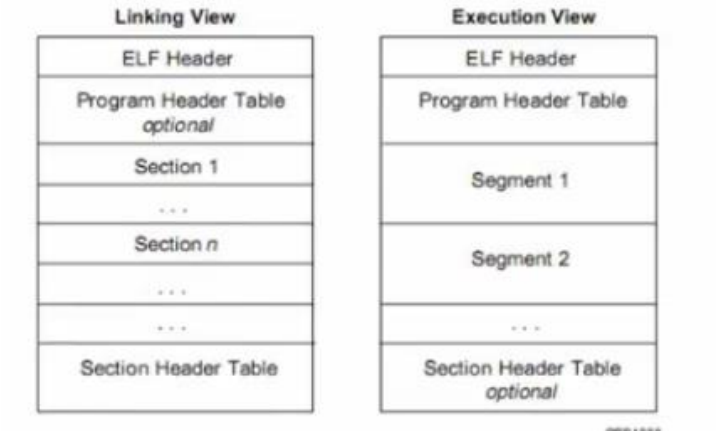
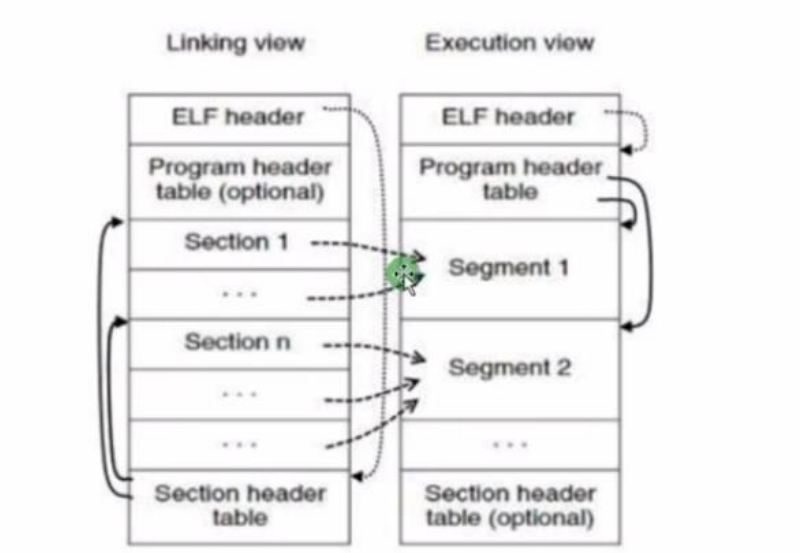
1. **来自于 Linux 的 elf**

   Android 上的 so(elf)与 Linux 上的有 99%的相似，其中的知识点和结构是可以互通的。

2. **Elf 的链接视图**

   Elf 在加载前和加载后的结构是不一样的，这就给加密提供了方便。





   链接执行的时候，shdr 中的表将会被映射到 phdr 中，里面有 3 个头非常重要 elf_header，program header，section header，linker 将根据这三个头信息进行 so 文件加载

3. **Elf header**

   存储 so 文件最基本的信息，比如 so 运行 cpu 平台，program header 数量，section header 数量等，重要性等同于 dex header

4. **Elf Section header**

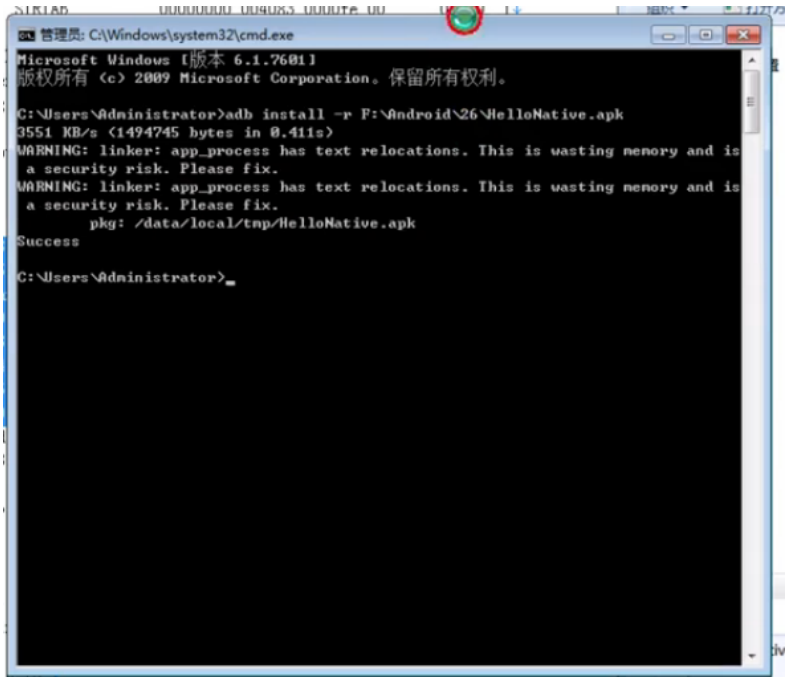
   存储 so 文件的链接用信息，主要是用于给外部程序提供详细的本 so 文件的信息，比如第几行对应哪个函数，什么名字，对应的源码位置等等。Ida 则是通过读取该头信息进行 so 文件分析的。

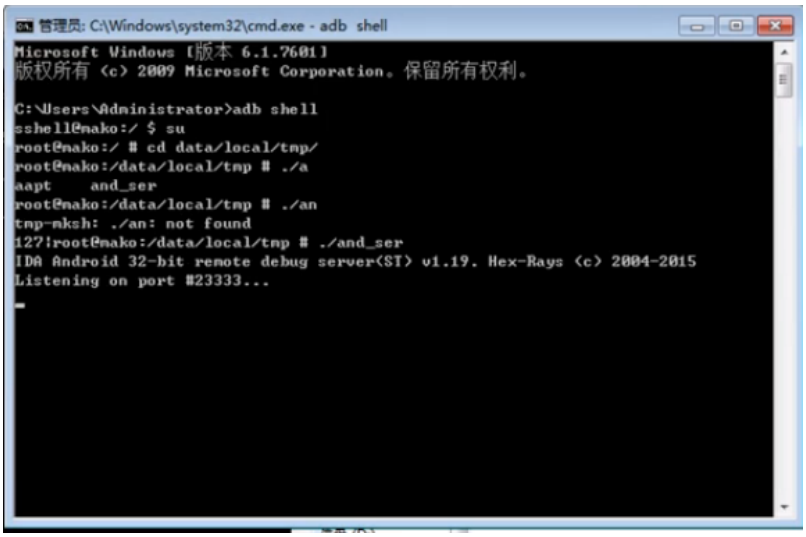
5. **Elf Program header**

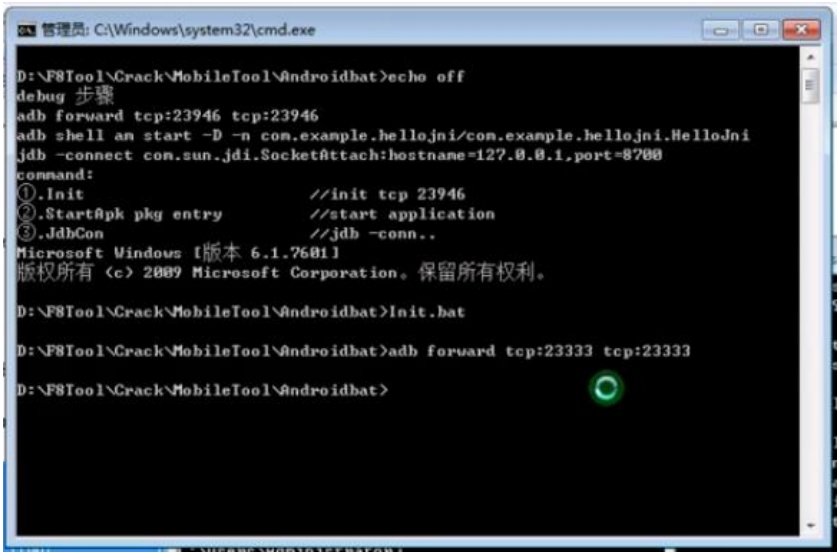
   存储 so 文件运行时需要的信息，该信息会直接被 linker 所使用，运用于 so 文件的加载过程中。因此 header 的数据是肯定可信的。

6. **从内存中 dump 出 so 文件**

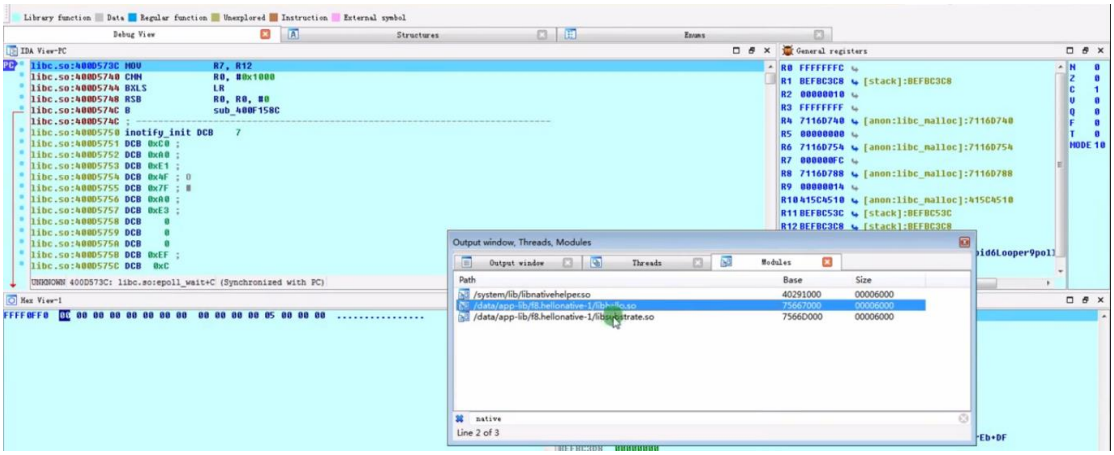
   对目标程序进行安装启动和端口转发


管理员: C:\Windows\system32\cmd.exe
Microsoft Windows [版本 6.1.7601]
版权所有 (c) 2009 Microsoft Corporation。保留所有权利。
C:\Users\Administrator>adb install -r F:\Android\26\HelloNative.apk
3551 KB/s (1494745 bytes in 0.411s)
WARNING: linker: app_process has text relocations. This is wasting memory and is a security risk. Please fix.
WARNING: linker: app_process has text relocations. This is wasting memory and is a security risk. Please fix.
pkg: /data/local/tmp/HelloNative.apk
Success
C:\Users\Administrator>

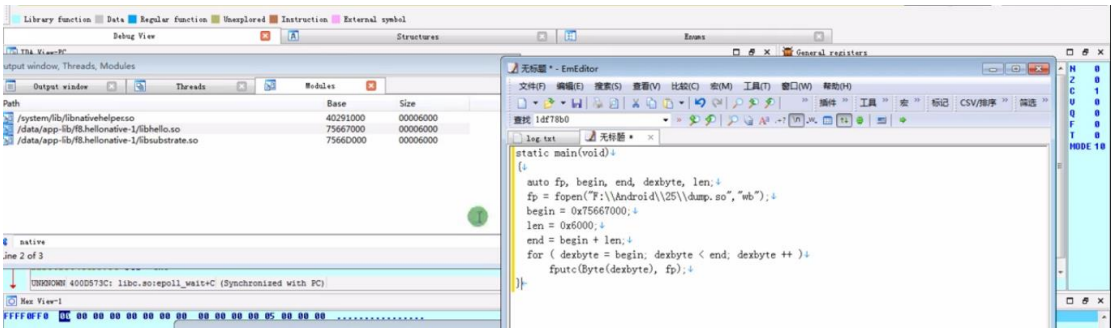

管理员: C:\Windows\system32\cmd.exe - adb shell
Microsoft Windows [版本 6.1.7601]
版权所有 (c) 2009 Microsoft Corporation。保留所有权利。
C:\Users\Administrator>adb shell
sshell@nako:/ $ su
root@nako:/ # cd data/local/tmp/
root@nako:/data/local/tmp # ./a
aapt and_ser
root@nako:/data/local/tmp # ./an
tmp-mksh: ./an: not found
127!root@nako:/data/local/tmp # ./and_ser
IDA Android 32-bit remote debug server(ST) v1.19. Hex-Rays (c) 2004-2015
Listening on port #23333...

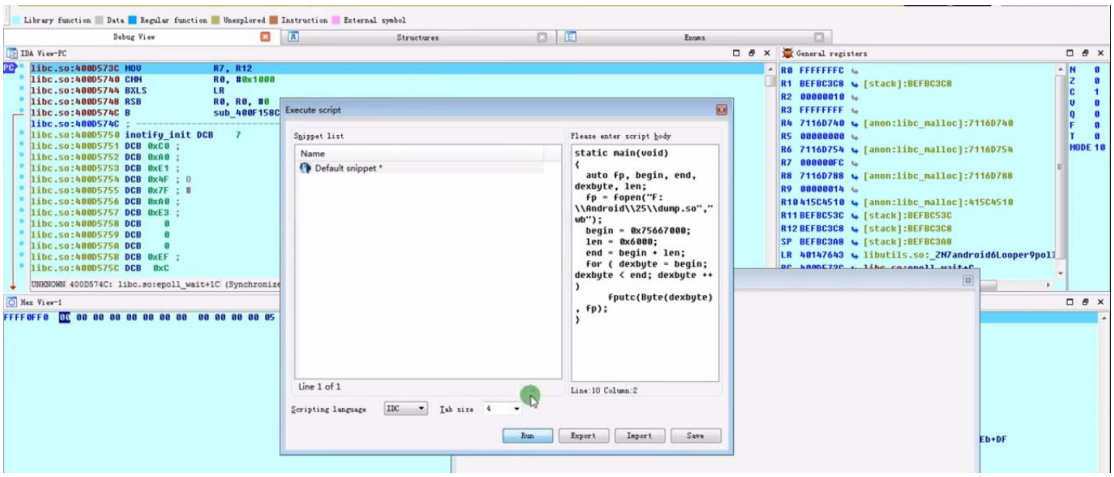

管理员: C:\Windows\system32\cmd.exe
D:\F8Tool\Crack\MobileTool\Androidbat>echo off
debug 步骤
adb forward tcp:23946 tcp:23946
adb shell am start -D -n com.example.hellojni/com.example.hellojni.HelloJni
jdb -connect com.sun.jdi.SocketAttach:hostname=127.0.0.1,port=8700
command:
①.Init //init tcp 23946
②.StartApk pkg entry //start application
③.JdbCon //jdb -conn..
Microsoft Windows [版本 6.1.7601]
版权所有 (c) 2009 Microsoft Corporation。保留所有权利。
D:\F8Tool\Crack\MobileTool\Androidbat>Init.bat
D:\F8Tool\Crack\MobileTool\Androidbat>adb forward tcp:23333 tcp:23333
D:\F8Tool\Crack\MobileTool\Androidbat>

使用 ida 打开目标程序进行挂接



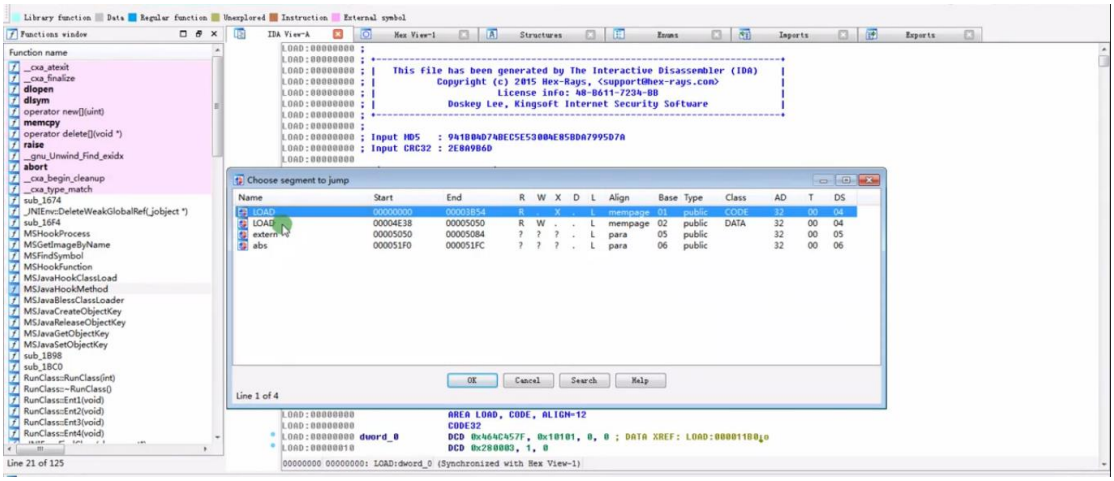
定位到目标 so 文件，得到该文件的起始地址和文件大小，通过脚本程序从内存中 dump 出来

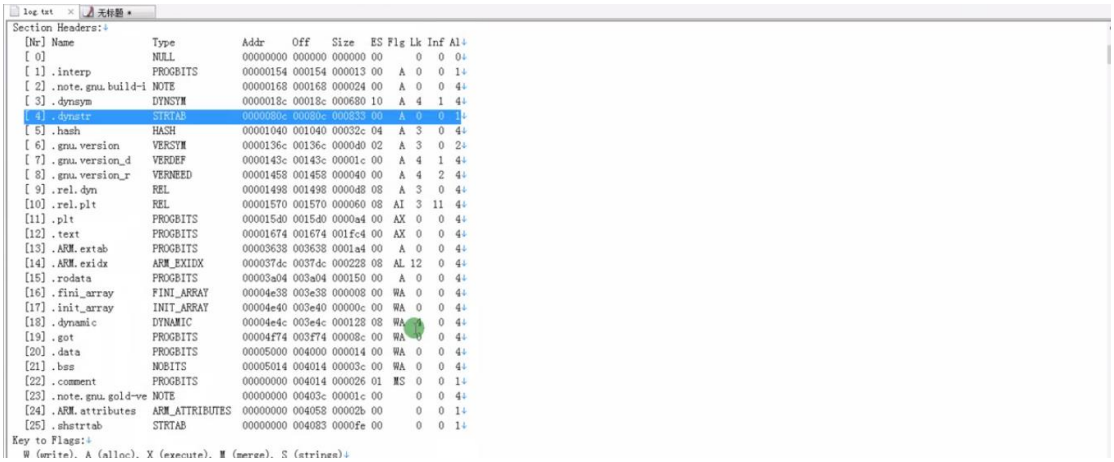




dump 出来的 so 文件由于加载时候进行了内存对齐，所以可能会大一些

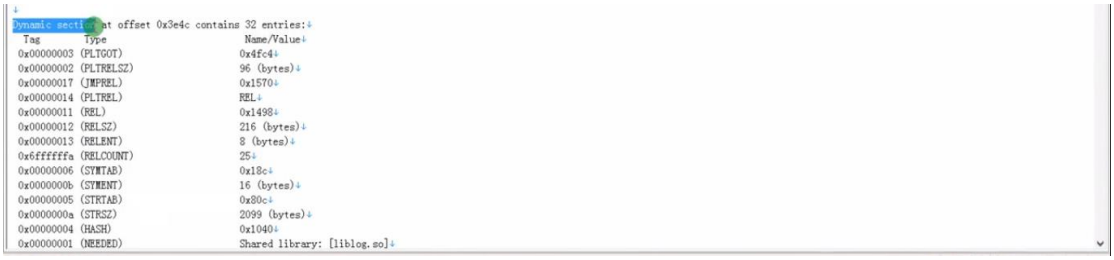
内存中所加载的段如下图





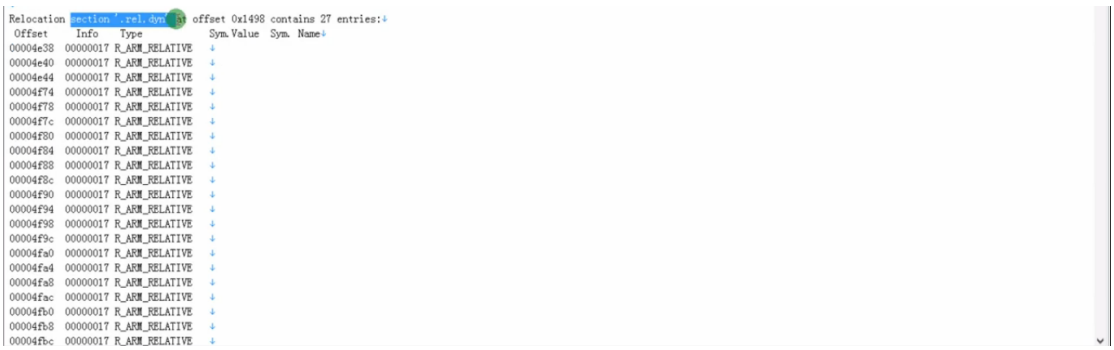
```
Section Headers:
  [Nr] Name              Type            Addr     Off    Size   ES Flg Lk Inf Al
  [ 0]                   NULL            00000000 000000 000000 00      0   0  0
  [ 1] .interp           PROGBITS        00000154 000154 000013 00   A  0   0  1
  [ 2] .note.gnu.build-i NOTE            00000168 000168 000024 00   A  0   0  4
  [ 3] .dynsym           DYNSYM          0000018c 00018c 000680 10   A  4   1  4
  [ 4] .dynstr           STRTAB          0000080c 00080c 000833 00   A  0   0  1
  [ 5] .hash             HASH            00001040 001040 00032c 04   A  3   0  4
  [ 6] .gnu.version      VERSYM          0000136c 00136c 000040 02   A  3   0  2
  [ 7] .gnu.version_d    VERDEF          0000143c 00143c 00001c 00   A  4   1  4
  [ 8] .gnu.version_r    VERNEED         00001458 001458 000040 00   A  4   2  4
  [ 9] .rel.dyn          REL             00001498 001498 0000d8 08   A  3   0  4
  [10] .rel.plt          REL             00001570 001570 000060 08  AI  3  11  4
  [11] .plt              PROGBITS        000015d0 0015d0 0000a4 00  AX  0   0  4
  [12] .text             PROGBITS        00001674 001674 001fc4 00  AX  0   0  4
  [13] .ARM.extab        PROGBITS        00003638 003638 0001a4 00   A  0   0  4
  [14] .ARM.exidx        ARM_EXIDX       000037dc 0037dc 000228 08  AL 12   0  4
  [15] .rodata           PROGBITS        00003a04 003a04 000150 00   A  0   0  4
  [16] .fini_array       FINI_ARRAY      00004e38 003e38 000008 00  WA  0   0  4
  [17] .init_array       INIT_ARRAY      00004e40 003e40 00000c 00  WA  0   0  4
  [18] .dynamic          DYNAMIC         00004e4c 003e4c 000128 08  WA  4   0  4
  [19] .got              PROGBITS        00004f74 003f74 00008c 00  WA  0   0  4
  [20] .data             PROGBITS        00005000 004000 000014 00  WA  0   0  4
  [21] .bss              NOBITS          00005014 004014 00003c 00  WA  0   0  4
  [22] .comment          PROGBITS        00000000 004014 000026 01  MS  0   0  1
  [23] .note.gnu.gold-ve NOTE            00000000 00403c 00001c 00      0   0  4
  [24] .ARM.attributes   ARM_ATTRIBUTES  00000000 004058 00002b 00      0   0  1
  [25] .shstrtab         STRTAB          00000000 004083 0000fe 00      0   0  1
Key to Flags:
  W (write), A (alloc), X (execute), M (merge), S (strings)
```

通过分析.dynstr 的 section 表中数据可以找到想要的函数名和某些加密方法



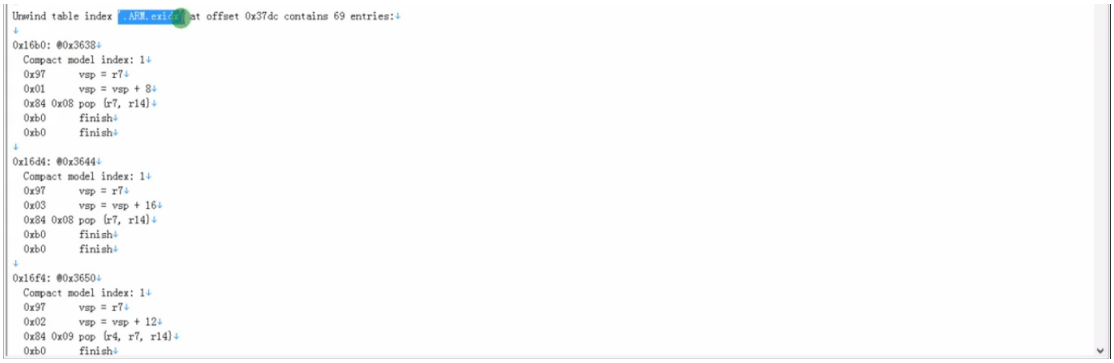
```
Dynamic section at offset 0x3e4c contains 32 entries:
  Tag        Type                         Name/Value
 0x00000003 (PLTGOT)                     0x4fc4
 0x00000002 (PLTRELSZ)                   96 (bytes)
 0x00000017 (JMPREL)                     0x1570
 0x00000014 (PLTREL)                     REL
 0x00000011 (REL)                        0x1498
 0x00000012 (RELSZ)                      216 (bytes)
 0x00000013 (RELENT)                     8 (bytes)
 0x6ffffffa (RELCOUNT)                   25
 0x00000006 (SYMTAB)                     0x18c
 0x0000000b (SYMENT)                     16 (bytes)
 0x00000005 (STRTAB)                     0x80c
 0x0000000a (STRSZ)                      2099 (bytes)
 0x00000004 (HASH)                       0x1040
 0x00000001 (NEEDED)                     Shared library: [liblog.so]
```

Dynamic section 中存放着函数和所依赖的 so 文件



```
Relocation section '.rel.dyn' at offset 0x1498 contains 27 entries:
 Offset     Info    Type            Sym.Value  Sym. Name
00004e38  00000017 R_ARM_RELATIVE
00004e40  00000017 R_ARM_RELATIVE
00004e44  00000017 R_ARM_RELATIVE
00004f74  00000017 R_ARM_RELATIVE
00004f78  00000017 R_ARM_RELATIVE
00004f7c  00000017 R_ARM_RELATIVE
00004f80  00000017 R_ARM_RELATIVE
00004f84  00000017 R_ARM_RELATIVE
00004f88  00000017 R_ARM_RELATIVE
00004f8c  00000017 R_ARM_RELATIVE
00004f90  00000017 R_ARM_RELATIVE
00004f94  00000017 R_ARM_RELATIVE
00004f98  00000017 R_ARM_RELATIVE
00004f9c  00000017 R_ARM_RELATIVE
00004fa0  00000017 R_ARM_RELATIVE
00004fa4  00000017 R_ARM_RELATIVE
00004fa8  00000017 R_ARM_RELATIVE
00004fac  00000017 R_ARM_RELATIVE
00004fb0  00000017 R_ARM_RELATIVE
00004fb8  00000017 R_ARM_RELATIVE
00004fbc  00000017 R_ARM_RELATIVE
```

Relocation 中存放重定位数据



```
Unwind table index .ARM.exidx at offset 0x37dc contains 69 entries:

0x16b0: @0x3638
  Compact model index: 1
  0x97      vsp = r7
  0x01      vsp = vsp + 8
  0x84 0x08 pop {r7, r14}
  0xb0      finish
  0xb0      finish

0x16d4: @0x3644
  Compact model index: 1
  0x97      vsp = r7
  0x03      vsp = vsp + 16
  0x84 0x08 pop {r7, r14}
  0xb0      finish
  0xb0      finish

0x16f4: @0x3650
  Compact model index: 1
  0x97      vsp = r7
  0x02      vsp = vsp + 12
  0x84 0x09 pop {r4, r7, r14}
  0xb0      finish
```

ARM 文件中特有的表，存放着一些函数的中转表

```
Symbol table '.dynsym' contains 104 entries:
   Num:    Value  Size Type    Bind   Vis      Ndx Name
     0: 00000000     0 NOTYPE  LOCAL  DEFAULT  UND
     1: 00000000     0 FUNC    GLOBAL DEFAULT  UND __cxa_finalize@LIBC (2)
     2: 00000000     0 FUNC    GLOBAL DEFAULT  UND __cxa_atexit@LIBC (2)
     3: 000016b1    36 FUNC    WEAK   DEFAULT   12 _ZN7_JNIEnv19DeleteWeakGl
     4: 00002c24     8 FUNC    WEAK   DEFAULT   12 __aeabi_unwind_cpp_pr1
     5: 00005014     4 OBJECT  GLOBAL DEFAULT   21 imageBase
     6: 00005018     4 OBJECT  GLOBAL DEFAULT   21 fmsHookProcess
     7: 0000501c     4 OBJECT  GLOBAL DEFAULT   21 fmsGetImageByName
     8: 00005020     4 OBJECT  GLOBAL DEFAULT   21 fmsFindSymbol
     9: 00005024     4 OBJECT  GLOBAL DEFAULT   21 fmsHookFunction
    10: 00005028     4 OBJECT  GLOBAL DEFAULT   21 fmsJavaHookClassLoad
    11: 0000502c     4 OBJECT  GLOBAL DEFAULT   21 fmsJavaHookMethod
    12: 00005030     4 OBJECT  GLOBAL DEFAULT   21 fmsJavaEnableClassLoader
    13: 00005034     4 OBJECT  GLOBAL DEFAULT   21 fmsJavaCreateObjectKey
    14: 00005038     4 OBJECT  GLOBAL DEFAULT   21 fmsJavaReleaseObjectKey
    15: 0000503c     4 OBJECT  GLOBAL DEFAULT   21 fmsJavaGetObjectKey
    16: 00005040     4 OBJECT  GLOBAL DEFAULT   21 fmsJavaSetObjectKey
    17: 00000000     0 FUNC    GLOBAL DEFAULT  UND dlopen@LIBC (3)
    18: 00000000     0 FUNC    GLOBAL DEFAULT  UND dlsym@LIBC (3)
    19: 000018f5    68 FUNC    GLOBAL DEFAULT   12 MSHookProcess
    20: 00001939    56 FUNC    GLOBAL DEFAULT   12 MSGetImageByName
    21: 00001971    64 FUNC    GLOBAL DEFAULT   12 MSFindSymbol
    22: 000019b1    60 FUNC    GLOBAL DEFAULT   12 MSHookFunction
```

dynsym 中能够找到需要导入的所有函数的名字、起始地址、大小和访问权限等

7. **Others**

Android DVM 脱壳 1

1. Apk dex 加固

目前存在对 apk 中的 classes.dex 进行加密的技术，称为加壳。通过对 dex 文件的加壳，可以达到减少体积(?)，隐藏真实代码的效果。而这一类的技术，也把很多新手挡在了逆向的门外。

2. 运行时解密

与 Windows 上的 Shell 一样，在程序运行时，先到达壳的入口点，运行解壳代码，然后再到达程序入口点，运行代码。

而要脱壳，则是要在程序解码完毕，到达程序真实入口点中间某个位置，把原始的 dex 代码给 dump 出来，还原到 apk 文件中。

3. 一个简单的壳示例

入口点改变：

2. 运行时解密

与 Windows 上的 Shell 一样，在程序运行时，先到达壳的入口点，运行解壳代码，然后再到达程序入口点，运行代码。

而要脱壳，则是要在程序解码完毕，到达程序真实入口点中间某个位置，把原始的 dex 代码给 dump 出来，还原到 apk 文件中。

3. 一个简单的壳示例

入口点改变：

壳入口：

```
<application android:name="com.ali.mobisecenhance.StubApplication" />
```

程序入口：

```
<activity android:name="com.ali.encryption.MainActivity"/>
```

壳代码中 Java 层转 jni 层

```
protected native void attachBaseContext(Context arg1) {
}

public native void onCreate() {
}
```

3. 一个简单的壳示例

入口点改变：

壳入口：

```
<application android:name="com.ali.mobisecenhance.StubApplication" />
```

程序入口：

```
<activity android:name="com.ali.encryption.MainActivity"/>
```

壳代码中 Java 层转 jni 层

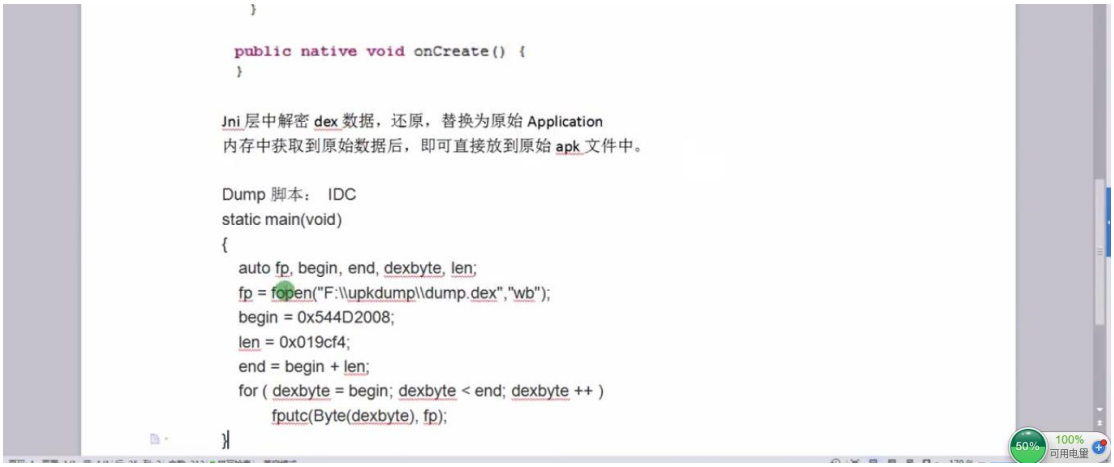
```
protected native void attachBaseContext(Context arg1) {
}

public native void onCreate() {
}
```

Jni 层中解密 dex 数据，还原，替换为原始 Application

内存中获取到原始数据后，即可直接放到原始 apk 文件中。

Dump 脚本： IDC

```
static main(void)
{
  auto fp, begin, end, dexbyte, len;
```



```
}

public native void onCreate() {
}
```

Jni 层中解密 dex 数据，还原，替换为原始 Application

内存中获取到原始数据后，即可直接放到原始 apk 文件中。

Dump 脚本： IDC

```
static main(void)
{
  auto fp, begin, end, dexbyte, len;
  fp = fopen("F:\\upkdump\\dump.dex","wb");
  begin = 0x544D2008;
  len = 0x019cf4;
  end = begin + len;
  for ( dexbyte = begin; dexbyte < end; dexbyte ++ )
      fputc(Byte(dexbyte), fp);
}
```

8. **Android linker**

Android 上的 elf 文件是通过 linker 加载到内存中并进行执行的。所以通过研究 linker 可以清楚地知道 Android 系统到底使用了 so 文件中的哪几个数据，linker 启动的时候会首先对自身的函数表数据等进行重定位，然后再对其他 so 文件进行定位。

Linker 源代码位置：Bionic/Linker

9. **Program header 和 Section header**

Linker 加载中只会用到 Program header，而 Section header 基本上不会用到，甚至直接删

除 Section header 也是可以的。

10. Program header 解析

So 加载：linker.cpp --> soinfo* do_dlopen(const char* name, int flags);

find_library(name);　　//加载

si --> CallConstructors();　　//调用初始化函数

CallFunction(“DT_INIT”, init_func); --> so 脱壳点

CallArray(“DT_INIT_ARRAY”, init_array, init_array_count, false); --> dex 脱壳点