Dalvik字节码自篡改原理及实现

Author:leonnewton

0x00 前言

早在2013年，bluebox就发布了一个可以在运行时篡改Dalvik字节码的Demo。之后，国内有很多对这个Demo进行分析的资料，分析得也非常详细。但是看了很多资料以后，感觉缺少从实现的角度可以给学习的同学进行练手的资料。因此，本文从实现的角度，解释篡改的原理并实现篡改的过程，并附上实现的源代码。希望可以帮助想自己实现着玩的同学。

代码： Dalvik字节码自篡改原理及实现

0x01 篡改的原理

由于Dex文件被映射为只读的，因此从Dex文件自己的代码里来修改自己的字节码是不行的。但是native代码和DVM却运行在相同的比字节码低的层次，所以可以通过native代码来修改字节码。在修改字节码之前，用mprotect把字节码所在的内存重新映射为可写，然后把新的字节码覆盖原来的字节码即可。

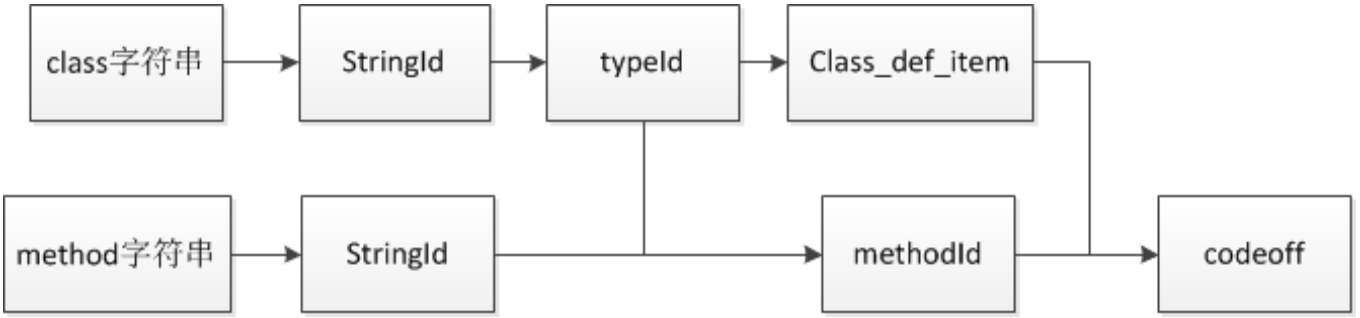
而在应用程序的进程空间，有一段是加载的odex文件，odex的0x28偏移开始的地方就是Dex文件。

```
5e536000-5e693000 r--p 00000000 b3:14 3736       /data/dalvik-cache/data@app@com
.example.selfmodify-2.apk@classes.dex
```

在得到Dex文件在内存中的地址之后，通过查找Dex一系列相应的结构，最终找到存放指令的地址，把新的指令覆盖原来的指令即可。

0x02 搜索method指令位置的过程

找到了Dex文件的位置，需要经过几个查找步骤，最终定位到method存放指令的位置。具体的过程如图所示：



1. 首先是需要确定method的名字，也就是method字符串，再确定method所在class的名字，也就是class字符串。这确定了要修改的方法。
2. 把2个字符串到DexStringId结构的位置进行搜索，进而得到2个字符串在DexStringId中的索引序号。
3. 对于class字符串，再通过DexStrngId索引序号，到DexTypeId中搜索，得到class字符串在

DexTypeId中的索引序号。

4. 还是class字符串，得到DexTypeId中的索引序号后，再到DexClassDef中搜索，就可以得到class的DexClassDef的位置。其中classDataOff字段记录了DexClassData偏移，这个结构里可以找到DexMethod结构。
5. 而对于method字符串，找到DexStringId索引后，再结合上面class的DexTypeId，可以确定DexMethodId的索引序号。
6. 现在我们知道了DexMethodId的索引序号，也知道了存放DexMethod的位置，直接搜索就可以得到我们要找的DexMethod。这个结构里的codeoff指向了存放指令的位置。

0x03 实验的设计

首先写一个应用，应用的TestAdd类有一个add方法，这个方法代码里进行的是乘法，计划篡改字节码后，运行的时候进行的是加法。如下：

```
public int add(int a, int b){
                int c;
                c = a * b;
                return c;
        }
```

篡改后返回a + b的值。

0x04 代码实现

搜索**Dex**文件开头

读取/proc/self/maps，搜索odex文件的地址，从而得到odex的起始地址和结束地址。

```
FILE *fp;
   fp = fopen("/proc/self/maps", "r");
   if(fp!=NULL)
      {
        char line [ 2048 ];
        while ( fgets ( line, sizeof line, fp ) != NULL )
               {
                  if (strstr(line, "/data/dalvik-cache/data@app@com.example.selfmodify") != NU
                      {
                         if (strstr(line, "classes.dex") != NULL)
                             {
                                s = strchr(line, '-');
                                if (s == NULL)
                                    LOGD(" Error: string NULL");
                               *s++ = '\0';
                                start = (void *)strtoul(line, NULL, 16);
```

```
                                end = (void *)strtoul(s, NULL, 16);

                                LOGD(" startAddress = %x", (unsigned int)start);//得到odex起始地
                                LOGD(" endAddress = %x", (unsigned int)end);//得到odex结束地址
                            }
                        }
                    }
            fclose ( fp);
        }
```

其实直接在起始地址的那一页应该就是odex的开头，但是这里还是模拟从后面往前面搜索，search_start_page是odex结束地址的那一页。

```
do{
  search_start_page -= page_size; //page_size是sysconf得到的页大小
  search_start_position = search_start_page + 40;  //加40是因为Dex在odex偏移0x28处
  }while(!findmagic( (void *)(search_start_page + 40) ) ); //findmagic是搜索Dex文件开头的maigc r
```

这样我们就得到了Dex文件的起始地址search_start_position。

得到**method**、**class**字符串的**DexStringId**序号

在DexHeader找到StingId的偏移，根据StringId结构指示的字符串地址，一项一项匹配是否是我们要找的字符串，具体见注释。

```
int getStrIdx(int search_start_position, char *target_string,int size )
{
int index;
int stringidsoff;
int stringdataoff;
int *stringaddress;
int string_num_mutf8;

if(*(int *)(search_start_position+56))//StringId个数不为0
{
   index = 0;
   stringidsoff = search_start_position + *(int *)(search_start_position+60) //StringId在内存的

   while(1)
{
  stringdataoff = *(int *)stringidsoff;
  stringidsoff +=4; //指向下一个StringId结构
  stringaddress = (int *)(search_start_position + stringdataoff); //字符串数据在内存的地址
```

```
  string_num_mutf8 = 0; //首先记录的是后续字符串占的字节数
  if( readUleb128(stringaddress,(int)&string_num_mutf8) == size && !strncmp((char *)stringaddr
//这里readUleb128读取的是记录后面字符串数据占的字节数；strncmp比较的是字符串的内容
       break; //找到了就结束搜索
   ++index; //记录索引序号
  if(*(int *)(search_start_position+56) <= index ) //搜索次数大于StringId数目则结束
     { index = -1;break;}
}
}else
{
index = -1;
}
return index;
}
```

得到**typeId**、**methodId**序号，**ClassDef**地址

这几个过程都很相似，这里只放typeId，其他的代码见链接。

解析过程见注释。

```
signed int  getTypeIdx(int search_start_position, int strIdx)
{
  int typeIdsSize;
  int typeIdsOff;
  int typeid_to_stringid;
  signed int result;
  int next_typeIdsOff;
  int next_typeid_to_stringid;

  typeIdsSize = *(int *)(search_start_position + 64);//typeId的个数
  if ( !typeIdsSize )
    return -1;
  typeIdsOff = search_start_position + *(int *)(search_start_position + 68);//typeId在内存的地址
  typeid_to_stringid = *(int *)typeIdsOff; //指向StringId的索引序号
  result = 0;
  next_typeIdsOff = typeIdsOff + 4; //下一个typeId项
  if ( typeid_to_stringid != strIdx )
  {
    while ( 1 )
    {
      ++result;//记录索引序号
      if ( result == typeIdsSize ) //搜索完所有的typeId项
        break;
      next_typeid_to_stringid = *(int *)next_typeIdsOff;
      next_typeIdsOff += 4; //下一typeId项
```

```
      if ( next_typeid_to_stringid == strIdx )//找到了指定StringId的typeId项
        return result;
    }
    return -1;
  }
  return result;
}
```

得到字节码存放的地址

根据methodIdx和DexClassDef的地址，搜索所有的DexMethod结构，找到要修改的指令。

具体见注释。

```
int getCodeItem(int search_start_position, int class_def_item_address, int methodIdx)
{
  int *classDataOff;
  int staticFieldsSize;
  int *classDataOff_new_start;
  int instanceFieldsSize;
  int directMethodsSize;
  int virtualMethodSize;
  int *after_skipstaticfield_address;
  int *DexMethod_start_address;
  int result;
  int DexMethod_methodIdx;
  int *DexMethod_accessFlagsstart_address;
  int Uleb_bytes_read;
  int tmp;

  classDataOff = (int *)(*(int *)(class_def_item_address + 24) + search_start_position);//得到D
  Uleb_bytes_read = 0; //每次读取的Uleb格式的字节数
  staticFieldsSize = readUleb128(classDataOff, (int)&Uleb_bytes_read); //静态字段的个数
  classDataOff_new_start = (int *)((char *)classDataOff + Uleb_bytes_read); //让地址前进，指向下
  instanceFieldsSize = readUleb128(classDataOff_new_start, (int)&Uleb_bytes_read);//实例字段个数
  classDataOff_new_start = (int *)((char *)classDataOff_new_start + Uleb_bytes_read);
  directMethodsSize = readUleb128(classDataOff_new_start, (int)&Uleb_bytes_read);//直接方法个数
  classDataOff_new_start  = (int *)((char *)classDataOff_new_start + Uleb_bytes_read);
  virtualMethodSize = readUleb128(classDataOff_new_start, (int)&Uleb_bytes_read);//虚方法个数
  after_skipstaticfield_address = skipUleb128(2 * staticFieldsSize, (int *)((char *)classDataO
  DexMethod_start_address = skipUleb128(2 * instanceFieldsSize, after_skipstaticfield_address)
  result = 0;
  if ( directMethodsSize )//直接方法字段存在
  {
    DexMethod_methodIdx = 0;
    do
```

```
    {
      DexMethod_methodIdx = readUleb128(DexMethod_start_address, (int)&Uleb_bytes_read);//直接
      DexMethod_accessFlagsstart_address = (int *)((char *)DexMethod_start_address + Uleb_byte
      if ( DexMethod_methodIdx == methodIdx )//如果是要查找的methodIdx
      {
        readUleb128(DexMethod_accessFlagsstart_address, (int)&Uleb_bytes_read);//读取accessFlag
        return readUleb128((int *)((char *)DexMethod_accessFlagsstart_address + Uleb_bytes_rea
      }
      --directMethodsSize;
      DexMethod_start_address = skipUleb128(2, DexMethod_accessFlagsstart_address);//如果上面不
    }while ( directMethodsSize );
    result = 0;
  }


//跟上面的逻辑是一样的
  if ( virtualMethodSize )
  {
    DexMethod_methodIdx = 0;
    do
    {
      DexMethod_methodIdx = readUleb128(DexMethod_start_address, (int)&Uleb_bytes_read);
      DexMethod_accessFlagsstart_address = (int *)((char *)DexMethod_start_address + Uleb_byte
      if ( DexMethod_methodIdx == methodIdx )
      {
        readUleb128(DexMethod_accessFlagsstart_address, (int)&Uleb_bytes_read);
        return readUleb128((int *)((char *)DexMethod_accessFlagsstart_address + Uleb_bytes_rea
      }
      --virtualMethodSize;
      DexMethod_start_address = skipUleb128(2, DexMethod_accessFlagsstart_address);
    }while ( virtualMethodSize );
    result = 0;
  }
  return result;
}
```

改变权限并复制字节码

codeItem_address是上面我们找到的DexCode开始的地址。

```
void *codeinsns_page_address = (void *)(codeItem_address + 16 - (codeItem_address + 16) % (un
mprotect(codeinsns_page_address,page_size,3);//改为可写
char inject[]={0x90,0x00,0x02,0x03,0x0f,0x00};
memcpy(code_insns_address,&inject,6);//将字节码复制过去
```

0x04 结果

可以看到各个字段读取的结果，本来应该是1*2=2，修改字节码之后为1+2=3了。