

ELF Format DIY For Android

Author: ThomasKing

本文只讨论安卓平台 ELF 格式一些可以 DIY 的地方。当然，有些 DIY 有使用价值，有些 DIY 仅好玩而已。为了完整性，均在下文讨论。

一、Elf32_Ehdr

1. e_ident[16]

这个字段，现 ELF 标准只使用了前 7 个字节，后 9 个字节是未定义的。在 linux 平台，这 9 个字节是填 0，且不能动。而安卓平台，参看 linker 源码，对 so 文件格式只判定前 4 个字节，即 '7f', '45', '4c', '46'。故可 DIY 如下图：

Offset	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	
00000000	7F	45	4C	46	42	79	54	68	6F	6D	61	73	4B	69	6E	67	ELFByThomasKing
00000010	03	00	28	00	01	00	00	00	00	00	00	00	34	00	00	004...
00000020	34	31	00	00	00	00	00	05	34	00	20	00	07	00	28	00	41.....4. ... (.
00000030	15	00	14	00	06	00	00	00	34	00	00	00	34	00	00	004...4...

图 1

2. 平台相关性标识

在 Elf32_Ehdr 中，于平台性标识有：ELF 文件类型 e_type；CPU 平台属性 e_machine；ELF 版本号 e_version(ELF 版本只有 1.2，故该值始终为 1)；文件相关属性 e_flag。这些字段都是来说明 ELF 文件信息，类似产品说明，对 SO 文件的使用无任何影响。故可 DIY 如图。

Offset	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	
00000000	7F	45	4C	46	42	79	54	68	6F	6D	61	73	4B	69	6E	67	ELFByThomasKing
00000010	01	00	14	00	02	00	00	00	00	00	00	00	34	00	00	004...
00000020	34	31	00	00	00	00	00	00	34	00	20	00	07	00	28	00	41.....4. ... (.
00000030	15	00	14	00	06	00	00	00	34	00	00	00	34	00	00	004...4...

type
machine
version
flags

图 2

Readelf 查看信息：

```

/cygdrive/d/workplace
$ readelf.exe -h libhookTest.so
ELF Header:
  Magic:   7f 45 4c 46 42 79 54 68 6f 6d 61 73 4b 69 6e 67
  Class:                                <unknown: 42>
  Data:                                      <unknown: 79>
  Version:                               84 <unknown: %lx>
  OS/ABI:                                <unknown: 68>
  ABI Version:                           111
  Type:                                  REL (Relocatable file)
  Machine:                               PowerPC
  Version:                               0x2
  Entry point address:                   0x0
  Start of program headers:              52 (bytes into file)
  Start of section headers:             12596 (bytes into file)
  Flags:                                0x0
  Size of this header:                   52 (bytes)
  Size of program headers:               32 (bytes)
  Number of program headers:              7
  Size of section headers:               40 (bytes)
  Number of section headers:             21
  Section header string table index:     20

Thomas@Thomas-PC /cygdrive/d/workplace
$

```

图 3

3. e_entry

对于 SO 文件来说，这个值是无意义的。所以随便怎么都行。

4. 与 section 相关

与 section 相关的 e_shoff，e_shentsize，e_shnum, e_shstrndx 随意 DIY，<http://bbs.pediy.com/showthread.php?t=192874> 写得清楚，就不赘述。

5. 其余字段

其余字段由于加载时会被使用，故不能 DIY。详见 linker 源码。

上述 DIY 无处理时机的限制，即既可对 SO 作预处理时，也可在代码中。

二、Section

1. 移动 section

使用 readelf -l 查看 so 文件的 Section to Segment mapping:

```

01 .dynsym .dynstr .hash .rel.dyn .rel.plt .plt .text .ARM.extab .ARM.exidx .rodata
02 .fini_array .init_array .dynamic .got .data .bss
03 .dynamic

```

图 4

总的来说，除了与代码相关受寻址影响的 section 外，其余 section 都是可以移动的。受代码访问影响的 section 有：.plt, .ARM.extab, .ARM.exidx, .rodata, .got, .data, BSS。其余 section 可以随意移动。为了处理方便，移动到的位置最好选在当前所处 LOAD 末尾。由于受到 segment 的属性(RWX)影响，跨 LOAD 处理稍微繁琐。以移动到 LOAD 末尾为例，具体移动某 section 的处理流程如下：

Step1: 选定移动位置。

Step2: 根据对齐属性，计算合适的起始位置。

Step3: 复制 section 数据到新位置。

Step4: 修订 section 在.dynamic 中的位置信息，即 p_offset、p_vaddr 和 p_paddr。

Step5: 若移动 segment，修订对应段在 segment header 中的信息。

这里就不给出例子，下文将看到。

2. 增删 section

查看 section 信息可知，LOAD 内 section 之间是紧凑排列的。删除某 section 的数据，可不移动 section。但增加 section 内容，就需要移动，并且修订 section 时，需修订 p_filesz 和 p_memsz。有些 section 可以单独修改，而有些 section 修改后，需要重新调整与之相关的 section。比如往 dynsym 末尾添加一个符号，并为该符号在 dynstr 添加一 name 字符串。便于查找，计算出 name 的 hash 值，然后往 hash 表中添加。如果还涉及到 rel，还需修改 rel 的 r_offset 和 r_info 字段。根据上述处理，再移动修改相应的 section。这里就不给例子，下文将看到。

3. 修改 init_array

对 fini_array 和 init_array 类似，以 init_array 为例，讨论 init_array 的一些 DIY。

3.1 变更执行顺序

通过 __attribute__((constructor(num))) 声明某一函数(num 值越小，越先执行)，即指定了在 .init_array 中的位置，修改其顺序即可实现。例如：

```
void __attribute__((constructor(101))) kingcoming();
void __attribute__((constructor(200))) soldiercoming();
void kingcoming(){
    __android_log_print(ANDROID_LOG_INFO, "init_array", "King is coming!");
}
void soldiercoming(){
    __android_log_print(ANDROID_LOG_INFO, "init_array", "A soldier is coming!");
}
```

执行结果：

I	10-13 15:42:43.128	14244	14244	com.example.initttest	init_array	King is coming!
I	10-13 15:42:43.128	14244	14244	com.example.initttest	init_array	A soldier is coming!

图 5

修改其顺序，即交换图中红色区域中的数据：

```
00002EA0 | 00 00 00 00 40 0C 00 00 00 00 00 00 51 0C 00 00
00002EB0 | 6D 0C 00 00 00 00 00 00 03 00 00 00 D4 3F 00 00
```

图 6

再执行：

I	10-13 16:08:03.898	14580	14580		init_array	A soldier is coming!
I	10-13 16:08:03.898	14580	14580		init_array	King is coming!

图 7

3.2 普通函数——init 函数

将普通的函数，升级为 init 函数。具体操作步骤：

Step1: 查找目标函数的起始位置。

Step2: 移动 rel.dyn 到 LOAD1 末尾

Step3: 移动 init_array 到 LOAD2 末尾，添加目标函数地址

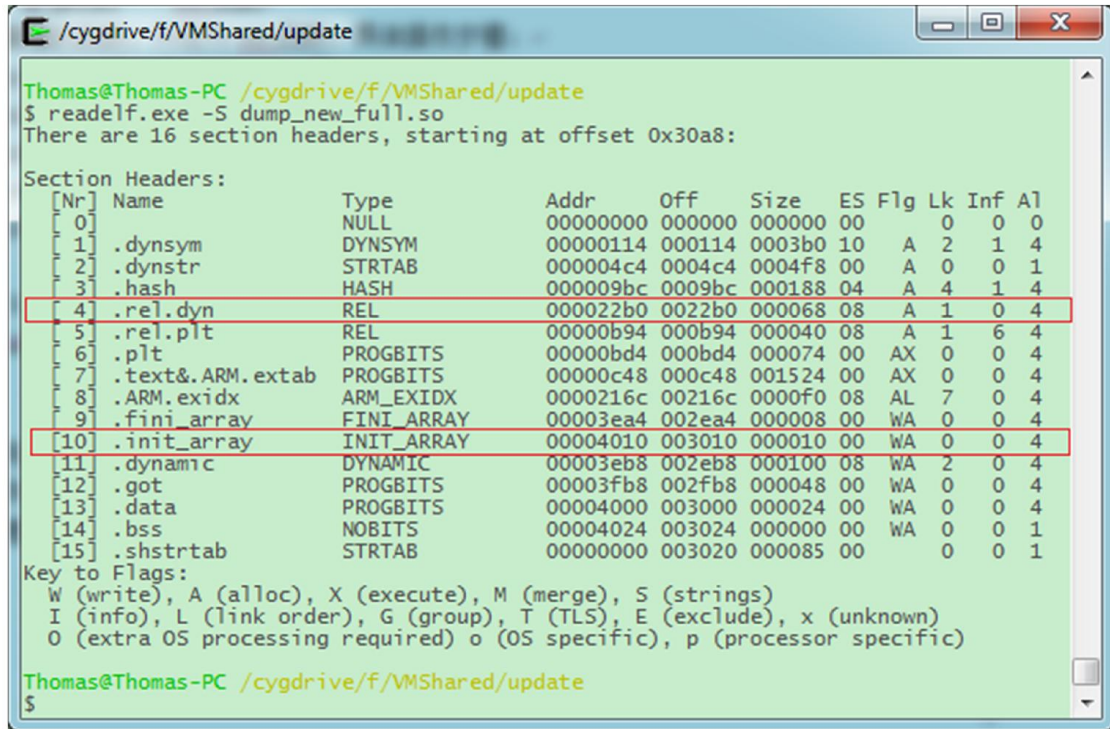
Step4: 修订 rel.dyn 和 init_array 在 .dynamic 中的位置。

Step5: 修订 LOAD1 和 LOAD2 的长度信息

在 3.1 代码中，添加：

```
void justHello(){
    __android_log_print(ANDROID_LOG_INFO, "JNI Tag", "Hello!");
}
```

现将其修改，将 justHello 提升为 init 函数，且最先执行。实现流程上述已讨论，具体代码就不贴了，参看附件中的 updater.c。处理后的 so 文件，需要重建 section 才能查看下图(重建工具：<http://bbs.pediy.com/showthread.php?t=192874>):



```

Thomas@Thomas-PC /cygdrive/f/VMShared/update
$ readelf.exe -S dump_new_full.so
There are 16 section headers, starting at offset 0x30a8:

Section Headers:
[Nr] Name                Type           Addr          Off           Size       ES Flg Lk  Inf Al
[ 0]                     NULL           00000000      000000      000000      00  0  0  0  0
[ 1] .dynsym                DYNSYM         00000114      000114      0003b0      10  A  2  1  4
[ 2] .dynstr                STRTAB         000004c4      0004c4      0004f8      00  A  0  0  1
[ 3] .hash                 HASH           000009bc      0009bc      000188      04  A  4  1  4
[ 4] .rel.dyn              REL            000022b0      0022b0      000068      08  A  1  0  4
[ 5] .rel.plt              REL            00000b94      000b94      000040      08  A  1  6  4
[ 6] .plt                  PROGBITS       00000bd4      000bd4      000074      00  AX 0  0  4
[ 7] .text&.ARM.extab      PROGBITS       00000c48      000c48      001524      00  AX 0  0  4
[ 8] .ARM.exidx            ARM_EXIDX      0000216c      00216c      0000f0      08  AL 7  0  4
[ 9] .fini_array           FINI_ARRAY     00003ea4      002ea4      000008      00  WA 0  0  4
[10] .init_array           INIT_ARRAY     00004010      003010      000010      00  WA 0  0  4
[11] .dynamic              DYNAMIC        00003eb8      002eb8      000100      08  WA 2  0  4
[12] .got                  PROGBITS       00003fb8      002fb8      000048      00  WA 0  0  4
[13] .data                 PROGBITS       00004000      003000      000024      00  WA 0  0  4
[14] .bss                  NOBITS         00004024      003024      000000      00  WA 0  0  1
[15] .shstrtab             STRTAB         00000000      003020      000085      00  0  0  0  1

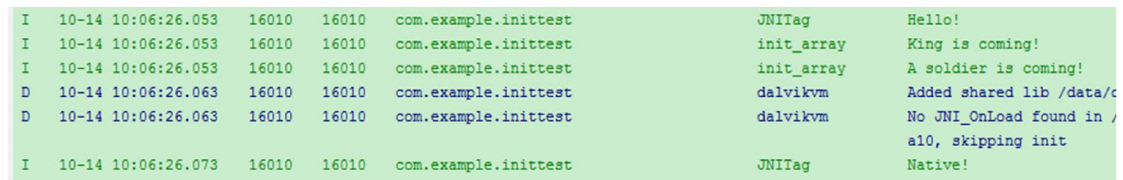
Key to Flags:
W (write), A (alloc), X (execute), M (merge), S (strings)
I (info), L (link order), G (group), T (TLS), E (exclude), x (unknown)
O (extra OS processing required) o (OS specific), p (processor specific)

Thomas@Thomas-PC /cygdrive/f/VMShared/update
$

```

图 8

处理后的 so 文件执行结果:



```

I 10-14 10:06:26.053 16010 16010 com.example.inittest JNIITag Hello!
I 10-14 10:06:26.053 16010 16010 com.example.inittest init_array King is coming!
I 10-14 10:06:26.053 16010 16010 com.example.inittest init_array A soldier is coming!
D 10-14 10:06:26.063 16010 16010 com.example.inittest dalvikvm Added shared lib /data/c
D 10-14 10:06:26.063 16010 16010 com.example.inittest dalvikvm No JNI_OnLoad found in /
a10, skipping init
I 10-14 10:06:26.073 16010 16010 com.example.inittest JNIITag Native!

```

图 9

3.3 init 函数转普通函数

相对 3.2 来说，init 函数转普通函数算较简单吧，这里就不赘述了

4. GOT 表 —— From HOOK to Self Patch

针对 GOT 表的 HOOK 技术屡见不鲜，这里还是啰嗦下一般的 HOOK 应用场景和流程，以便讨论 SO 自 Patch，实现类似目前基于函数 Patch 的第二代 dex 加固技术。下图简单描述了 SO 函数 HOOK 的基本原理。

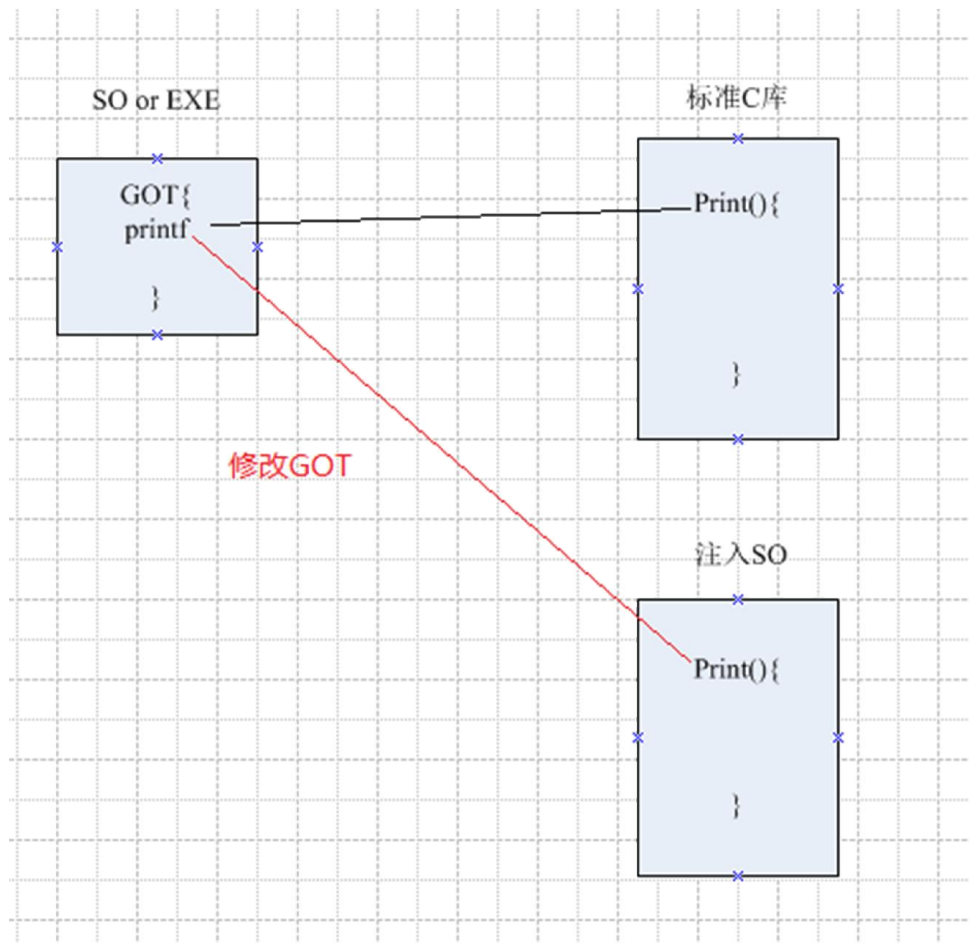


图 10

其中,个人认为有一个很重要的点就是:HOOK 的是 Import 函数,即访问的是外部函数。那么如果 HOOK 自身的函数,达到替换的函数的目的,就实现了类似 DEX 的函数 patch 效果(纯属个人 YY)。

在 linux 平台, PIC 模式的 SO 文件是不区分本地符号还是外部符号,即不管是 Import 符号还是 Export 符号,都走 GOT 过程。采用 HOOK 原理,即可实现对 non-static 的变量和函数的替换,达到 HOOK 和自 Patch 的效果。

思路清晰,似乎就成功了。但是,随意打开一个 ARM 架构的 SO 文件查看 GOT 表,发现 Export 函数根本不在 GOT 中,不过 non-static 变量还是在的(具体 LINUX 平台和 android 在这方面的比较,在附件《Android ELF GOT section の不同之处》,此文中仅根据表象,分析了差异点。限于水平,未找到相关资料佐证,其中的原因分析纯属个人 YY,难免有错误之处,请各位批评指正)。

一种简单的 Patch 思路是,通过生成一个空壳 libFuncStub.so 文件,把需要 Patch 的函数放在其中。目的是构造一个 stub 在 GOT 中。然后 SO 加载起来之后,通过 HOOK GOT 自身函数,达到 Patch 效果。由于 linker 会将依赖的 so 也加载起来,libFuncStub.so 不能扔掉。

作为 SO DIY,不能扔掉是不能忍的。想扔掉 libFuncStub.so,即要绕过 linker 加载机制。一旦绕过 linker,让 libFuncStub.so 不加载,同时函数在执行前被 patch 掉,就能无缝完成这个过程。Patch 利用 HOOK 原理实现,重点就在如何绕过 linker 不加载 libFuncStub.so。

了解 linker 加载 so 大体过程的都知道,linker 会将 so 所依赖的加载起来。具体是:通过 DT_NEEDED 找到对应 dynstr 中的 name 再加载。同一个 so 文件不会被加载二次。另外,libdl.so 一定会被加载。为了让 linker 不加载 libFuncStub.so,我这里采用修改 DT_NEEDED 所

指向的 libFuncStub.so 为 libdl.so，达到 fake 的目的。另外还有一个问题就是函数重定向。

还是查看 linker 源码，发现 linker 对所有的重定位都采用同样的方法，如图所示：

```

switch (type) {
#ifdef ANDROID_ARM_LINKER
    case R_ARM_JUMP_SLOT:
    case R_ARM_GLOB_DAT:
    case R_ARM_ABS32:
    case R_ARM_RELATIVE: /* Don't care. */
    case R_ARM_NONE: /* Don't care. */
#elif defined(ANDROID_X86_LINKER)
    case R_386_JUMP_SLOT:
    case R_386_GLOB_DAT:
    case R_386_32:
    case R_386_RELATIVE: /* Dont' care. */
#endif /* ANDROID_*_LINKER */

```

图 11

似乎可以不用作任何处理。测试发现，加载时报出重定位错误，经过仔细分析 linker 重定位过程，发现一点：由于 _elf_lookup 函数寻找符号时，有此 if(s->st_shndx == 0) continue; 判定，返回 NULL，导致 relocate 错误。st_shndx = 0 即 SHN_UNDEF，故将此设置为非 0 即可。那么整个流程的具体步骤就是：

预处理：

Step1: 抹去 DT_NEEDED 中对应的 so 文件

Step2: 找到对应的函数符号，修改 st_shndx 信息

调用函数前的 Patch 流程：

Step1: 找到 SO 起始内存

Step2: 找到符号表、字符串表、重定位表

Step3: 找到 stub 函数并替换

下面，构造一个简单的例子来说明。为了简单期间，不涉及 section 移动，rel 表组合，加密等等。Java 调用 naïve patchTest，调用 getNameStub 函数获得字符串并打印。getNameStub 定义在 libFuncStub.so 中。实现目标：通过 patch，调用 getNameStub 即调用 SO 自身的 getName 函数。流程上述已说明，代码就不贴了，见附件。patchTest 函数：

```

void patchTest(){
    char name[20];
    int i = 0;
    if(flag) //第一次调用函数时，进行Patch
    {
        patchName("getNameStub", "getName"); //Patch函数
        flag = 0;
    }
    getNameStub(name); //获取到字符串"ThomasKing"，并非"Stub!"
    __android_log_print(ANDROID_LOG_INFO, "JNIITag", "Show my name: %s", name);
}

```

运行效果很简单，就打印一句话：

I	10-14	21:16:16.307	1846	1888	ActivityManager	Displayed com.example.patchtest/.MainActivity: +341ms
I	10-14	21:16:18.357	4273	4273	com.example.patchtest	JNITag Find getNameStub
I	10-14	21:16:18.357	4273	4273	com.example.patchtest	JNITag Find getName
I	10-14	21:16:18.357	4273	4273	com.example.patchtest	HookProcess getNameStub got addr 0x3fe8
I	10-14	21:16:18.357	4273	4273	com.example.patchtest	HookProcess getName addr 0x80a0154d
I	10-14	21:16:18.357	4273	4273	com.example.patchtest	JNITag Show my name: ThomasKing

图 12

当然，PatchName 函数不仅仅只 Patch，可以把基于函数解密融合在一起(基于函数加解密实

现见贴: <http://bbs.pediy.com/showthread.php?t=191649>)。

这里稍微再 YY 一下, 针对无源码加解密实现。(就起原因是上次做了一个很挫的 SO 加解密投去 ALICTF 热身赛)。一种较好的大致思路是, 将原 SO 抹去 section, 加密添加在壳子 SO 末尾。把原 SO 的 rel 表组合到壳子的 rel 表, 壳子可以自身进行基于 SECTION、函数等等加密。执行时, 将原 SO 匿名映射到内存, 修复 SO 的 rel 表。当然, 如果为了保证 JNI 入口地址的一致性, 再使用 NDK HOOK(<http://bbs.pediy.com/showthread.php?t=192047>)到原 SO 函数。

这个 Patch 还算是完美, 毕竟不依靠 stub.so。不过编译时能否不依靠呢? 即又回到如何在 GOT 表中构造 stub 的问题。上述方案是通过 HOOK 函数符号实现。从原理上, 还可以 HOOK non-static 变量。可能会问, non-static 变量本来就是可以访问的, HOOK 无非可以改变函数地址而已。从函数指针的间值寻址出发, 又可以构造一种 Patch 方案, 可使 stub 函数存在于自身 SO 中, 在编译时不依靠 stub.so 文件。具体实现流程和上述差不多, 只是修改 rel.dyn, 限于篇幅, 就不贴了吧。相信各位读者都能做到。

三、Segment

由于 segment 已经包含了一些 section, 对 segment 的 DIY 只是简单的整体移动和长度增长。前面例子已经看到, 就不赘述了。

四、参考文献

Linker 源码

ELF 文件格式

<http://bbs.pediy.com/showthread.php?t=191649>

<http://bbs.pediy.com/showthread.php?t=192047>

<http://bbs.pediy.com/showthread.php?t=192874>