Elf 文件 segment 区加解密

1. linker 运行原理

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

（1）soinfo 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 _constructor -> call init_array ->

Main()->fini_array

（2）对于安卓的 linker，对 e_type、e_machine、e_version 和 e_flags 字段并不关心

（3）so 装载与装载视图紧密相关，e_phoff、e_phentsize 和 e_phnum 这些字段是不能动的

2. __attribute__属性

__attribute__((constructor(n))) void init_1();

__attribute__((section (“.mydefsegment”))) void xxx_fun();

constructor(n)：在 init_array 函数中运行， n 指定的值越小，越先运行

section （“xxxx”）:构造制定的 section 节区

```
void init_1() __attribute__((constructor(1)));
void init_2() __attribute__((constructor(2)));
void init_3() __attribute__((constructor(3)));

//__attribute__((section (".init_array"))) const char* init_value="t0000
__attribute__((section ("xxx2"))) const char* xxx2value="segment_tes
__attribute__((section ("xxx"))) void segment_test(){
    LOGI ("segment xxx: segment_test");
    printf("%s",xxx2value);
    printf ("xxxx2312313xx");
    printf ("%d",213);
}
```

3. 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.myencry 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。

ehdr.e_entry 保存了 section 的地址 sh_size 和 elf 文件占页 nsize，

ehdr.e_shoff 保存了加密的偏移地址

```
int fd;
fd = open(libname, O_RDWR);
if(fd < 0){
    printf("open %s failed\n", libname );
    goto _error;
}

if(read(fd, &ehdr, sizeof(Elf32_Ehdr)) != sizeof(Elf32_Ehdr)){
    puts("Read ELF header error");
    goto _error;
}
```

```
//str segment
lseek(fd, ehdr.e_shoff + sizeof(Elf32_Shdr) * ehdr.e_shstrndx, SEEK_SET);
if(read(fd, &shdr, sizeof(Elf32_Shdr)) != sizeof(Elf32_Shdr)){
    puts("Read ELF section string table error");
    goto _error;
}
if((shstr = (char *) malloc(shdr.sh_size)) == NULL){
    puts("Malloc space for section string table failed");
    goto _error;
}
lseek(fd, shdr.sh_offset, SEEK_SET);
if(read(fd, shstr, shdr.sh_size) != shdr.sh_size){
    puts("Read string table failed");
    goto _error;
}
//
```

```
lseek(fd, ehdr.e_shoff, SEEK_SET);
for(i = 0; i < ehdr.e_shnum; i++){
    if(read(fd, &shdr, sizeof(Elf32_Shdr)) != sizeof(Elf32_Shdr)){
        puts("Find section .text procedure failed");
        goto _error;
    }
    if(strcmp(shstr + shdr.sh_name, target_section) == 0){
        base = shdr.sh_offset;                    //search target section
        length = shdr.sh_size;
        printf("Find section %s\n", target_section);
        break;
    }
}
lseek(fd, base, SEEK_SET);                        //target to encode
content = (char*) malloc(length);
if(content == NULL){
    puts("Malloc space for content failed");
    goto _error;
}
if(read(fd, content, length) != length){
    puts("Read section .text failed");
```

```
nblock = length / block_size;
nsize = base / 4096 + (base % 4096 == 0 ? 0 : 1);     //save sh_off sh_size
printf("base = %4x, length = %d\n", base, length);    //nsize for mprocte
printf("nblock = %d, nsize = %d\n", nblock, nsize);

ehdr.e_entry = (length << 16) + nsize;
ehdr.e_shoff = base;

//encode the segment "xxx" code
for(i=0;i<length;i++){                                //encode the target section
    content[i] = ~content[i];
}
```

```
lseek(fd, 0, SEEK_SET);
if(write(fd, &ehdr, sizeof(Elf32_Ehdr)) != sizeof(Elf32_Ehdr)){
    puts("Write ELFhead to .so failed");        //save header
    goto _error;
}

lseek(fd, base, SEEK_SET);
if(write(fd, content, length) != length){          //save segmetn
                                                   context
    puts("Write modified content to .so failed");
    goto _error;
}
puts("Completed");
```

4. 解密流程：

1) 首先调用 init_array 函数中的解密函数

2) 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.myencry 大小和所在内存块

4) 修改.myencry 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

```
extern "C" void decode_segment(const char *libname,const char* segment_name){
    unsigned long  libAddr =getlibaddr (libname);
    unsigned int nblock= 0;
    unsigned int nsize=0;
    Elf32_Ehdr*  elf_hdr=NULL;
    Elf32_Shdr* segment_shder=NULL;
    Elf32_Phdr* program_phdr=NULL;
    unsigned  long textcode_addr;
    elf_hdr=(Elf32_Ehdr*)libAddr;
    textcode_addr=libAddr+elf_hdr->e_shoff;
    nblock = elf_hdr->e_entry>>16;
    nsize=elf_hdr->e_entry&0xFFFF;
    LOGI("nclock = %d \n",nblock);
    LOGI("nsize = %d \n",nsize);
    if(mprotect ((void*)libAddr,4096*nsize,PROT_WRITE|PROT_READ|PROT_EXEC)!=0){
        LOGI("mem privilege change failed");
    }
    int i;
    for(i=0;i< nblock; i++){
        char *addr = (char*)(textcode_addr + i);
        *addr = ~(*addr);
    }
```

```
if(mprotect ((void*)libAddr,4096*nsize,PROT_READ|PROT_EXEC)!=0){
    LOGI("mem privilege change failed");
}
```

附件:

segment.cpp

segment.h

segment_enc.cpp

segment_enc.h

segment_test.c

segment_testlib.c

Android.mk

原文参考：
http://www.520monkey.com/archives/563
https://www.cnblogs.com/ichunqiu/p/5999799.html