Androguard补完计划——提取加密字符串

2016-10-24 penguin_wwy 京东安全应急响应中心

0X01 概述

上篇分析了Androguard如何读取dex，而且还提到Androguard很适合进行扩展或者移植成为自己项目的某一模块。

本篇文章就来研究一下如何在Androguard基础上进行扩展。

App对抗静态分析的方法之一就是利用反射，如果对反射的字符串进行加密会得到更好的效果，而且不但反射可以通过加密字符串，凡是动态注册、加载都可以通过加密字符串提高隐蔽性，对抗静态分析。

比如这样：

```
1 localIntentFilter.addAction(Fegli.a("OBMpJw07KEgVPC1TLjoMPGI1NBcXOA4BKwQFMiIGGjUMGyUX:Y}MUbRLf{Y"));
```

或者这样：

```
1 Object localObject2 = this.h.getString(Fegli.a("PS5CRyQ=:YK.&]$zqZ1"), Fegli.a(":qO7sY!p@wN"));
```

下面我们就给Androguard补充一个功能，提取dex文件中经过加密的字符串。

要提取dex文件中的加密字符串，主要两个步骤：

1、提取dex文件中的字符串池，获取全部字符串；

2、判断是否为加密字符串。

0X02 提取

首先如何提取？上篇提到Androguard读取dex文件对各个item进行处理，处理逻辑在MapItem.next函数中看一下源码，找到跟字符串相关的Item，有两处。

一个是StringIdItem：

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

主要记录String的偏移：

```
class StringIdItem(object):
    """
        This class can parse a string_id_item of a dex file

        :param buff: a string which represents a Buff object of the string_id_item
        :type buff: Buff object
        :param cm: a ClassManager object
        :type cm: :class:`ClassManager`
    """
    def __init__(self, buff, cm):
        self.__CM = cm
        self.offset = buff.get_idx()

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

一个是StringDataItem：

```
elif TYPE_MAP_ITEM[ self.type ] == "TYPE_STRING_DATA_ITEM":
    self.item = [ StringDataItem( buff, cm ) for i in xrange(0, self.size) ]
```

看看代码：

```
class StringDataItem(object):
    """
        This class can parse a string_data_item of a dex file

        :param buff: a string which represents a Buff object of the string_data_item
        :type buff: Buff object
        :param cm: a ClassManager object
        :type cm: :class:`ClassManager`
    """
    def __init__(self, buff, cm):
        self.__CM = cm

        self.offset = buff.get_idx()

        self.utf16_size = readuleb128( buff )

        self.data = utf8_to_string(buff, self.utf16_size)   #重点，保存String data
        expected = buff.read(1)
        if expected != '\x00':
            warning('\x00 expected at offset: %x, found: %x' % (buff.get_idx(), expected))
```

重点在self.data，通过utf8_to_string函数将字节码转换为字符串。

我们知道了每个字符串保存在每个StringDataItem.data中，那我们如何获得它们呢。回到next函数，MapItem.Item保存所有StringDataItem组成的列表：

```
for i in xrange(0, self.size):
    idx = buff.get_idx()

    mi = MapItem( buff, self.CM )
    self.map_item.append( mi )

    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()

    self.CM.add_type_item( TYPE_MAP_ITEM[ mi.get_type() ], mi, c_item )
```

而这个MapItem会被加入到MapList.map_item这个队列以及self.CM中，当然加入到ClassManager中的过程更复杂。如果从map_item中获取到字符串，需要首先找到处理StringDataItem的mi，然后遍历map_item中的所有MapItem对象，依次拿到MapItem.data，这无疑很复杂。那就让我们把目光放到ClassManager上，看看add_type_item：

```
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
    #当处理StringDataItem时
    if item != None:
        if isinstance(item, list):  #条件为真
            for i in item:          #i为StringDataItem对象
                goff = i.offset     #每个String再dex文件中的偏移
                self.__manage_item_off.append( goff )

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

                if sdi == True:
                  self.__strings_off[ goff ] = i    #字典中保存StringDataItem
        else:
            self.__manage_item_off.append( c_item.get_offset() )
```

咦，似乎有了意外的发现，当处理到StringDataItem时，会设置一个标志位。当标志位为真时，self.__strings_off这个字典才会保存数据，也就是StringDataItem相关的数据。

我们来仔细研究一下这段代码，先理解参数。type_item表示Item的类型，c_item则是mi = MapItem(buff, self.CM)的mi，也就是一个完整的MapItem对象。参数中的item则是mi.get_item( )，也就是MapItem.item。所以当type为StringDataItem时item就是保存StringDataItem对象的列表。

整理一下思路，现在的情况是我们可以从ClassManager中的__strings_off字典根据偏移得到每个StringDataItem。但是悲催的是ClassManager当中并没有获得__strings_off的方法，我们只能自己先加一个：

```
def get__strings_off(self):
    return self.__strings_off
```

只要遍历__strings_off，拿到每个Item，获取data就可以得到字符串了。

类似如下处理：

```
soff = vm.get_class_manager().get__strings_off()
str_list = []
for i in soff:
    str_list.append(soff.get_data())
```

str_list就会保存dex文件中的所有字符串了。

0X03 判断加密字符串

得到所有字符串之后，我们就依次判断它是否是加密字符串。如何判断呢？公司倒是有一个判断随机字符串的工具（也就是人类无法识别的字符串），但毕竟是公司的东西，也没有源码。搞一个字典太费劲，而且字典越大也会影响运行时间。我暂时想了一个办法来判断随机字符串。

首先，先弄个小字典，大概十几二十个有关单词（果然是小字典…），先用小字典过滤一下。对于剩下的字符串，将字符分为大写英文，小写英文，和其他字符（除 \ / .; 以及空格）。对于一般有意义的字符串，ASCII 相对集中，以大写或小写为主。比如ACCESS_CHECKIN_PROPERTIES这是权限，Landroid/os/Debug这是类名。而加密后的字符串ASCII分布就会相对随机比如KS9FRUc6HgxFByUhQ1A=:MN1$gNqcet，这三种字符或者其中两种的字符数量相差就不会太大。我们可以统计一个字符串中三种字符的数量，如果其中两种或三种数量相对接近，就认为是随机字符。

```python
def flag(s, long):
    if s[0] > (0.25 * long) and s[2] < (0.35 * long):
        return True

    if s[0] < (0.25 * long) and (s[2] - s[1]) < (0.1 * long):
        return True

class randStr:
    def __init__(self, data):
        self.data = data
        self.count = len(data)
        self.lowCount = 0
        self.upCount = 0
        self.othCount = 0
        self.isRand = False

    def analyze(self):
        for i in self.data:
            if i in r'\/ .;':
                continue
            elif i >= 'a' and i <= 'z':
                self.lowCount += 1
            elif i >= 'A' and i <= 'Z':
                self.upCount += 1
            else:
                self.othCount += 1
```

```python

        for i in d:
            if i in self.data:
                return False

        if self.count < 5:
            return False
        elif self.othCount == 0 or self.lowCount == 0 or self.upCount == 0:
            return False
        else:
            s = [self.lowCount, self.upCount, self.othCount]
            s.sort()
            self.isRand = flag(s, self.count)

            return self.isRand
```

这是我写的代码，以供参考。

0X04 测试

将模块代码补充完整，我们来测试一波。

准备两个dex文件，一个有加密字符串，一个没有，将加密字符串输出到屏幕。

先测试未加密的dex文件。

```
~/androguard$ ./androstrdata.py -i weijiamiclasses.dex
~/androguard$
```

没有加密，所以没有输出。

再测试一下有加密的dex文件。

```
mfwyvVeKxdPX9C426cCXKd4wY2r5nKaQvgXSHmtl6i2aEsRkLJPrqO7iLT2zhg3E
Sx4EJgMcMHpfMUsVHzwZVi4g:dm}Uwy]U,U
FgFJGywLNjYd:us,zXnRwi6
GFEl:k8ADj*3tu8
Games.API_1P
Gravity.RIGHT or Gravity.NO_GRAVITY
GwURPA==:vdxRtyThNx
HIT_ID in (%s)
  Op #
VAomGiwNMh0RW0Mc:7eHtInFtg2
+ZR3123cdcCzDnr/T2xLFkOfXgPtaoiKSrRS+ZTmmP0=
, mId=
0qxcGqM5j6H5VOXc28Imq02o2iDJzF2LijchZVuusipECWbqfD8CuWSaIowIXsVI
LFoJ:A.mCz,g}Ib
L_PARAMETERS:Ljava/lang/Object;
3B4Vnl+si70eVtw8Yh4NrGqS9wt0YpDAszneZxSqGA2rTwpBsAsSSIREAUQ/Wo88
Vmrsh5SFoeaS2GstmMOODOh2nyuwwLJN2arMT14YfKg=
<TT;>.zza;>;
<TT;>.zzc<
fldV/m3DQfpMh/XkZ7PVfMZ1Iutm0Sb00WK0tHDOxXFMfcZbQOL2m0r0fDApX5hJ
ABMENw==:.waO[1^H8s
WlopDi1D:75M{A&^FR^
s6v+PY0ZikfZAYTGqdgLv/fCC78CCoTMx3IgbrK5VjOd/IM870cjIhMi6ywBt/Ex
LgcCDhc+ACgcGjYBNRg=:@hAayPeKhs
sPuW9cMnn4JlIZD4TqWMl/JyLcHN+NFurYlkNBHpZVAtFN0iAg/JhJBrmGb1Jtrg
Ad Request JSON:
IyoVEzg=:GOyrAbHtZ,
AnychZgAlAV3ZnoBMQDVpC6izMmt//ud3tAm+424uqHZW1kXBktDF/bNPbEluVc2
AwQRLC8rAgQe:vjuIIBlazx
ByQr:FaxbggYR.P
CTtD:}Z$]JsdV$3
^(\d+|FULL_WIDTH)\s*[xX]\s*(\d+|AUTO_HEIGHT)$
abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ1234567890
abcdefghijkmnopqrstxyzABCDEFGHJKLMNPQRSTXY3456789
hieBZwy+ShTIe5lnOXJXLck19dWk3BEnvR2qIYS0QQM=
```

统计一下，共输出45个，有30个经辨认为随机字符串。效果还凑活

如果增加一下字典，再改进算法应该可以更高。

0X05 总结

思路：阅读源码查找字符串池保存位置 ——> 设计输出字符串池 ——> 判断是否加密。