深入Androguard源码——how to 读取dex

2016-10-17 penguin_wwy 京东安全应急响应中心

点击上方蓝字关注

0X01 概述

Androguard是一款出色的开源静态分析apk工具，功能强大。比如以下这些功能模块：
androgexf.py 生成函数调用图；
apkviewer.py 生成指令级别的调用图；
androlyze.py 交互分析环境；
androdiff.py 比较两个APK间的差异。
此外，Androguard不单单功能强大，还很适合进行扩展或者移植成为自己项目的某一模块。

0X02 源码分析

本次看的源码是dex文件读取解析的过程，重点在如何解析保存文件结构。
一般情况下，输入会有如下的逻辑：

```
if ret_type == "APK":
    x = apk.APK( i )
    bc = dvm.DalvikVMFormat( x.get_dex() )
elif ret_type == "DEX":
    bc = dvm.DalvikVMFormat( read(i) )
```

重点看实例化DalvikVMFormat的过程：

```
def __init__(self, buff, decompiler=None, config=None, using_api=None):
    #to allow to pass apk object ==> we do not need to pass additionally target version
    if isinstance(buff, APK):
        self.api_version = buff.get_target_sdk_version()
        buff = buff.get_dex() #如果传入的是一个APK，则从APK中获得dex文件
    elif using_api:
        self.api_version = using_api
    else:
        self.api_version = CONF["DEFAULT_API"]

    #TODO: can using_api be added to config parameter?
    super(DalvikVMFormat, self).__init__(buff)

    self.config = config    #初始化配置信息
    if not self.config:
      self.config = {"RECODE_ASCII_STRING": CONF["RECODE_ASCII_STRING"],
                     "RECODE_ASCII_STRING_METH": CONF["RECODE_ASCII_STRING_METH"],
                     "LAZY_ANALYSIS": CONF["LAZY_ANALYSIS"]}

    self.CM = ClassManager(self, self.config)   #实例化一个ClassManager
    self.CM.set_decompiler(decompiler)          #默认反编译器ded

    self._preload(buff)
    self._load(buff)        #读取dex文件内容
```

先看一下ClassManager的实例化过程，第一个参数为self。找到ClassManager的构造函数：

```
def __init__(self, vm, config):
    self.vm = vm
    self.buff = vm
```

也就是说一个DalvikVMFormat对象实例的CM成员的vm成员和buff成员就是这个DalvikVMFormat对象实例它自己。

这句话读起来有点晕，先不管它，往下看，之后就会知道为神马会有这句绕口令。

Androguard在处理的时候会使用反编译器，有些功能会将反编译后的.java文件传出。

默认情况下使用ded反编译，有些情况下可以由使用者指定。之后的一句self._preload()是个空函数。

前面这些都只是小菜，真正的干货在self._load()函数中。正常解析一个文件第一步先读文件头：

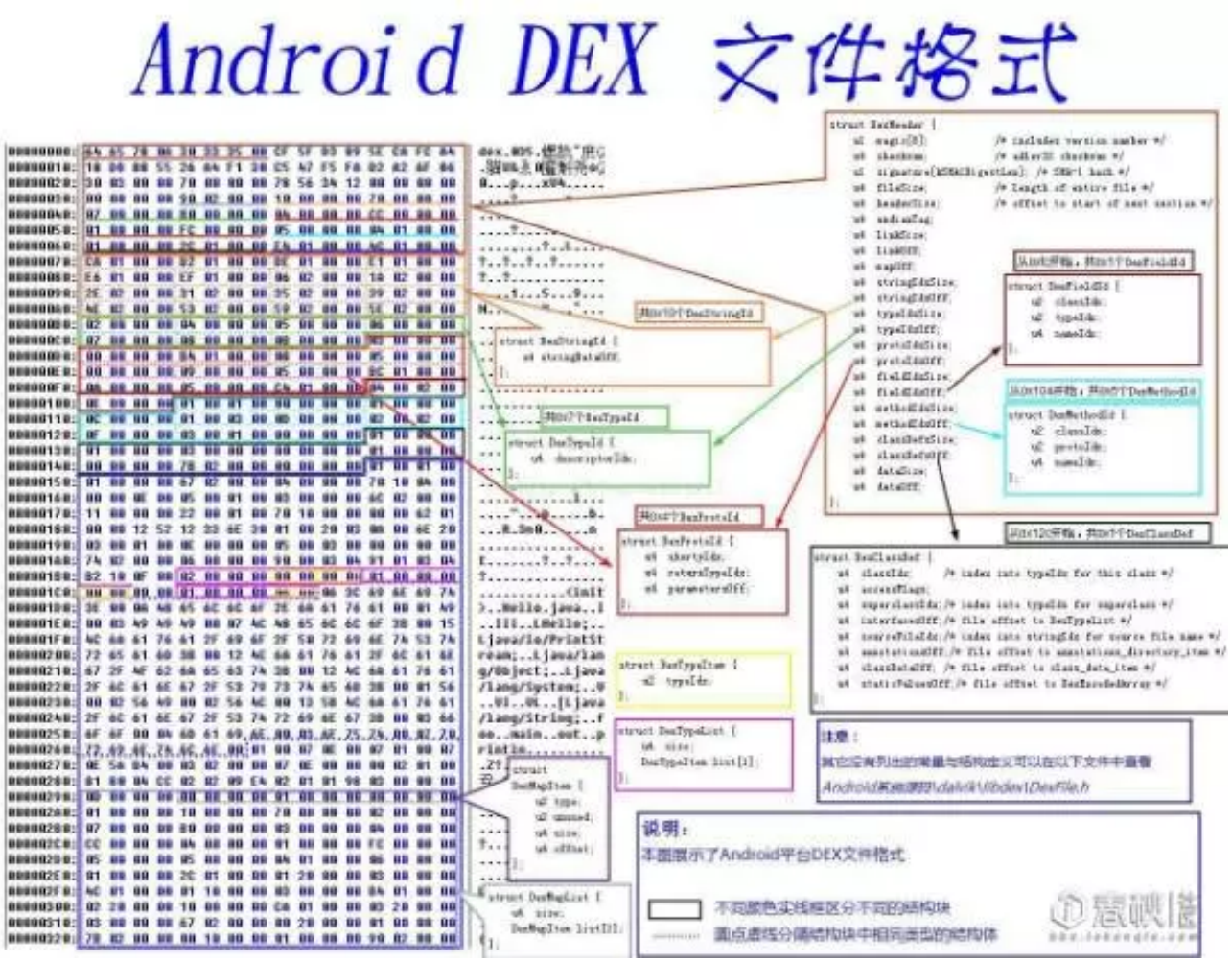
```
self.__header = HeaderItem(0, self, ClassManager(None, self.config))    #解析dex文件头
```

HeaderItem是文件头解析类，构造函数如下：

```python
def __init__(self, size, buff, cm):
    self.__CM = cm

    self.offset = buff.get_idx()

    self.magic = unpack("=Q", buff.read(8))[0]              #魔数
    self.checksum = unpack("=i", buff.read(4))[0]           #文件校验码
    self.signature = unpack("=20s", buff.read(20))[0]       #SHA-1
    self.file_size = unpack("=I", buff.read(4))[0]          #文件大小
    self.header_size = unpack("=I", buff.read(4))[0]        #文件头大小
    self.endian_tag = unpack("=I", buff.read(4))[0]         #大小端标签
    self.link_size = unpack("=I", buff.read(4))[0]          #链接数据大小
    self.link_off = unpack("=I", buff.read(4))[0]           #链接数据偏移
    self.map_off = unpack("=I", buff.read(4))[0]            #map item 的偏移地址
    self.string_ids_size = unpack("=I", buff.read(4))[0]    #字符串池大小
    self.string_ids_off = unpack("=I", buff.read(4))[0]     #字符串池偏移
    self.type_ids_size = unpack("=I", buff.read(4))[0]      #类型数据结构大小
    self.type_ids_off = unpack("=I", buff.read(4))[0]       #类型数据结构偏移
    self.proto_ids_size = unpack("=I", buff.read(4))[0]     #元数据信息数据结构大小
    self.proto_ids_off = unpack("=I", buff.read(4))[0]      #元数据信息数据结构偏移
    self.field_ids_size = unpack("=I", buff.read(4))[0]     #字段信息数据结构大小
    self.field_ids_off = unpack("=I", buff.read(4))[0]      #字段信息数据结构偏移
    self.method_ids_size = unpack("=I", buff.read(4))[0]    #方法信息数据结构大小
    self.method_ids_off = unpack("=I", buff.read(4))[0]     #方法信息数据结构偏移
    self.class_defs_size = unpack("=I", buff.read(4))[0]    #类信息数据结构大小
    self.class_defs_off = unpack("=I", buff.read(4))[0]     #类信息数据结构偏移
    self.data_size = unpack("=I", buff.read(4))[0]          #数据区域大小
    self.data_off = unpack("=I", buff.read(4))[0]           #数据区域偏移


    self.map_off_obj = None
    self.string_off_obj = None
    self.type_off_obj = None
    self.proto_off_obj = None
    self.field_off_obj = None
    self.method_off_obj = None
    self.class_off_obj = None
    self.data_off_obj = None
```

是不是感觉很简单！嗯，我也觉得。

下面我们来看一张神图：



文件头的重点在map_off，map_off指向MapList结构，而MapList则是整个dex文件的映射。MapList包含N个MapItem结构，每个MapItem包含四个元素：

1、type 表示该 map_item 的类型；

2、unuse 是用对齐字节的 ，无实际用处；

3、size 表示再细分此 item ， 该类型的个数；

4、offset 是第一个元素的针对文件初始位置的偏移量 。

举个栗子，0x0表示文件头，所以type为0；一个文件只有一个文件头，所以size为1；而文件头从整个文件偏移为0的位置开始，所以offset为0。这样一个MapItem就是文件头的MapItem，理解了这些我们就可以继续看代码了。

```
if self.__header.map_off == 0:
    bytecode.Warning("no map list ...")
else:
    self.map_list = MapList( self.CM, self.__header.map_off, self )
```

如果map_off存在，就实例化MapList，进行处理。

```
def __init__(self, cm, off, buff):
    self.CM = cm

    buff.set_idx( off )

    self.offset = off

    self.size = unpack("=I", buff.read( 4 ) )[0]

    self.map_item = []
    for i in xrange(0, self.size):
        idx = buff.get_idx()    #获取每个item的偏移

        mi = MapItem( buff, self.CM )   #实例化MapItem
        self.map_item.append( mi )      #每个实例加入map_item

        buff.set_idx( idx + mi.get_length() )

        c_item = mi.get_item()
        if c_item == None:
          mi.set_item( self )
          c_item = mi.get_item()

```

当获得下一个要处理的item在文件中的偏移之后就实例化MapItem对它进行处理。

```python
class MapItem(object):
    def __init__(self, buff, cm):
        self.__CM = cm

        self.off = buff.get_idx()   #获取偏移
        #读取元素
        self.type = unpack("=H", buff.read(2))[0]
        self.unused = unpack("=H", buff.read(2))[0]
        self.size = unpack("=I", buff.read(4))[0]
        self.offset = unpack("=I", buff.read(4))[0]

        self.item = None

        buff.set_idx( self.offset ) #设置偏移

        lazy_analysis = self.__CM.get_lazy_analysis()

        if lazy_analysis:   #懒人模式
          self.next_lazy(buff, cm)
        else:
          self.next(buff, cm)
```

在读取完元素，获取到偏移之后就要对这个item映射的内容，也就是偏移指向的内容进行处理。这里可以选择懒人模式，不过大家都是勤奋的好青年，所以我们看勤快模式，也就是self.next(buff, cm)。之前说每个item都有type，那一共有多少种type呢。

```python
TYPE_MAP_ITEM = {
                        0x0:    "TYPE_HEADER_ITEM",
                        0x1:    "TYPE_STRING_ID_ITEM",
                        0x2:    "TYPE_TYPE_ID_ITEM",
                        0x3:    "TYPE_PROTO_ID_ITEM",
                        0x4:    "TYPE_FIELD_ID_ITEM",
                        0x5:    "TYPE_METHOD_ID_ITEM",
                        0x6:    "TYPE_CLASS_DEF_ITEM",
                        0x1000:  "TYPE_MAP_LIST",
                        0x1001:  "TYPE_TYPE_LIST",
                        0x1002:  "TYPE_ANNOTATION_SET_REF_LIST",
                        0x1003:  "TYPE_ANNOTATION_SET_ITEM",
                        0x2000:  "TYPE_CLASS_DATA_ITEM",
                        0x2001:  "TYPE_CODE_ITEM",
                        0x2002:  "TYPE_STRING_DATA_ITEM",
                        0x2003:  "TYPE_DEBUG_INFO_ITEM",
                        0x2004:  "TYPE_ANNOTATION_ITEM",
                        0x2005:  "TYPE_ENCODED_ARRAY_ITEM",
                        0x2006:  "TYPE_ANNOTATIONS_DIRECTORY_ITEM",
                    }
```

比如TYPE_HEADER_ITEM就是指文件头，TYPE_STRING_ID_ITEM就是字符串id等等。
有了这个dict，我们看next函数的内容。

```
def next(self, buff, cm):
    if TYPE_MAP_ITEM[ self.type ] == "TYPE_STRING_ID_ITEM":
        self.item = [ StringIdItem( buff, cm ) for i in xrange(0, self.size) ]

    elif TYPE_MAP_ITEM[ self.type ] == "TYPE_CODE_ITEM":
        self.item = CodeItem( self.size, buff, cm )

    elif TYPE_MAP_ITEM[ self.type ] == "TYPE_TYPE_ID_ITEM":
        self.item = TypeHIdItem( self.size, buff, cm )

    elif TYPE_MAP_ITEM[ self.type ] == "TYPE_PROTO_ID_ITEM":
        self.item = ProtoHIdItem( self.size, buff, cm )

    elif TYPE_MAP_ITEM[ self.type ] == "TYPE_FIELD_ID_ITEM":
        self.item = FieldHIdItem( self.size, buff, cm )

    elif TYPE_MAP_ITEM[ self.type ] == "TYPE_METHOD_ID_ITEM":
        self.item = MethodHIdItem( self.size, buff, cm )

    elif TYPE_MAP_ITEM[ self.type ] == "TYPE_CLASS_DEF_ITEM":
        self.item = ClassHDefItem( self.size, buff, cm )

    elif TYPE_MAP_ITEM[ self.type ] == "TYPE_HEADER_ITEM":
        self.item = HeaderItem( self.size, buff, cm )

```

```
26      elif TYPE_MAP_ITEM[ self.type ] == "TYPE_ANNOTATION_ITEM":
27          self.item = [ AnnotationItem( buff, cm ) for i in xrange(0, self.size) ]
28
29      elif TYPE_MAP_ITEM[ self.type ] == "TYPE_ANNOTATION_SET_ITEM":
30          self.item = [ AnnotationSetItem( buff, cm ) for i in xrange(0, self.size) ]
31
32      elif TYPE_MAP_ITEM[ self.type ] == "TYPE_ANNOTATIONS_DIRECTORY_ITEM":
33          self.item = [ AnnotationsDirectoryItem( buff, cm ) for i in xrange(0, self.size) ]
34
35      elif TYPE_MAP_ITEM[ self.type ] == "TYPE_ANNOTATION_SET_REF_LIST":
36          self.item = [ AnnotationSetRefList( buff, cm ) for i in xrange(0, self.size) ]
37
38      elif TYPE_MAP_ITEM[ self.type ] == "TYPE_TYPE_LIST":
39          self.item = [ TypeList( buff, cm ) for i in xrange(0, self.size) ]
40
41      elif TYPE_MAP_ITEM[ self.type ] == "TYPE_STRING_DATA_ITEM":
42          self.item = [ StringDataItem( buff, cm ) for i in xrange(0, self.size) ]
43
44      elif TYPE_MAP_ITEM[ self.type ] == "TYPE_DEBUG_INFO_ITEM":
45          self.item = DebugInfoItemEmpty( buff, cm )
46
47      elif TYPE_MAP_ITEM[ self.type ] == "TYPE_ENCODED_ARRAY_ITEM":
48          self.item = [ EncodedArrayItem( buff, cm ) for i in xrange(0, self.size) ]
```

```
49
50      elif TYPE_MAP_ITEM[ self.type ] == "TYPE_CLASS_DATA_ITEM":
51          self.item = [ ClassDataItem(buff, cm) for i in xrange(0, self.size) ]
52
53      elif TYPE_MAP_ITEM[ self.type ] == "TYPE_MAP_LIST":
54          pass # It's me I think !!!
55
56      else:
57          bytecode.Exit( "Map item %d @ 0x%x(%d) is unknown" % (self.type, buff.get_idx(), buff.get_idx()))
```

虽说懒人模式，但其实也很简单，就是根据type选择不同的处理方法嘛。比如当type为TYPE_HEADER_ITEM就调用HeaderItem。

等一等，HeaderItem为什么这么眼熟，似乎之前用过。没错在DalvikVMFormat._load函数中用过，但是调用过程不一样。

之前一次是这样DalvikVMFormat ——> __header

这一次是这样DalvikVMFormat ——> map_list ——> map_item

实例化后赋予的变量是不同的。

当处理完毕之后回到MapList。我们看self.CM.add_type_item(TYPE_MAP_ITEM[mi.get_type()], mi, c_item)这句。又出现了CM这个成员变量。我们思考一下，为什么要设计ClassManager这个类和CM这个变量。

从DalvikVMFormat实例化一个与自己相爱相杀的ClassManager之后，每次实例化时（MapList

和MapItem）都要传入这个CM作为参数。
我们看看add_type_item这个函数里的内容：

```
def add_type_item(self, type_item, c_item, item):
    self.__manage_item[ type_item ] = item

    self.__obj_offset[ c_item.get_off() ] = c_item
    self.__item_offset[ c_item.get_offset() ] = item

    sdi = False
    if type_item == "TYPE_STRING_DATA_ITEM":
        sdi = True

    if item != None:
        if isinstance(item, list):
            for i in item:
                goff = i.offset
                self.__manage_item_off.append( goff )

                self.__obj_offset[ i.get_off() ] = i

                if sdi == True:
                  self.__strings_off[ goff ] = i
        else:
            self.__manage_item_off.append( c_item.get_offset() )
```

虽然写的不简单，但是其实目的就是将传入的item放到ClassManager准备好的容器中，比如__manage_item_off或者__obj_offset。
再回到DalvikVMFormat._load我们看看当MapList初始化完毕后的执行情况：

```
self.classes = self.map_list.get_item_type( "TYPE_CLASS_DEF_ITEM" )
self.methods = self.map_list.get_item_type( "TYPE_METHOD_ID_ITEM" )
self.fields = self.map_list.get_item_type( "TYPE_FIELD_ID_ITEM" )
self.codes = self.map_list.get_item_type( "TYPE_CODE_ITEM" )
self.strings = self.map_list.get_item_type( "TYPE_STRING_DATA_ITEM" )
self.debug = self.map_list.get_item_type( "TYPE_DEBUG_INFO_ITEM" )
self.header = self.map_list.get_item_type( "TYPE_HEADER_ITEM" )
```

咦，怎么也是各种成员变量保存map_list当中的item？难道作者闲的蛋疼？
当然不是。仔细看，ClassManage的成员命名，都是以off结尾（变量名也是很有意义的），说明是以原始文件的位移来保存。而DalvikVMFormat的变量都是说明methods、strings，都是按照item的意义来保存。
也就是说ClassManage是从文件的角度来分类存储item，而DalvikVMFormat则是从抽象或者说从人的视角来保存。
想像一下这样的业务场景。你需要解析一个二进制文件。文件本身有自己的格式，字符串池、类型池、指令池等等。但是，这样的格式并不是我们熟悉或者我们需要的格式，我们希望它按照一

个一个包、类、函数、变量。
前一种是现实中文件的格式，后一种是抽象后便于我们理解的格式。所以，为什么DalvikVMFormat有一个ClassManage类型的CM变量，而同时DalvikVMFormat又是它CM变量的vm变量。因为它们从等级上来说都是一样的，一个是抽象文件格式，一个是二进制文件格式。这两个格式合起来就是我们要处理的。
在源码的注释中是这样解释ClassManage的：
This class is used to access to all elements (strings, type, proto ...) of the dex format
of the dex format不就是现实中的格式嘛~
这两个类相生相爱，缺一不可。

0X03 结束语

到这里，dex文件的读取就结束了，之后不同的功能模块会对其进行不同的处理，源码有机会再解读。
再加一点私货：个人认为读源代码先理清楚业务场景，输入是什么输出是什么，目的是什么。之后看整体结构（包结构、类结构、继承关系），最后才是处理细节。所以很多地方看的不够细致，但我会力争把事情说清楚。
感兴趣的各位可以自己阅读代码中的细节处理。（Androguard中有很多一行代码解决问题的地方，可以好好揣摩一下）


微信公众号：jsrc_team
新浪官方微博：
京东安全应急响应中心

固定栏目

技术分享 | 安全意识 | 安全小课堂

点击“阅读原文”

阅读原文