Linker 笔记

2017年9月6日 星期三　　下午2:18

先来个4.4.3 的源码分析

直接从__linker_init 开始分析

首先就是把自己的的一些elf header里的偏移赋值

```
1835  linker_so.size = phdr_table_get_load_size(phdr, elf_hdr->
e_phnum);
1836  linker_so.load_bias = get_elf_exec_load_bias(elf_hdr);
```

这两个函数一个获取LOAD 段的起始和结束地址，第二个函数获取第一个LOAD段

调用soinfo_link_image()进行自举和重定位

获取phdr 和dynamic section 的地址保存起来

另外获取一个arm_exidx section的内容，可能是跟异常处理有关的

，然后就是解析dynamicsection 里的 各个 字段，把这些值赋值到传进来的soinfo里，解析的东西比较多，期间还有一些条件编译宏，比如MIPS的，但是这个并不重要，我们只分析arm的， 这些case可以自己去看。

赋值完了之后，对几个值做一个检查， 比 如strtab，strsym，nbucket。

然后就是加载环境变量里指定的so，这个操作主要是针对要加载进来的so的，但是我们现在的linker还在自举当中，所以不需要

```
    // If this is the main executable, then load all of the libraries from
LD_PRELOAD now.
1494    if (si->flags & FLAG_EXE) {
1495        memset(gLdPreloads, 0, sizeof(gLdPreloads));
1496        size_t preload_count = 0;
1497        for (size_t i = 0; gLdPreloadNames[i] != NULL; i++) {
```

```
1498            soinfo* lsi = find_library(gLdPreloadNames[i]);
1499            if (lsi != NULL) {
1500                gLdPreloads[preload_count++] = lsi;
1501            } else {
1502                // As with glibc, failure to load an LD_PRELOAD library is
just a warning.
1503                DL_WARN("could not load library \"%s\" from
LD_PRELOAD for \"%s\"; caused by %s",
1504                        gLdPreloadNames[i], si->name,
linker_get_error_buffer());
1505            }
1506        }
1507    }
```

在这之后，linker开始加载dynamic里tag为DT_NEEDED的字段，也就是一些so文件，忘findlibrary跟，就会发现又跑到soinfo_link_image这里了，所以就不分析了。

```
    soinfo** needed = (soinfo**) alloca((1 + needed_count) *
sizeof(soinfo*));
1510    soinfo** pneeded = needed;
1511
1512    for (Elf32_Dyn* d = si->dynamic; d->d_tag != DT_NULL; ++d) {
1513        if (d->d_tag == DT_NEEDED) {
1514            const char* library_name = si->strtab + d->d_un.d_val;
1515            DEBUG("%s needs %s", si->name, library_name);
1516            soinfo* lsi = find_library(library_name);
1517            if (lsi == NULL) {
1518                strlcpy(tmp_err_buf, linker_get_error_buffer(),
sizeof(tmp_err_buf));
```

```
1519            DL_ERR("could not load library \"%s\" needed by \"%s\"; caused by %s",
1520                    library_name, si->name, tmp_err_buf);
1521            return false;
1522        }
1523        *pneeded++ = lsi;
1524    }
1525  }
1526  *pneeded = NULL;
```

然后是判断so有无重定位，有的话就修改属性，之后的话还得改回来。
最后就是最重要的重定位了，总共两个，一个plt_rel,一个rel， 重定位的主要工作主要是修复导入符号的引用

```
1542
1543  if (si->plt_rel != NULL) {
1544      DEBUG("[ relocating %s plt ]", si->name );
1545      if (soinfo_relocate(si, si->plt_rel, si->plt_rel_count, needed)) {
1546          return false;
1547      }
1548  }
1549  if (si->rel != NULL) {
1550      DEBUG("[ relocating %s ]", si->name );
1551      if (soinfo_relocate(si, si->rel, si->rel_count, needed)) {
1552          return false;
1553      }
1554  }
```

看4.4.3的代码，这一块，这两个重定位都调用了soinfo_relocate()

一个重定位数据的格式如下， struct{

Elf32_Addr r_offset； 需要重定位的信息

Elf32_Word r_info ；低8为重定位类型，高24代表符号表中的位置

}

程序首先 符号表的索引值有么有，然后获取符号名称， 看其他so是否有加载这个符号， 也就是调用了

```
9           sym_name = (char *)(strtab + symtab[sym].st_name);
870           s = soinfo_do_lookup(si, sym_name, &lsi, needed);
```

，不过有就用找到的这个，没有就为0

首先看soinfo_do_lookup干了些什么事情 ， 这个函数主要是根据名字，去so自己，或者是其他加载的so里去找符号的信息，最终 返回一个符号信息 ，查找符号主要是用到了nbucket chain，这个后面可以再分析，这两个其实是为了快速找到符号而准备的。正常情况下这个符号是一定找的到的，也就是最后sym_addr 一定有值的，如果没有，没准就是被修改了。。。

接着又是一个switch，把sym地址根据不同的类型给reloc ， 如果函数是内部的，直接加上so的机制就好

关于符号表类型的问题可以参考boyliang的博客

http://blog.csdn.net/L173864930/article/details/40507359，其中也讲到了bucket，chain的东西，我就不再展开了。

这个就是重定位的流程了。

在重定位完成之后，代码 执行__linker_init_post_relocation(args,linker) 执行主逻辑，和linker不是特别相关的代码就略过了。

第一步__libc_init_tls(args) ,初始化tls ， 然后 linker_env_init ,检查setuid ，还有一些加载的环境变量。然后获取LD_PRELOAD 宏定义的so，如果有的话，后续会进行加载。接着分配一个soinfo结构体，并对其进行赋值

然后又调用了soinfo_link_image() ,

接着add_vdso() , 这个给漏洞利用提供了一种新的思路,这个args 是kernle 传进来的，暂时就不分析了

```
1591static void add_vdso(KernelArgumentBlock& args UNUSED) {
1592#ifdef AT_SYSINFO_EHDR
1593  Elf32_Ehdr* ehdr_vdso = reinterpret_cast<Elf32_Ehdr*>
(args.getauxval(AT_SYSINFO_EHDR));
1594
1595  soinfo* si = soinfo_alloc("[vdso]");
1596  si->phdr = reinterpret_cast<Elf32_Phdr*>(reinterpret_cast<char*>
(ehdr_vdso) + ehdr_vdso->e_phoff);
1597  si->phnum = ehdr_vdso->e_phnum;
1598  si->link_map.l_name = si->name;
1599  for (size_t i = 0; i < si->phnum; ++i) {
1600    if (si->phdr[i].p_type == PT_LOAD) {
1601      si->link_map.l_addr = reinterpret_cast<Elf32_Addr>(ehdr_vdso) - si->
phdr[i].p_vaddr;
1602      break;
1603    }
1604  }
1605#endif
1606}
```

最后调用

```
1735  si->CallPreInitConstructors();
1736
1737  for (size_t i = 0; gLdPreloads[i] != NULL; ++i) {
1738    gLdPreloads[i]->CallConstructors();
1739  }
1740
1741  /* After the link_image, the si->load_bias is initialized.
1742   * For so lib, the map->l_addr will be updated in notify_gdb_of_load.
1743   * We need to update this value for so exe here. So Unwind_Backtrace
1744   * for some arch like x86 could work correctly within so exe.
1745   */
1746  map->l_addr = si->load_bias;
1747  si->CallConstructors();、
```

做一些main之前的初始化操作

之后如果定义了relro这个flag的话，就把重定位表设置成只读，这个写利用的

同学就比较了解了，里面实际是用了mprotect进行保护
最后还有一个操作通知gdb，so加载完成了。。。。。厉害咯

最后 设置soinfo_pool为只读属性

```
static void set_soinfo_pool_protection(int protection) {
290  for (soinfo_pool_t* p = gSoInfoPools; p != NULL; p = p->next) {
291    if (mprotect(p, sizeof(*p), protection) == -1) {
292      abort(); // Can't happen.
293    }
294  }
295}
```

通过System.loadLibrary进入
最后在so层会通过dlopen打开，这个函数会调用find_library 加载一个so ，返回soinfo，然后，调用so的init函数。Find_library 函数调用load_library 进行so的加载，这个load_libray 可以看作是一个__linker_init 函数了，因为它做的事和linker有点像，都调用了soinfo_link_image ,然后就返回了，soinfo_link_image .
关键点是调用loadlibrary的时候，有一个ElfReader ， 这个会调用Load()进行加载。

```
34bool ElfReader::Load() {
135  return ReadElfHeader() &&
136         VerifyElfHeader() &&
137         ReadProgramHeader() &&
138         ReserveAddressSpace() &&
139         LoadSegments() &&
140         FindPhdr();
141}
```

实际就是在加载so ，这几个都比较简单， 易懂就不在分析了。

7.1.1 源码分析

7.0以后linker这里改了好多，用了很多c++代码

Linker一开始 创建了一个linker_so对象，然后又是赋值，接着调用 linker_so.prelink_image ,Androidxref 貌似对c++支持不太好，搜定义的时候会搜不到，只能全局搜索。

这个prelink_image 跟之前的soinfo_link_image 有一定的相似，但是并没有加载其他的so，也没有进行重定位什么的操作

接着调用link_image, 基本也是一样的，所以4.4的linker版本 soinfo_link_image == 7.1 的pre_link 和link_image

往下走 ，4489 行初始化一些全局变量，这些变量是linker，libc 共同需要的。4492 行调用一些main函数执行前的函数，比如init， init_array 指定的函数

```
 // Initialize the main thread (including TLS, so system calls really work).
4482  __libc_init_main_thread(args);
4483
4484  // We didn't protect the linker's RELRO pages in link_image because we
4485  // couldn't make system calls on x86 at that point, but we can now...
4486  if (!linker_so.protect_relro()) __linker_cannot_link(args);
4487
4488  // Initialize the linker's static libc's globals
4489  __libc_init_globals(args);
4490
4491  // Initialize the linker's own global variables
4492  linker_so.call_constructors();
```

4493

再往下是获取libdl的so信息，并把so加到全局列表里，暂时不清楚这个是要干啥，至此linker自举过程就完成了，然后又开始调用主流程

```
4497 solist = get_libdl_info();
4498 sonext = get_libdl_info();
4499 g_default_namespace.add_soinfo(get_libdl_info());
```

主流程和之前的没什么太大差别，就不在重复了。