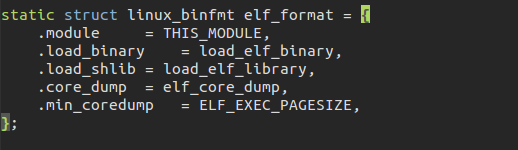
# Android 链接加载器(Linker)

(代码分析基于Android 5.1.1)

## 1. kernel加载ELF文件到linker入口

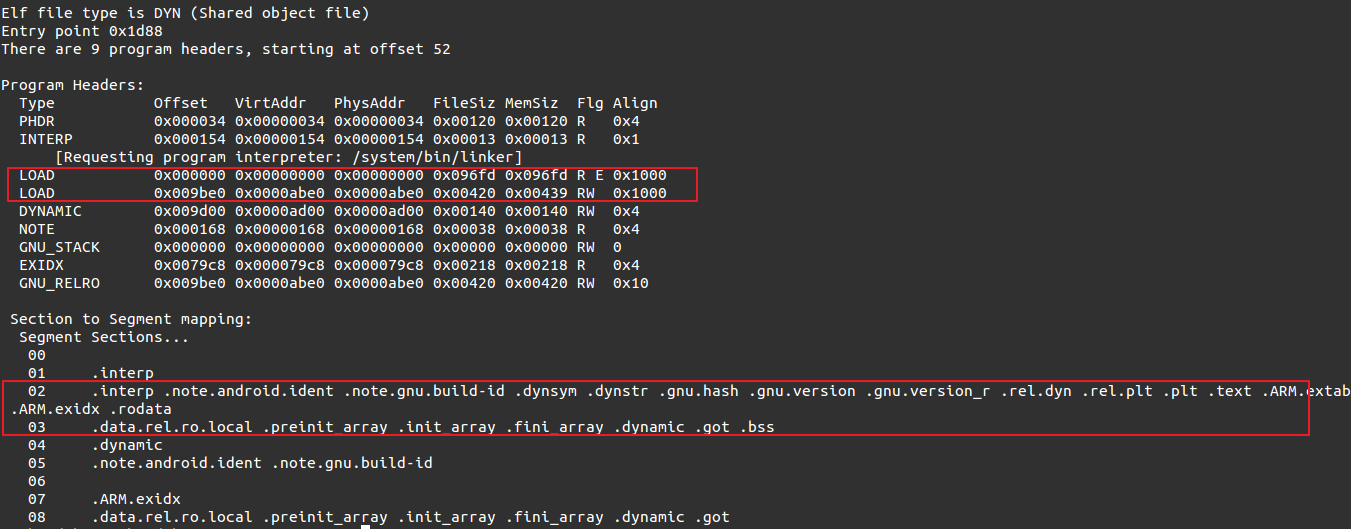
在Android上一个可执行程序的执行需要先调用fork系统调用来分配进程资源，然后调用execve系统调用来执行传入的某可执行程序，这里讨论该可执行程序为ELF文件格式。

execve打开可执行文件后，search\_binary\_handler()尝试遍历可用的文件格式加载模块，其中ELF格式的加载模块实现于binfmt\_elf.c：



**图1**

调用到.load\_binary后，按照ELF文件格式函数会先读取ELF Header(*struct Elf32\_Ehdr*)做文件格式和平台的基本检查；然后根据*Elf32\_Ehdr.e\_phentsize*和*Elf32\_Ehdr*.*e\_phnum*读出ELFa文件中的所有SEGMENT描述结构体*struct Elf32\_Phdr*, 以下是用*readelf*读出的一个ELF文件SEGMENT信息：

******

**图2**

函数*load\_elf\_binary()*再会找到INTERP SEGMENT，按照ELF文件规范这是链接加载器的文件路径；由于INTERP本身也是一个ELF文件，找到INTERP后的存放路径后，函数也会先读出其ELF Header并做一些基本检查；

然后根据找到的*Elf32\_Phdr*来遍历类型为LOAD的SEGMENT，这是ELF规范中的可加载到内存中的SEGMENG；这些SEGMENT描述的是ELF文件中某段文件内容，*Elf32\_Phdr*中的

*p\_offset*和*p\_filesz*描述了其在文件中的偏移和大小；同时*readelf*也给出了这些SEGMENT包含的SECTION，不同的SECTION有不同的作用，它们也由ELF规定的其他结构描述。对于每个LOAD SEGMENT，*Elf32\_Phdr.p\_vaddr*也指定了其加载到内存中的虚拟地址，但是通过查看对应process的/proc/pid/maps可以看到，实际的加载地址并不一样。如上的ELF文件中LOAD SEGMETN加载后的地址是：

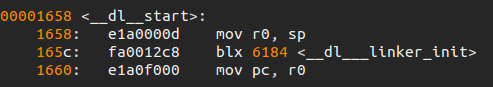


**图3**

考虑对齐，这两个加载地址和ELF文件中的*p\_vaddr*都有一个固定偏移；这是kernel为安全性实现的*ASLR*(*Address Space Layout Randomization*)表现的一部分。

同时把INTERP文件中的LOAD SEGMENT也通过*mmap()*加载到内存中，kernel会根据ELF文件的本身的加载地址来保证INTERP的加载不会有地址覆盖问题。

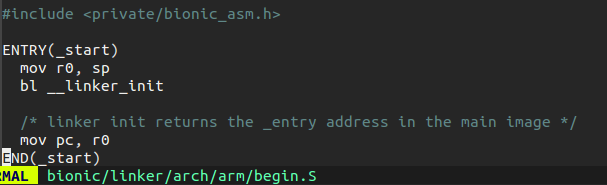
最后调用*create\_elf\_table()*将ELF文件的加载地址以及入口地址等传递给用户空间，最后将PC指向INTERP的入口地址(*Elf32\_Ehdr.e\_entry*)。这个地址可以用objdump看出其对应的符号为:



**图4**

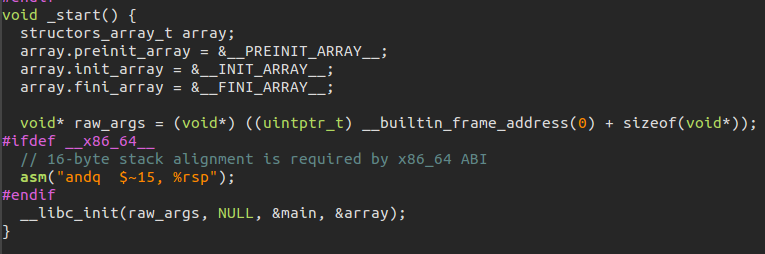
## 2. linker解析ELF文件的section并到main入口

静态链接时toolchain中ld默认将符号*\_start*的地址作为ELF文件的入口地址，linker的*\_start*符号定义在*begin.S*中：



**图5**

而其他可执行程序的*\_start*符号则在libc中的crtbegin.c中:



**图6**

### 2.1 ELF中主要section

kernel按照SEGMENT将ELF文件中的对应内容map到内存中，链接加载器linker则是处理各个SECTION。*struct Elf32\_Ehdr*.*e\_shoff*和*struct Elf32\_Ehdr*.*e\_shnum*描述了SECTION在文件中的偏移和数目，*struct Elf32\_Shdr*结构体用来描述SECTION，其中*Elf32\_Shdr.sh\_type*表示该SECTION的类型，不同的类型的SECTION可以表示该SECTION具有不同的数据结构，如:

*PROGBITS*表示程序自定义的数据，如代码段.text;

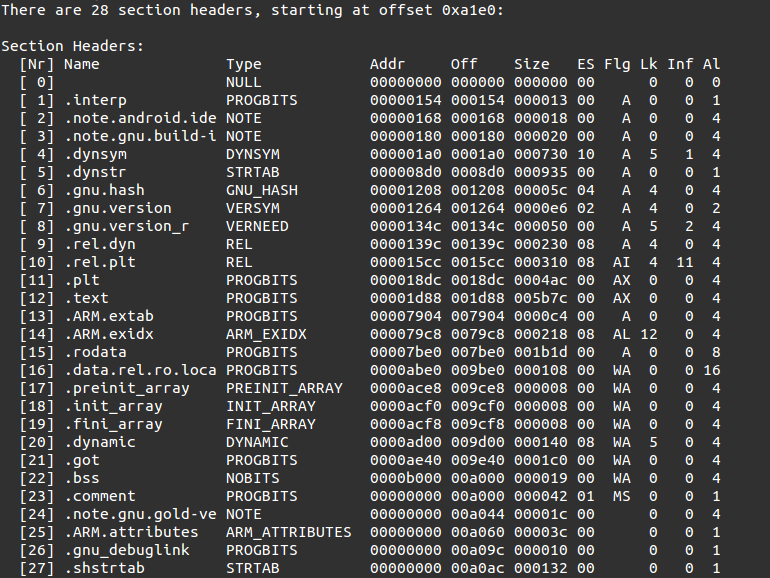
*STRTAB* 是专门用来存放字符串的SECTION, 以空字符开始以空字符结尾;

*DYNAMIC* 数据结构为*struct Elf32\_Dyn*的SECTION, 这里面保存了其他SECTION的起始地址，其他库依赖等动态链接阶段使用的重要信息;

*REL* 数据结构为*struct Elf32\_Rel*的SECTION，描述了符号或数据的重定位信息;

*DYNSYM* 数据结构为*struct Elf32\_Sym*的SECTION，描述了ELF文件中的符号信息，类型、地址等;

用*readelf*查看某ELF文件中的SECTION信息为：



**图7**

### 2.2 linker处理.dynamic SECTION

*begin.S*中linker的入口直接调用了*\_\_linker\_init，*该函数拿到了kernel传递的参数后先调用*soinfo::PrelinkImage()*。该接口拿到.dynamic的地址后，按照结构*Elf32\_Dyn*遍历处理不同的*Elf32\_Dyn.d\_tag*，如：

DT\_STRTAB *Elf32\_Dyn.d\_un.d\_ptr*里面是STRTAB SECTION的地址, 拿到后加上加载地址得到STRTAB在内存的虚拟地址功后续使用;

DT\_SYMTAB *Elf32\_Dyn.d\_un.d\_ptr*里面是DYNSYM SECTION的地址, 拿到后加上加载地址得到DYNSYM在内存的虚拟地址功后续使用;

DT\_JMPREL *Elf32\_Dyn.d\_un.d\_ptr*里面是REL SECTION(*.rel.plt*)的地址, 拿到后加上加载地址得到*.rel.plt*在内存的虚拟地址供后续做重定位使用;

DT\_PLTRELSZ *Elf32\_Dyn.*d\_un.d\_val中存放的是DT\_JMPREL对应的大小;

DT\_REL *Elf32\_Dyn.d\_un.d\_ptr*里面是REL SECTION(*.rel.dyn*)的地址, 拿到后加上加载地址得到*.rel.dyn*在内存的虚拟地址供后续做重定位使用;

DT\_RELSZ *Elf32\_Dyn.*d\_un.d\_val中存放的是DT\_REL对应的大小;

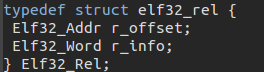
DT\_NEEDED 表示该ELF的依赖文件，*Elf32\_Dyn.*d\_un.d\_val存放的是依赖文件名在STRTAB中的索引;

DT\_PREINIT\_ARRAY、DT\_INIT、DT\_INIT\_ARRAY 、DT\_FINI\_ARRAY、DT\_FINI是可能存在的ELF文件在调用main符号之前和exit之前需要执行的函数; 而DT\_PLTGOT在Android的链接加载器不做处理，因为该linker没有实现符号的延迟重定位。

### 2.3 linker处理重定位

*\_\_linker\_init()*在执行完*soinfo::PrelinkImage()*后，再会调用*soinfo::LinkImage()*。这是所有ELF文件的公共函数，这里选取*.rel.plt*和*.rel.dyn*都存在的/system/bin/debuggerd来说明。

对*.rel.plt*和*.rel.dyn*的处理，都会根据处理.dynamic阶段拿到的对应地址和大小信息来调用函数*soinfo::Relocate(ElfW(Rel)\* rel, unsigned count)*。该函数处理的过程是，根据对应REL段的大小来遍历处理结构*struct Elf32\_Rel,*该结构有如下成员:

**

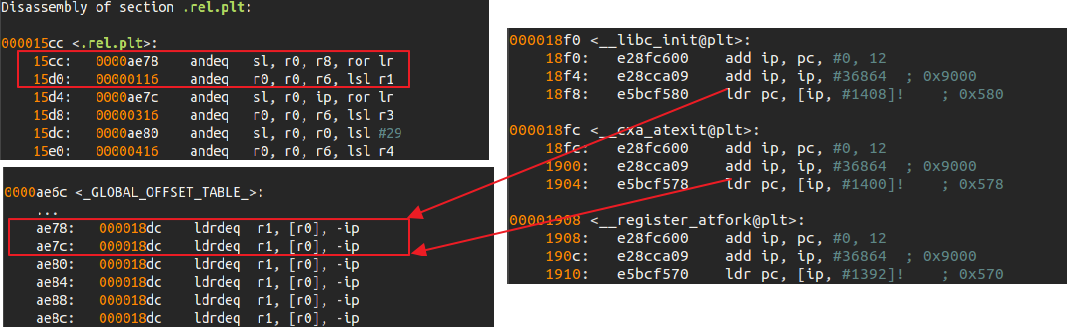
**图8**

其中*offset*为需要重定位进行修改的数据地址; *info*的低8位位重定位类型，不同的重定位类型表示不同的计算方式来修改*offset*地址里的内容，*info*的高8位如果非0时表示的是某个符号在SYMTAB/DYNSYM SECTION中的索引。

当遍历处理*.rel.plt*段时，通过*objdump*来找到相关的信息如下图。对应*.re.plt*中的第一个*Elf32\_Rel*结构，其要修改的重定位地址(*offset*)为*0xae78，*其重定位的类型为0x16对应为

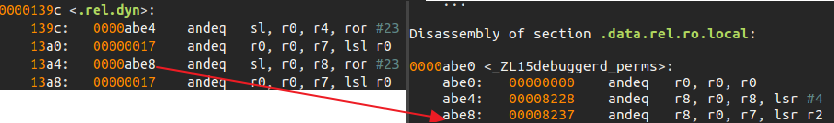
*#define R\_ARM\_JUMP\_SLOT 22*

其重地位的符号索引为1。按照*R\_ARM\_JUMP\_SLOT*的对应规则，linker对这一个重定位项完成的是: 找到符号偏移为1的函数地址写入地址0xae78中。重定位完成后编译链接过程生成的对应的*.plt*段代码就可以在对应地址，如\_\_libc\_init对应的地址0xae78中调用到真正的符号地址。



**图9**

当遍历处理*.rel.dyn*段时，完成的工作类似，不过.rel.dyn完成的是变量的重定位，当ELF文件内部引用了某个静态变量时，编译链接后对这个变量的引用是相对地址，而程序加载到内存中后，变量的地址是有偏移的，因此需要做重定位。

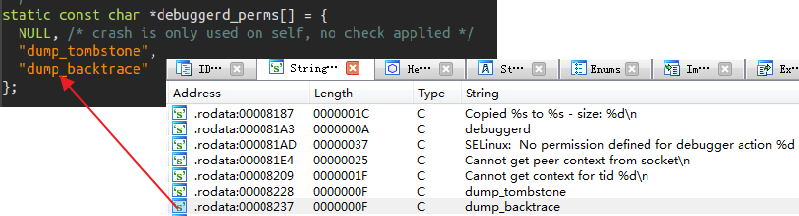


**图10**

同样用*objdump*找到对应的SECTION对比可以看到，linker对*.rel.dyn*的第二项将按照0x17的重定位类型：

*#define R\_ARM\_RELATIVE 23*

把地址为0xabe8中的内容，重定位为原值+偏移，即找到变量加载后的绝对地址。借助*IDA*工具可以看到重定位前，0xabe8中的原值0x8237是*.rodata*段中的字符串，对应源码中的一个静态字符串；重定位后，程序运行对该字符串的引用就可以拿到绝对地址了。



**图11**

另外，在*soinfo::LinkImage()*完成重定位后，会调用一个*phdr\_table\_protect\_gnu\_relro()*函数，该函数会把SEGMENT *PT\_GNU\_RELRO*调用*mprotect()*设置成只读。从图2可以看到，

*PT\_GNU\_RELRO*和第二个LOAD SEGMENT是有多个SECTION重合的，实际上这个步骤就是把第二个LOAD SEGMENT的部分地址在kernel将其mmap到内存后，在完成重定位后修改其读写属性。

**

**图12**

这也是为什么对应ELF文件只有两个LOAD SEGMENT时，查看对应map表会有三个map的地址段的原因。

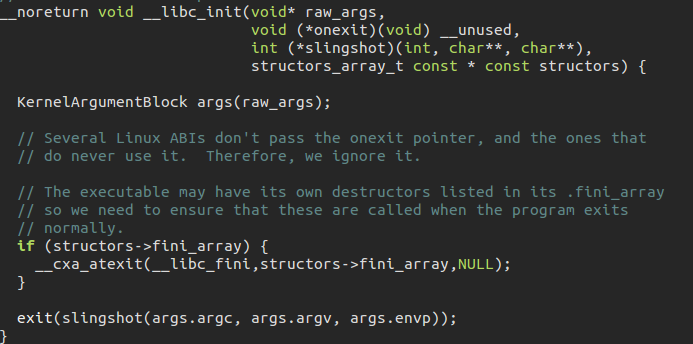
### 2.4 linker处理ELF文件对其他ELF依赖

在完成linker自己的*soinfo::PrelinkImage()*和*soinfo::LinkImage()*后，接下来linker对要真正运行的程序做链接加载，这里记为*process*。在*\_\_linker\_init\_post\_relocation()*函数中，在对*process*做*PrelinkImage*和*LinkImage*之间，linker会把*process*的*.dynamic*段中所有DT\_NEEDED指定的依赖ELF文件进行:

1. 遍历处理*process*的所有依赖DT\_NEEDED ELF；
2. 如果某个DT\_NEEDED没有加载过，则打开ELF文件并mmap LOAD SEGMENT；
3. *PrelinkImage()*该ELF目标;
4. 遍历该ELF的DT\_NEEDED添加到总的遍历列表中去;

循环遍历加载完成后，再对所有的ELF目标做*LinkImage()*。

所有的链接加载完成后，*\_\_linker\_init()*将返回*process*的入口地址，即执行图6的*\_start()*：



**图13**

调用到*\_\_libc\_init()*，其中参数*slingshot*即为main函数。这样可执行程序就通过系统调用，到链接加载器后执行到了用户入口main。

# 参考资料:

<https://duo.com/blog/a-look-at-aslr-in-android-ice-cream-sandwich-4-0>

<http://blog.csdn.net/tabactivity/article/details/78201300>

Android7.0上对于so的加载:

1. linker的增加了struct android\_namespace\_t, 每个namespace

都有可以加载so的路径;

2. linker中有两个默认的namespace:

android\_namespace\_t g\_default\_namespace;

static android\_namespace\_t\* g\_anonymous\_namespace = &g\_default\_namespace;

default的初始化在\_\_linker\_init\_post\_relocation()函数中调用init\_default\_namespace(),

初始化化后, 添加的so加载路径为:

static const char\* const kDefaultLdPaths[] = {

kSystemLibDir,

kVendorLibDir,

nullptr

};

g\_anonymous\_namespace默认和default一致;

3. Android7.0新加的libnativeloader负责APP加载本地so的namespace以及

配置的管理;

3.1 APP创建后,在JVM的创建入口JNI\_CreateJavaVM()会调用libnativeloader的InitializeNativeLoader()

函数, 这个阶段会加载系统所有APP公用的系统的native so, 依据为配置文件:

system/core/rootdir/ptwo/etc/public.libraries.android.txt

vendor/qcom/proprietary/common/config/public.libraries.txt

格式为:

libxx.so [32|64]

3.2 然后APP创建自己的ClassLoader来加载对应的java classs文件时,

就会调用libnativeloader中的CreateClassLoaderNamespace()函数,

该流程会保证:

InitPublicNamespace()被调用过一次;该函数会调用linker的android\_init\_namespaces(),

该流程过后,对应进程中linker的g\_anonymous\_namespace不再是默认值;

调用android\_create\_namespace()创建对应classloader自己的namespace, 即后续该

APP进程会使用的namespace, 该namespace包含的路径有两部分:

java层传递下来的APP自身的加载路径,如:

/data/app/srw.com.ponetest-2/lib/arm:/system/fake-libs:/data/app/srw.com.ponetest-2/base.apk!/lib/armeabi

libnativeloader中配置的系统中所有APP共有的加载路径:

static constexpr const char\* kWhitelistedDirectories = "/data:/mnt/expand:/system/lib:/system/lib64";

3.3 APP启动后, 需要加载第一个native so都要调用的System.loadLibrary()函数, 在对应JVM

的实现JavaVMExt::LoadNativeLibrary()中,也会调用到libnativeloader中的OpenNativeLibrary(),

并使用对应classloader的namespace, 这个时候加载权限就已经被限制了;

4. linker本身对于so的链接加载,除了根据namespace来加载指定路径中的so外,

还提供了一个函数static bool is\_greylisted()来提供给非default namespace

的程序来加载本地so, 其匹配的是一个静态的字符串数组:

static const char\* const kLibraryGreyList[] = {

"libandroid\_runtime.so",

"libbinder.so",

"libcrypto.so",

"libcutils.so",

"libexpat.so",

"libgui.so",

"libmedia.so",

"libnativehelper.so",

"libskia.so",

"libssl.so",

"libstagefright.so",

"libsqlite.so",

"libui.so",

"libutils.so",

"libvorbisidec.so",

nullptr

};

但是这个函数,有可能有其他逻辑会失效,比如

if (get\_application\_target\_sdk\_version() > 23) {

return false;

}

另外,如果将某个依赖的系统so添加到这个list中,在当前版本中不需要考虑其本身的依赖,

因为函数会有如下判断:

if (needed\_by != nullptr && is\_system\_library(needed\_by->get\_realpath())) {

return true;

}