SO 的内存加载及改进加载器

Author: ThomasKing

一、前言

在上文 http://bbs.pediy.com/showthread.php?t=197512《SO 文件抽取及加载器实现》中，讨论了对 SO 文件的简单抽取及加载器的实现。抽取后的 SO 文件与加载器是分离的。本文的目的是，将抽取后的 SO 文件附带在加载器上。不难看出，linker 加载加载器之后，此时的 SO 文件已经处于内存之后，加载器需要从内存中对 SO 进行加载。为了更清楚的说明这个过程，本文将讨论如何在内存中加载普通的 SO 文件；再修改上文中的加载器，实现加载器对抽取 SO 文件的无缝加载和替换。限于水平，难免会有错误和疏漏之处，请各位大大斧正，小弟感激不尽。

二、SO 的内存加载

与 linker 从文件进行加载类似，SO 从内存加载也分为三大步：1、内存映射及文件格式信息解析；2、重定位；3、构造函数的调用。对构造函数调用与 linker 相同，就不赘述。

1、 内存映射

不管是从网上下载，还是从文件解密到内存。此时的 SO 文件已经存在内存中，但并不能保证 SO 的起始地址已经页对齐(页对齐是因为 mprotect 改变内存权限时，是按页处理的)。仍需通过 mmap 重新分配内存，通过格式信息解析，获取 segment 信息，对内存进行填充。

2、重定位

非外部符号的重定位与 linker 相同。对于外部符号，由于程序此时已经加载起来，直接可以通过 dlsym 获取外部符号的地址，重定位即可。

在加载过程中，需要构造一个 soinfo 结构体，目的在于将其链接到 SO list 中。首先看看 linker 是如何管理 so list 的。从 linker.c 的源码中可以看到，soinfo 存放在 sopool 中，其数组最大值为 128。另外，还存在 sonext，freelist，分别指向已加载的 soinfo 链表的尾部，freelist 指向了下一个 sopool 中可用的 soinfo。对于 soinfo 的分配和释放，其实质是对 sonext 和 freelist 进行操作，在函数 alloc_info 和 free_info 中可以看到:

```
static soinfo *alloc_info(const char *name){
    …
    si = freelist;
    freelist = freelist->next;        //取出 si，freelist 指向下一个可用 soinfo
    …
    sonext->next = si;   //更新已加载 soinfo 链表的尾节点
    si->next = NULL;
    …
```

```
}
static void free_info(soinfo *si){
    …
    //回收 soinfo
    prev->next = si->next;
    if (si == sonext) sonext = prev;
    si->next = freelist;
    freelist = si;
}
```

Sopool 等被 static 修饰为模块内变量，是不能直接访问，直接操作 sonext 把构造的 soinfo 链入有点困难。在 linker.h 中，Soinfo 结构体中有一个字段：soinfo *next，指向了下一个 soinfo。故可通过 dlopen 获得任意 SO 的 soinfo *handle(最好打开 libc.so 等系统库)，直接操作其 next(链表的插入操作)，将内存加载 SO 构造的 soinfo 插入 solist。

由于此 soinfo 并非 sopool 分配，稍微存在一些副作用。若此 SO 后续被其他 SO 所依赖，其他 SO 卸载时会调用 unload_library 函数。由于构造的 soinfo 地址不在 sopool 数组范围，validate_soinfo 函数返回 false，即内存加载的 SO 不会调用 unload_library 函数进行释放。解决方法是遵循谁分配谁释放原则，将内存释放在加载器的析构函数中完成。另外，稍微啰嗦一点，可以把内存加载 SO 的 soinfo->recount 置为一个较大的数，保证其在被 dlclose 调用时，unload_library 不会释放内存，始终由加载器处理。

三、改进加载器

在《SO 文件抽取及加载器实现》虽然实现了文件的抽取和加载器，但二者分离，稍微显得笨拙了。抽取后的 SO 文件从内存加载和上面讨论的并无区别，下面简单讨论。

Linker 根据 dynamic 获取的 DT_LOAD 进行内存文件映射，标准的 SO 文件存在 2 个 LOAD 段，可将抽取后的 SO 文件附带在加载器任意一个 LOAD 段，再对加载器文件结构进行调整。

若附带在 LOAD1，之前讨论动了骨架需要修改指令集，故附带在 LOAD1 尾部不妥。注意到 load_library 函数中对 Phdr 的查找 si->phdr = (Elf32_Phdr *)((unsigned char *)si->base + hdr->e_phoff)。可以将抽取后的 SO 文件插入到 Elf32_Ehdr 和 Elf32_Phdr 之间，故须修改 e_phoff、load1 的 offset、vaddr、filesz 和 memsz。插入后由于虚地址变化，需要作一些调整，在《SO 文件抽取及加载器实现》已经讨论，就不再赘述。

附带到 LOAD1 稍显复杂，附带到文件末尾也即 LOAD2 末尾就简单许多，不影响加载器的虚地址，只需修正 load2 的 filesz 和 memsz 即可完成。最后，将附带的抽取 SO 文件虚地址存放在 e_shoff 中，便于加载器提取数据。也可将抽取 SO 文件页对齐附带在加载器文件之后，省去内存分配。

现加载器与抽取 SO 文件融合在一起后，整个加载过程及其后，加载器对于上层应是透明的。所有这里并不采用前文将 soinfo 插入链表，而采取移花接木的方式，直接利用 linker

分配给加载器的 soinfo。

在之前的文章中，简单分析了 soinfo 的各个字段，这里就不再赘述。一个 SO 文件依赖另外一个 SO 文件，其本质就是查找被依赖文件的导出符号。故仅将加载器 soinfo 与符号查找相关的部分替换成抽取后的 SO 文件即可：

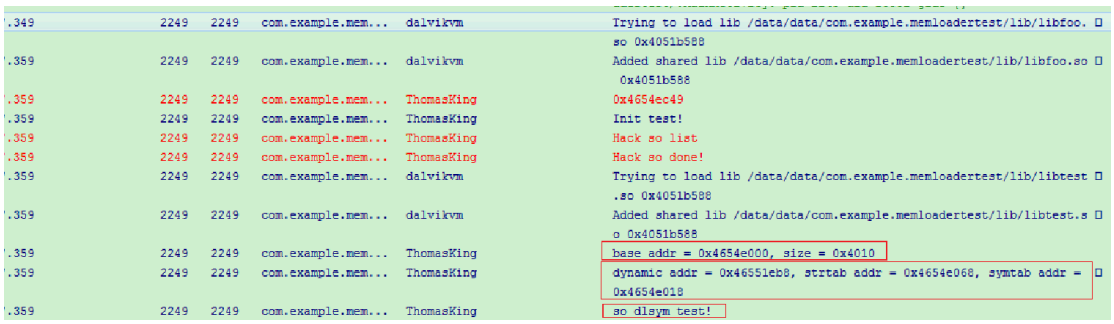
```
loader->strtab = si.strtab;
loader->symtab = si.symtab;
loader->nbucket = si.nbucket;
loader->nchain = si.nchain;
loader->bucket = si.bucket;
loader->chain = si.chain;
```

四、测试

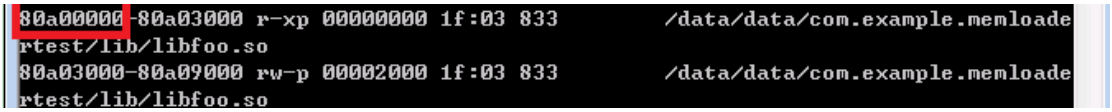
为了从 log 中看出效果，并作 loader->base = si.base; loader->size = si.size;替换。在原测试中增加一个 libtest.so，在其 JNI_OnLoad 中通过 dlsym 查找 foo 中的 test 方法，以说明抽取 SO 文件已被无缝加载起来。

```
JNIEXPORT jint JNICALL JNI_OnLoad(JavaVM* vm, void* reserved)
{
    void (*call_test)();
    soinfo *handle;
    handle = (soinfo*)dlopen("libfoo.so", RTLD_NOW);
    if(handle != NULL){
        LOGD("base addr = 0x%x, size = 0x%x\n", handle->base, handle->size);
        LOGD("dynamic addr = %p, strtab addr = %p, symtab addr = %p\n", handle->dynamic,
handle->strtab, handle->symtab);
        call_test = (void (*)())dlsym(handle, "test");
        if(call_test != NULL){
            call_test();
        }else{
            LOGD("cannot find %s\n", "test");
        }
    }else{
        LOGD("dlopen %s failed\n", "libfoo.so");
    }
    return JNI_VERSION_1_4;
}
```

Log 输出：



Maps 查看 libfoo.so 的起始地址：



至此，对 ELF 文件格式和 linker 的学习告一段落。

五、参考文献

linker 源码