

简单粗暴的 so 加解密实现

一、概述

利用动态链接库实现安卓应用的核心部分，能一定程度的对抗逆向。由于 ida 等神器的存在，还需要对核心部分进行加密。动态链接库的加密，在我看来，有两种实现方式：1. 有源码；2、无源码。无源码的加密，类似 window 平台的加壳和对 .dex 文件的加壳，需要对文件进行分析，在合适的地方插入解密代码，并修正一些参数。而如果有源码，则可以构造解密代码，并让解密过程在 .so 被加载时完成。（当然，应用程序加载了 .so 文件后，内存中 .so 数据已经被解密，可直接 dump 分析。同时，也有一些对抗 dump 的方法，这里就不展开了）。

下文只针对有源码这种方式进行讨论，分析一些可行的实现方法。主要是包含对 ELF header 的分析(不是讨论各个字段含义)；基于特定 section 和特定函数的加解密实现(不讨论复杂的加密算法)。

二、针对动态链接库的 ELF 头分析

网上有很多资料介绍 ELF 文件格式，而且写得很好很详细。我这里就不重复，不太了解的朋友，建议先看看。以下内容，我主要从链接视图和装载视图来分析 ELF 头的各个字段，希望能为读者提供一些 ELF 文件头的修正思路。

这里，我再罗嗦列出 ELF 头的各个字段：

```
typedef struct {  
    unsigned char    e_ident[EI_NIDENT]; /* File identification. */  
    Elf32_Half      e_type; /* File type. */  
    Elf32_Half      e_machine; /* Machine architecture. */  
    Elf32_Word      e_version; /* ELF format version. */  
    Elf32_Addr      e_entry; /* Entry point. */  
    Elf32_Off       e_phoff; /* Program header file offset. */  
    Elf32_Off       e_shoff; /* Section header file offset. */  
    Elf32_Word      e_flags; /* Architecture-specific flags. */  
    Elf32_Half      e_ehsize; /* Size of ELF header in bytes. */  
    Elf32_Half      e_phentsize; /* Size of program header entry. */  
    Elf32_Half      e_phnum; /* Number of program header entries. */  
    Elf32_Half      e_shentsize; /* Size of section header entry. */  
    Elf32_Half      e_shnum; /* Number of section header entries. */  
    Elf32_Half      e_shstrndx; /* Section name strings section. */  
} Elf32_Ehdr;
```

e_ident、e_type、e_machine、e_version、e_flags 和 e_ehsize 字段比较固定；e_entry 入口地址与文件类型有关。e_phoff、e_phentsize 和 e_phnum 与装载视图有关；e_shoff、e_shentsize、e_shnum 和 e_shstrndx 与链接视图有关。目前 e_ehsize = 52 字节，e_shentsize = 40 字节，e_phentsize = 32 字节。

下面看看这两种视图的排列结构：

链接视图	执行视图
ELF 头部	ELF 头部
程序头部表 (可选)	程序头部表
节区 1	段 1
...	
节区 n	段 2
...	
...	...
节区头部表	节区头部表 (可选)

图 1

直接从图中，可以得到一些信息：Program header 位于 ELF header 后面，Section Header 位于 ELF 文件的尾部。那可以推出：

$e_phoff = \text{sizeof}(e_ehsize);$

整个 ELF 文件大小 = $e_shoff + e_shnum * \text{sizeof}(e_shentsize) + 1$

$e_shstrndx$ 字段的值跟 strip 有关。Strip 之前：.shstrtab 并不是最后一个 section. 则 $e_shstrndx = e_shnum - 1 - 2;$

[22]	.got.plt	PROGBITS	00001aac	000aac	000044	04	WA	0	0	4
[23]	.data	PROGBITS	00001af0	000af0	000008	00	WA	0	0	4
[24]	.bss	NOBITS	00001af8	000af8	000008	00	WA	0	0	4
[25]	.comment	PROGBITS	00000000	000af8	00002c	01	MS	0	0	1
[26]	.shstrtab	STRTAB	00000000	000b24	0000ed	00		0	0	1
[27]	.symtab	SYMTAB	00000000	00109c	000450	10		28	46	4
[28]	.strtab	STRTAB	00000000	0014ec	000258	00		0	0	1

图 2

而经过 strip 之后，动态链接库末尾的.symtab和.strtab这两个 section 会被去掉。则 $e_shstrndx = e_shnum - 1$ 。

[22]	.got.plt	PROGBITS	00001aac	000aac	000044	04	WA	0	0	4
[23]	.data	PROGBITS	00001af0	000af0	000008	00	WA	0	0	4
[24]	.bss	NOBITS	00001af8	000af8	000008	00	WA	0	0	4
[25]	.comment	PROGBITS	00000000	000af8	00002c	01	MS	0	0	1
[26]	.shstrtab	STRTAB	00000000	000b24	0000dd	00		0	0	1

Key to Flags:

图 3

使用 ndk 生成在\libs\armeabi\下的.so 文件是经过 strip 的，也是被打包到 apk 中的。可以在\obj\local\armeabi\下找到未经过 strip 的.so 文件。到这里，我们就可以把 <http://bbs.pediy.com/showthread.php?t=188793> 帖子中提到的.so 文件修正。如果 e_shoff 和 e_shnum 都改成任意值，那么修正起来比较麻烦。

感觉上好像 e_shoff 、 e_shnum 等与 section 相关的信息任意修改，对.so 文件的使用毫无影响。的确是这样的，至少给出两个方面来佐证：

1. so 文件在内存中的映射

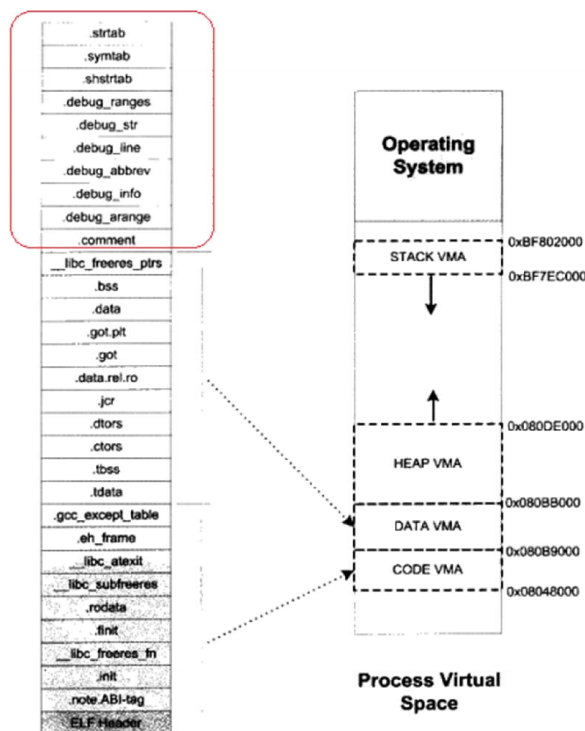


图 4

相信了解 elf 装载(执行)视图的朋友肯定清楚, .so 文件是以 segment 为单位映射到内存的。图中红色区域的 section 是没有被映射的内存, 当然也在 segment 中找不到。

2. 安卓 linker 源码

在 linker.h 源码中有一个重要的结构体 soinfo, 下面列出一些字段:

```
struct soinfo{
    const char name[SOINFO_NAME_LEN]; //so 全名
    Elf32_Phdr *phdr; //Program header 的地址
    int phnum; //segment 数量
    unsigned *dynamic; //指向.dynamic, 在 section 和 segment 中相同的
    //以下 4 个成员与.hash 表有关
    unsigned nbucket;
    unsigned nchain;
    unsigned *bucket;
    unsigned *chain;
```

```
    unsigned *preinit_array;
    unsigned preinit_array_count;
    //这两个成员只能会出现在可执行文件中
```

//指向初始化代码, 先于 main 函数之行, 即在加载时被 linker 所调用, 在 linker.c 可以看到: __linker_init -> link_image -> call_constructors -> call_array

```
    unsigned *init_array;
    unsigned init_array_count;
    void (*init_func)(void);
```

```
//与 init_array 类似，只是在 main 结束之后执行
unsigned *fini_array;
unsigned fini_array_count;
void (*fini_func)(void);
}
```

另外，linker.c 中也有许多地方可以佐证。其本质还是 linker 是基于装载视图解析的 so 文件的。

基于上面的结论，再来分析下 ELF 头的字段。

- 1) e_ident[EI_NIDENT] 字段包含魔数、字节序、字长和版本，后面填充 0。对于安卓的 linker，通过 verify_elf_object 函数检验魔数，判定是否为.so 文件。那么，我们可以向位置写入数据，至少可以向后面的 0 填充位置写入数据。遗憾的是，我在 fedora 14 下测试，是不能向 0 填充位置写数据，链接器报非 0 填充错误。
- 2) 对于安卓的 linker，对 e_type、e_machine、e_version 和 e_flags 字段并不关心，是可以修改成其他数据的(仅分析，没有实测)
- 3) 对于动态链接库，e_entry 入口地址是无意义的，因为程序被加载时，设定的跳转地址是动态连接器的地址，**这个字段是可以被作为数据填充的**。
- 4) so 装载时，与链接视图没有关系，即 e_shoff、e_shentsize、e_shnum 和 e_shstrndx 这些字段是可以**任意修改的**。被修改之后，使用 readelf 和 ida 等工具打开，会报各种错误，相信读者已经见识过了。
- 5) 既然 so 装载与装载视图紧密相关，自然 e_phoff、e_phentsize 和 e_phnum 这些字段是**不能动的**。

根据上述结论，做一个面目全非，各种工具打开报错的 so 文件就很 easy 了，读者可以试试，这里就不举例，你将在后续内容中看到。

三、基于特定 section 的加解密实现

这里提到基于 section 的加解密，是指将 so 文件的特定 section 进行加密，so 文件被加载时解密。下面给出实例。

假设有一个 shelldemo 应用，调用一个 native 方法返回一个字符串供 UI 显示。在 native 方法中，又调用 getString 方法返回一个字符串供 native 方法返回。我需要将 getString 方法加密。这里，将 getString 方法存放在 .mytext 中(指定 __attribute__((section(".mytext"))));，即是需要对 .mytext 进行加密。

加密流程：

- 1) 从 so 文件头读取 section 偏移 shoff、shnum 和 shstrtab
- 2) 读取 shstrtab 中的字符串，存放在 str 空间中
- 3) 从 shoff 位置开始读取 section header，存放在 shdr
- 4) 通过 shdr -> sh_name 在 str 字符串中索引，与 .mytext 进行字符串比较，如果不匹配，继续读取
- 5) 通过 shdr -> sh_offset 和 shdr -> sh_size 字段，将 .mytext 内容读取并保存在 content 中。
- 6) 为了便于理解，不使用复杂的加密算法。这里，只将 content 的所有内容取反，即 *content = ~(*content);
- 7) 将 content 内容写回 so 文件中

- 8) 为了验证第二节中关于 section 字段可以任意修改的结论, 这里, 将 shdr -> addr 写入 ELF 头 e_shoff, 将 shdr -> sh_size 和 addr 所在内存块写入 e_entry 中, 即 ehdr.e_entry = (length << 16) + nsize。当然, 这样同时也简化了解密流程, 还有一个好处是: 如果将 so 文件头修正放回去, 程序是不能运行的。

解密时, 需要保证解密函数在 so 加载时被调用, 那函数声明为: `init_getString __attribute__((constructor))`。(也可以使用 c++构造器实现, 其本质也是用 attribute 实现)

解密流程:

- 1) 动态链接器通过 call_array 调用 init_getString
 - 2) Init_getString 首先调用 getLibAddr 方法, 得到 so 文件在内存中的起始地址
 - 3) 读取前 52 字节, 即 ELF 头。通过 e_shoff 获得.mytext 内存加载地址, ehdr.e_entry 获取.mytext 大小和所在内存块
 - 4) 修改.mytext 所在内存块的读写权限
 - 5) 将[e_shoff, e_shoff + size]内存区域数据解密, 即取反操作: `*content = ~(*content);`
 - 6) 修改回内存区域的读写权限
- (这里是对代码段的数据进行解密, 需要写权限。如果对数据段的数据解密, 是不需要更改权限直接操作的)

利用 readelf 查看加密后的 so 文件:

```
Thomas@Thomas-PC /cygdrive/e/AndroidDevelop/Shelldemo/libs/armeabi
$ readelf.exe -S libdemo.so
There are 22 section headers, starting at offset 0x2474:

Section Headers:
  [Nr] Name                Type              Addr             Off             Size             ES Flg Lk Inf Al
  [ 0] <no-name>             LOUSER+5c58b6fc b867a72c b93f42f7 fffffe9d 8101b283 XSI
OGTxxxxxxxxop 2164371720 2214637744 0
readelf: Warning: section 0: sh_link value of 2164371720 is larger than the number of sections
  [ 1] <no-name>             NULL              8400b0b0 000000 8101b2a1 845fb0b0 x0xxx
xoE 28225712 0 2164336191
readelf: Warning: section 1: sh_link value of 28225712 is larger than the number of sections
  [ 2] <no-name>             LOUSER+101b108 00000000 8101b108 8400b0b0 00 MSLxxxoE
0 2164371720 2214637744
  [ 3] <no-name>             LOPROC+ffffffa8 80b108ab 7fffeb4c 7ffffffa4 8004afb0 XxM
ILGxxxxxxxxop 2147478348 2159063216 2147478364
readelf: Warning: section 3: sh_link value of 2147478348 is larger than the number of sections
  [ 4] <no-name>             LOUSER+a8b0b0 7ffffff90 7fffee98 80b108a9 80b10fa8 XxL
OTxxxxxxxxop 2147478272 2159063216 2147478332
```

图 6

运行结果很简单, 源码见附件



Native method return!

图 7

注意：并不是所有的 section 都能全加，有些数据是不能加密的。比如直接对.text 直接加密，会把与 crt 有关代码也加密，只能选择性的加密。下面将介绍如何实现

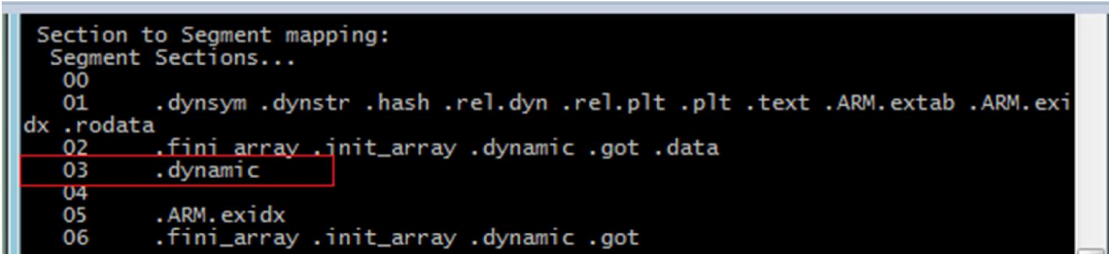
四、基于特定函数的加解密实现

上面的加解密方式可谓简单粗暴。采用这种方式实现，如果 ELF 头 section 被恢复，则很容易被发现 so 多了一个 section。那么，对 ELF 中已存在的 section 中的数据部分加密，可以达到一定的隐藏效果。

与上节例子类似，命名为 shelldemo2，只是 native 直接返回字符串给 UI。需要做的是对 Java_com_example_shelldemo2_MainActivity_getString 函数进行加密。加密和解密都是基于装载视图实现。需要注意的是，被加密函数如果用 static 声明，那么函数是不会出现在.dynsym 中，是无法在装载视图中通过函数名找到进行解密的。当然，也可以采用取巧方式，类似上节，把地址和长度信息写入 so 头中实现。Java_com_example_shelldemo2_MainActivity_getString 需要被调用，那么一定是能在.dynsym 找到的。

加密流程：

- 1) 读取文件头，获取 e_phoff、e_phentsize 和 e_phnum 信息
- 2) 通过 Elf32_Phdr 中的 p_type 字段，找到 DYNAMIC。从下图可以看出，其实 DYNAMIC 就是.dynamic section。从 p_offset 和 p_filesz 字段得到文件中的起始位置和长度



```
Section to Segment mapping:
Segment Sections...
00
01 .dynsym .dynstr .hash .rel.dyn .rel.plt .plt .text .ARM.extab .ARM.exidx
02 .rodata
03 .dynamic
04
05 .ARM.exidx
06 .fini_array .init_array .dynamic .got
```

图 8

- 3) 遍历.dynamic，找到.dynsym、.dynstr、.hash section 文件中的偏移和.dynstr 的大小。在我的测试环境下，fedora 14 和 windows7 Cygwin x64 中 elf.h 定义.hash 的 d_tag 标示是:DT_GNU_HASH;而安卓源码中的是:DT_HASH。
- 4) 根据函数名称，计算 hash 值
- 5) 根据 hash 值，找到下标 hash % nbuckets 的 bucket; 根据 bucket 中的值，

读取.dynsym 中的对应索引的 Elf32_Sym 符号；从符号的 st_name 所以找到在.dynstr 中对应的字符串与函数名进行比较。若不等，则根据 chain[hash % nbuckets]找下一个 Elf32_Sym 符号，直到找到或者 chain 终止为止。这里叙述得有些复杂，直接上代码。

```
for(i = bucket[funHash % nbucket]; i != 0; i = chain[i]){
    if(strcmp(dynstr + (funSym + i)->st_name, funcName) == 0){
        flag = 0;
        break;
    }
}
```

- 6) 找到函数对应的 Elf32_Sym 符号后，即可根据 st_value 和 st_size 字段找到函数的位置和大小
- 7) 后面的步骤就和上节相同了，这里就不赘述

解密流程为加密逆过程，大体相同，只有一些细微的区别，具体如下：

- 1) 找到 so 文件在内存中的起始地址
- 2) 也是通过 so 文件头找到 Phdr; 从 Phdr 找到 PT_DYNAMIC 后，需取 p_vaddr 和 p_filesz 字段，并非 p_offset，这里需要注意。
- 3) 后续操作就加密类似，就不赘述。对内存区域数据的解密，也需要注意读写权限问题。

加密后效果：

```
.text:00001360 ; -----
.text:00001360
.text:00001360          EXPORT Java_com_example_shelldemo2_MainActivity_getString
.text:00001360 Java_com_example_shelldemo2_MainActivity_getString
.text:00001360          BL          0xFFFF09DF8
.text:00001360 ; -----
.text:00001364          DCD 0x58B6FC97, 0x86FF64DC, 0x67A72CBB, 0x3F42F7B8, 0xFFE7D1B9
```

图 10

运行结果与上节相同，就不贴了。

五、参考资料

<http://blog.csdn.net/forlong401/article/details/12060605>

《ELF 文件格式》

Android linker 源码：bionic\linker

Android libc 源码：bionic\libc\bionic

谷歌：ELF 链接视图与装载视图相关资料