Android GOT表HOOK技术

Nov 2, 2014

本文主要介绍了Android平台GOT表HOOK技术。

CONTENTS

§ 原理

在完成对目标进程的动态库注入后，我们通常还需要改变目标进程的执行流程、替换原函数从而达到自己的目的。而所谓的Hook是指改变待Hook函数的入口地址，转而指向我们的函数，变更原函数功能。

§ 流程

GOT表Hook过程如下：

1. 注入SO(libhook.so)成功后，调用dlsym函数，获取SO中函数handle_hook的地址；
2. 调用函数handle_hook，完成Native Hook；

§ 实现

1. 注入SO(libhook.so)成功后，调用dlsym函数，获取SO中函数handle_hook的地址：
   首先调用dlopen(“libhook.so”, “RTLD_NOW”)，返回一个SO句柄handle，调用dlsym(handle, “handle_hook”),返回libhook.so中函数handle_hook的地址handle_hook_addr。
2. 调用函数handle_hook，完成GOT表Hook：
   通过解析SO文件，将待Hook函数在got表的地址替换为自己函数的入口地址，这样目标进程每次调用待Hook函数时，实际上是执行了我们自己的函数。（ps：事实上没有说的这么简单，需要对elf文件格式、动态库装载原理有深入理解）

§ 关键

1. elf中重要的三个表：
   字符串表：包含以空字符结尾的字符序列，使用这些字符来描绘符号和节名；
   符号表：保存了一个程序在定位和重定位时需要的定义和引用的信息；
   重定位表：保存了需要重定位的符号的信息；
   （ps：之前所说解析SO文件，其实主要目的就是获得这三个表的信息）

2. 如何找到待Hook函数在got表的地址以及自己函数的入口地址：

   - 读取ELF文件头（ELF文件头起始于 ELF 文件开始的第一字节），取出：
     节头表的文件偏移（shdr_base），获取节头表在文件中的位置；
     节头表的项数（shnum），获取节头表的项数；
     与节名称字符串表关联的项的节头表索引（shstr_base），并将其缓存起来（shstr）；

```c
int fd;
fd = open(module_path, O_RDONLY);
Elf32_Ehdr *ehdr = (Elf32_Ehdr *)malloc(sizeof(Elf32_Ehdr));
read(fd, ehdr, sizeof(Elf32_Ehdr)) != sizeof(Elf32_Ehdr);
uint32_t shdr_base = ehdr -> e_shoff;
uint16_t shnum = ehdr -> e_shnum;
uint32_t shstr_base = shdr_base + ehdr -> e_shstrndx * sizeof(Elf32_Shdr);
Elf32_Shdr *shstr = (Elf32_Shdr *)malloc(sizeof(Elf32_Shdr));
lseek(fd, shstr_base, SEEK_SET);
read(fd, shstr, sizeof(Elf32_Shdr));
```

   - 读取节头表索引，取出：
     节的大小（sh_size）、节的名称（sh_name）；
     遍历节名，分别将节名为.dynsym、.dynstr、.got、.rel.plt的节缓存起来（dynsym_shdr、dynstr_shdr、got_shdr、relplt_shdr）；
     通过上一步获取的节，分别获得对应的表，并将其缓存；

```c
char *shstrtab = (char *)malloc(shstr -> sh_size);
lseek(fd, shstr -> sh_offset, SEEK_SET);
read(fd, shstrtab, shstr -> sh_size);
```

```
Elf32_Shdr *shdr = (Elf32_Shdr *)malloc(sizeof(Elf32_Shdr));
lseek(fd, shdr_base, SEEK_SET);
uint16_t i;
char *str = NULL;
Elf32_Shdr *relplt_shdr = (Elf32_Shdr *) malloc(sizeof(Elf32_Shdr));
Elf32_Shdr *dynsym_shdr = (Elf32_Shdr *) malloc(sizeof(Elf32_Shdr));
Elf32_Shdr *dynstr_shdr = (Elf32_Shdr *) malloc(sizeof(Elf32_Shdr));
Elf32_Shdr *got_shdr = (Elf32_Shdr *) malloc(sizeof(Elf32_Shdr));
for(i = 0; i < shnum; ++i) {
    read(fd, shdr, sizeof(Elf32_Shdr));
    str = shstrtab + shdr -> sh_name;
    if(strcmp(str, ".dynsym") == 0)
        memcpy(dynsym_shdr, shdr, sizeof(Elf32_Shdr));
    else if(strcmp(str, ".dynstr") == 0)
        memcpy(dynstr_shdr, shdr, sizeof(Elf32_Shdr));
    else if(strcmp(str, ".got") == 0)
        memcpy(got_shdr, shdr, sizeof(Elf32_Shdr));
    else if(strcmp(str, ".rel.plt") == 0)
        memcpy(relplt_shdr, shdr, sizeof(Elf32_Shdr));
}

//读取字符表
char *dynstr = (char *) malloc(sizeof(char) * dynstr_shdr->sh_size);
lseek(fd, dynstr_shdr->sh_offset, SEEK_SET);
if(read(fd, dynstr, dynstr_shdr->sh_size) != dynstr_shdr->sh_size)
    return -1;

//读取符号表
Elf32_Sym *dynsymtab = (Elf32_Sym *) malloc(dynsym_shdr->sh_size);
printf("dynsym_shdr->sh_size\t0x%x\n", dynsym_shdr->sh_size);
lseek(fd, dynsym_shdr->sh_offset, SEEK_SET);
if(read(fd, dynsymtab, dynsym_shdr->sh_size) != dynsym_shdr->sh_size)
    return -1;

//读取重定位表
Elf32_Rel *rel_ent = (Elf32_Rel *) malloc(sizeof(Elf32_Rel));
lseek(fd, relplt_shdr->sh_offset, SEEK_SET);
```


- 获取指定符号在got表的偏移地址：

C

```
for (i = 0; i < relplt_shdr->sh_size / sizeof(Elf32_Rel); i++)
{
    uint16_t ndx = ELF32_R_SYM(rel_ent->r_info);
    LOGD("ndx = %d, str = %s", ndx, dynstr + dynsymtab[ndx].st_name);
    if (strcmp(dynstr + dynsymtab[ndx].st_name, symbol_name) == 0)
    {
        LOGD("符号%s在got表的偏移地址为: 0x%x", symbol_name, rel_ent->r_offset);
        offset = rel_ent->r_offset;
        break;
    }
    if(read(fd, rel_ent, sizeof(Elf32_Rel)) != sizeof(Elf32_Rel))
    {
        LOGD("获取符号%s的重定位信息失败", symbol_name);
        return -1;
    }
}

//获取指定符号的地址
if(offset == 0)
{
    LOGD("获取符号%s在got表中的偏移地址失败，可能为静态链接，开始重新获取符号地址", symbol_name);
    for(i = 0; i < (dynsym_shdr->sh_size) / sizeof(Elf32_Sym); ++i)

    {
        if(strcmp(dynstr + dynsymtab[i].st_name, symbol_name) == 0)
        {
            LOGD("符号%s的地址位: 0x%x", symbol_name, dynsymtab[i].st_value);
            offset = dynsymtab[i].st_value;
            break;
        }
    }
}

if(offset == 0)
{
    LOGD("符号%s地址获取失败", symbol_name);
    return -1;
}
```

3. 如何实现对接：
   在实际情况下，我们有一种需求是，在执行完我们自己的函数后，需要返回继续执行原函数，并且要对该函数持续的Hook。我的思路是这样的：在我们自己的函数中，首先采用内联汇编的方式将全部寄存器值压栈保存（STMFD SP!, {R0 - R12, LR}），接着执行自己的流程，最后将保存的寄存器值出栈恢复（LDMFD SP!, {R0 - R12}），并且强制跳转到原函数的入口地址（”LDR LR, =original_addr \n” “LDR LR, [LR] \n” “BLX LR \n”），最后强制程序返回。事实上，就是把原函数当做自己函数的子函数。

```c
__asm__ __volatile__ (
        //"SUB LR, LR, #4 \n"
        "STMFD SP!, {R0 - R12, LR} \n"
);

//自己的函数
myfun();

__asm__ __volatile__ (
    "LDMFD SP!, {R0 - R12} \n"
    "LDR LR, =original_addr \n"
    "LDR LR, [LR] \n"
    "BLX LR \n"
    "LDMFD SP!, {PC} \n"
);
```

§ 参考

Ele7enxxh's Blog

Android利用ptrace实现Hook API：http://blog.sina.com.cn/s/blog_dae890d10101f00d.html