**人怜直节生来瘦，自许高材老更刚。**



# Android so注入(inject)和Hook技术学习（二）——Got表hook之导入表hook

全局符号表(GOT表)hook实际是通过解析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。

GOT表其实包含了导入表和导出表，导出表指将当前动态库的一些函数符号保留，供外部调用，导入表中的函数实际是在该动态库中调用外部的导出函数。

这里有几个关键点要说明一下：

（1） so文件的绝对路径和加载到内存中的基址是可以通过 /proc/[pid]/maps 获取到的。

（2） 修改导入表的函数地址的时候需要修改页的权限，增加写权限即可。

（3） 一般的导入表Hook是基于注入操作的，即把自己的代码注入到目标程序，本次实例重点讲述Hook的实现，注入代码采用上节所有代码inject.c。

导入表的hook有两种方法，以hook fopen函数为例。

**方法一：**

通过解析elf格式，分析Section header table找出静态的.got表的位置，并在内存中找到相应的.got表位置，这个时候内存中.got表保存着导入函数的地址，读取目标函数地址，与.got表每一项函数入口地址进行匹配，找到的话就直接替换新的函数地址，这样就完成了一次导入表的Hook操作了。

hook流程如下图所示：

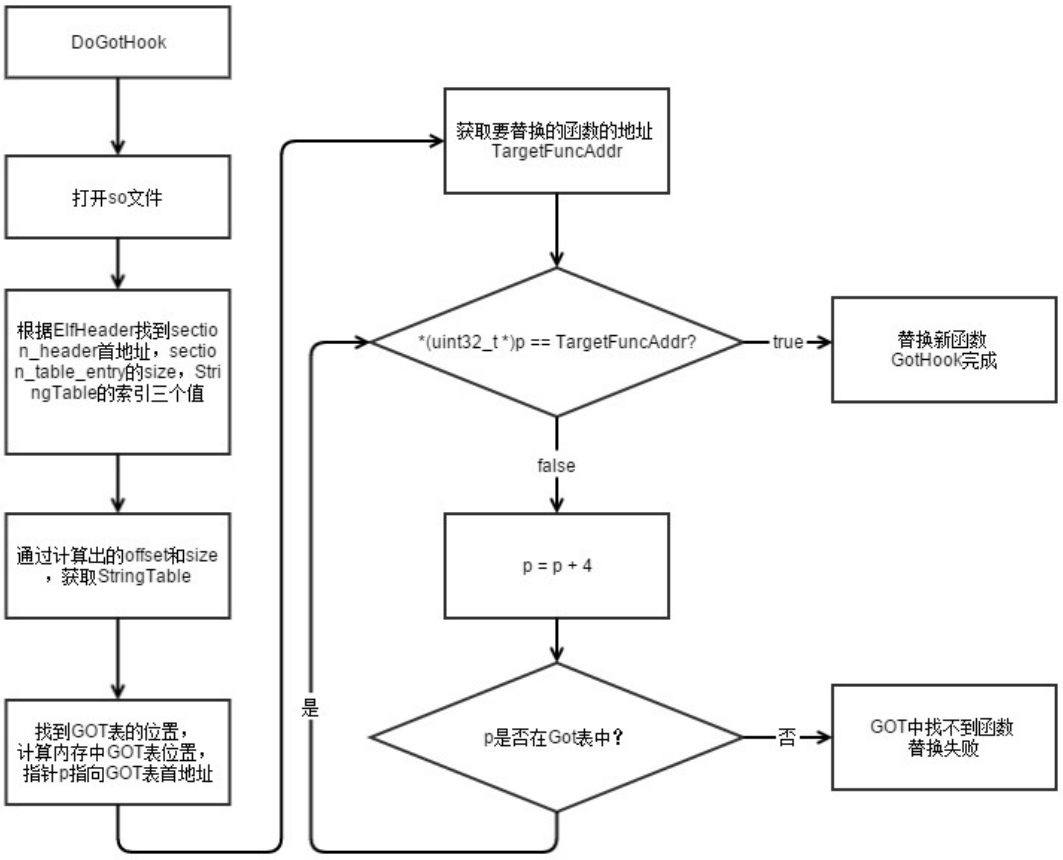


图1 导入表Hook流程图

具体代码实现如下：

entry.c:

```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <android/log.h>
#include <EGL/egl.h>
#include <GLES/gl.h>
#include <elf.h>
#include <fcntl.h>
#include <sys/mman.h>

#define LOG_TAG "INJECT"
#define LOGD(fmt, args...) __android_log_print(ANDROID_LOG_DEBUG, LOG_TAG, fmt, ##args)

//FILE *fopen(const char *filename, const char *modes)
FILE* (*old_fopen)(const char *filename, const char *modes);
FILE* new_fopen(const char *filename, const char *modes){
    LOGD("[+] New call fopen.\n");
    if(old_fopen == -1){
        LOGD("error.\n");
    }
    return old_fopen(filename, modes);
}

void* get_module_base(pid_t pid, const char* module_name){
    FILE* fp;
    long addr = 0;
    char* pch;
    char filename[32];
    char line[1024];

    // 格式化字符串得到 "/proc/pid/maps"
    if(pid < 0){
        snprintf(filename, sizeof(filename), "/proc/self/maps");
    }else{
        snprintf(filename, sizeof(filename), "/proc/%d/maps", pid);
    }

    // 打开文件/proc/pid/maps，获取指定pid进程加载的内存模块信息
    fp = fopen(filename, "r");
    if(fp != NULL){
        // 每次一行，读取文件 /proc/pid/maps中内容
        while(fgets(line, sizeof(line), fp)){
            // 查找指定的so模块
            if(strstr(line, module_name)){
                // 分割字符串
                pch = strtok(line, "-");
                // 字符串转长整形
                addr = strtoul(pch, NULL, 16);

                // 特殊内存地址的处理
                if(addr == 0x8000){
                    addr = 0;
                }
                break;
            }
        }
    }
    fclose(fp);
    return (void*)addr;
}

#define LIB_PATH "/data/app-lib/com.bbk.appstore-2/libvivosgmain.so"
int hook_fopen(){

    // 获取目标pid进程中"/data/app-lib/com.bbk.appstore-2/libvivosgmain.so"模块的加载地址
    void* base_addr = get_module_base(getpid(), LIB_PATH);
```

```
    LOGD("[+] libvivosgmain.so address = %p \n", base_addr);

    // 保存被Hook的目标函数的原始调用地址
    old_fopen = fopen;
    LOGD("[+] Orig fopen = %p\n", old_fopen);

    int fd;
    // 打开内存模块文件"/data/app-lib/com.bbk.appstore-2/libvivosgmain.so"
    fd = open(LIB_PATH, O_RDONLY);
    if(-1 == fd){
        LOGD("error.\n");
        return -1;
    }

        // elf32文件的文件头结构体Elf32_Ehdr
    Elf32_Ehdr ehdr;
    // 读取elf32格式的文件"/data/app-lib/com.bbk.appstore-2/libvivosgmain.so"的文件头信息
    read(fd, &ehdr, sizeof(Elf32_Ehdr));

    // elf32文件中节区表信息结构的文件偏移
    unsigned long shdr_addr = ehdr.e_shoff;
    // elf32文件中节区表信息结构的数量
    int shnum = ehdr.e_shnum;
    // elf32文件中每个节区表信息结构中的单个信息结构的大小（描述每个节区的信息的结构体的大小）
    int shent_size = ehdr.e_shentsize;

    // elf32文件节区表中每个节区的名称存放的节区名称字符串表，在节区表中的序号index
    unsigned long stridx = ehdr.e_shstrndx;

    // elf32文件中节区表的每个单元信息结构体（描述每个节区的信息的结构体）
    Elf32_Shdr shdr;
    // elf32文件中定位到存放每个节区名称的字符串表的信息结构体位置.shstrtab
    lseek(fd, shdr_addr + stridx * shent_size, SEEK_SET);
    // 读取elf32文件中的描述每个节区的信息的结构体（这里是保存elf32文件的每个节区的名称字符串的）
    read(fd, &shdr, shent_size);
    LOGD("[+] String table offset is %lu, size is %lu", shdr.sh_offset, shdr.sh_size); //41159, size is 2
54


    // 为保存elf32文件的所有的节区的名称字符串申请内存空间
    char * string_table = (char *)malloc(shdr.sh_size);
    // 定位到具体存放elf32文件的所有的节区的名称字符串的文件偏移处
    lseek(fd, shdr.sh_offset, SEEK_SET);
    // 从elf32内存文件中读取所有的节区的名称字符串到申请的内存空间中
    read(fd, string_table, shdr.sh_size);

    // 重新设置elf32文件的文件偏移为节区信息结构的起始文件偏移处
    lseek(fd, shdr_addr, SEEK_SET);

    int i;
    uint32_t out_addr = 0;
    uint32_t out_size = 0;
    uint32_t got_item = 0;
    int32_t got_found = 0;

    // 循环遍历elf32文件的节区表（描述每个节区的信息的结构体）
    for(i = 0; i<shnum; i++){
        // 依次读取节区表中每个描述节区的信息的结构体
        read(fd, &shdr, shent_size);
        // 判断当前节区描述结构体描述的节区是否是SHT_PROGBITS类型
        //类型为SHT_PROGBITS的.got节区包含全局偏移表
        if(shdr.sh_type == SHT_PROGBITS){
            // 获取节区的名称字符串在保存所有节区的名称字符串段.shstrtab中的序号
            int name_idx = shdr.sh_name;

            // 判断节区的名称是否为".got.plt"或者".got"
            if(strcmp(&(string_table[name_idx]), ".got.plt") == 0
                || strcmp(&(string_table[name_idx]), ".got") == 0){
                // 获取节区".got"或者".got.plt"在内存中实际数据存放地址
                out_addr = base_addr + shdr.sh_addr;
                // 获取节区".got"或者".got.plt"的大小
                out_size = shdr.sh_size;
```

```
                LOGD("[+] out_addr = %lx, out_size = %lx\n", out_addr, out_size);
                int j = 0;
                // 遍历节区".got"或者".got.plt"获取保存的全局的函数调用地址
                for(j = 0; j<out_size; j += 4){
                    // 获取节区".got"或者".got.plt"中的单个函数的调用地址
                    got_item = *(uint32_t*)(out_addr + j);
                    // 判断节区".got"或者".got.plt"中函数调用地址是否是将要被Hook的目标函数地址
                    if(got_item == old_fopen){
                        LOGD("[+] Found fopen in got.\n");
                        got_found = 1;
                        // 获取当前内存分页的大小
                        uint32_t page_size = getpagesize();
                        // 获取内存分页的起始地址（需要内存对齐）
                        uint32_t entry_page_start = (out_addr + j) & (~(page_size - 1));
                        LOGD("[+] entry_page_start = %lx, page size = %lx\n", entry_page_start, page_size
);
                        // 修改内存属性为可读可写可执行
                        if(mprotect((uint32_t*)entry_page_start, page_size, PROT_READ | PROT_WRITE | PROT
_EXEC) == -1){
                            LOGD("mprotect false.\n");
                            return -1;
                        }
                        LOGD("[+] %s, old_fopen = %lx, new_fopen = %lx\n", "before hook function", got_it
em, new_fopen);

                        // Hook函数为我们自己定义的函数
                        got_item = new_fopen;
                        LOGD("[+] %s, old_fopen = %lx, new_fopen = %lx\n", "after hook function", got_ite
m, new_fopen);
                        // 恢复内存属性为可读可执行
                        if(mprotect((uint32_t*)entry_page_start, page_size, PROT_READ | PROT_EXEC) == -1)
{
                            LOGD("mprotect false.\n");
                            return -1;
                        }
                        break;
                    // 此时，目标函数的调用地址已经被Hook了
                    }else if(got_item == new_fopen){
                        LOGD("[+] Already hooked.\n");
                        break;
                    }
                }
                // Hook目标函数成功，跳出循环
                if(got_found)
                    break;
            }
        }
    }
    free(string_table);
    close(fd);
}

int hook_entry(char* a){
    LOGD("[+] Start hooking.\n");
    hook_fopen();
    return 0;
}
```

运行ndk-build编译，得到libentry.so，push到/data/local/tmp目录下，运行上节所得到的inject程序，得到如下结果，表明hook成功：

```
D/INJECT  ( 4729): [+] Injecting process: 3883
D/INJECT  ( 4756): [+] Injecting process: 3883
D/INJECT  ( 4756): [+]get remote address: local[b6ef5000], remote[40091000]
D/INJECT  ( 4756): [+] Remote mmap address: 400a3d31
D/INJECT  ( 4756): [+]Calling mmap in target process.
D/INJECT  ( 4756): [+] Target process returned from mmap, return value = 7602f000, pc = 0
D/INJECT  ( 4756): [+]get remote address: local[b6f58000], remote[40076000]
D/INJECT  ( 4756): [+]get remote address: local[b6f58000], remote[40076000]
D/INJECT  ( 4756): [+]get remote address: local[b6f58000], remote[40076000]
D/INJECT  ( 4756): [+]get remote address: local[b6f58000], remote[40076000]
D/INJECT  ( 4756): [+] Get imports: dlopen: 40076f4d, dlsym: 40076e9d, dlclose: 40076e19, dlerror: 40076dc9
```

```
D/INJECT  ( 4756): [+]Calling dlopen in target process.
D/INJECT  ( 4756): [+] Target process returned from dlopen, return value = 75ffc24c, pc = 0
D/INJECT  ( 4756): [+]Calling dlsym in target process.
D/INJECT  ( 4756): [+] Target process returned from dlsym, return value = 760d1459, pc = 0
D/INJECT  ( 4756): hooke_entry_addr = 0x760d1459
D/INJECT  ( 4756): [+]Calling hook_entry in target process.
D/INJECT  ( 3883): [+] Start hooking.
D/INJECT  ( 3883): [+] libvivosgmain.so address = 0x75ff0000
D/INJECT  ( 3883): [+] Orig fopen = 0x400a59d9
D/INJECT  ( 3883): [+] String table offset is 41159, size is 254
D/INJECT  ( 3883): [+] out_addr = 75ffaf50, out_size = b0
D/INJECT  ( 3883): [+] Found fopen in got.
D/INJECT  ( 3883): [+] entry page start = 75ffa000, page size = 1000
D/INJECT  ( 3883): [+] before hook function, old_fopen = 400a59d9, new_fopen = 760d10c5
D/INJECT  ( 3883): [+] after hook function, old fopen = 760d10c5, new fopen = 760d10c5
D/INJECT  ( 4756): [+] Target process returned from hook_entry, return value = 0, pc = 0
```

图2.

**方法二**

通过分析program header table查找got表。导入表对应在动态链接段.got.plt（DT_PLTGOT）指向处，但是每项的信息是和GOT表中的表项对应的，因此，在解析动态链接段时，需要解析DT_JMPREL、DT_SYMTAB，前者指向了每一个导入表表项的偏移地址和相关信息，包括在GOT表中偏移，后者为GOT表。

具体代码如下：

```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <android/log.h>
#include <EGL/egl.h>
#include <GLES/gl.h>
#include <elf.h>
#include <fcntl.h>
#include <sys/mman.h>

#define LOG_TAG "INJECT"
#define LOGD(fmt, args...) __android_log_print(ANDROID_LOG_DEBUG, LOG_TAG, fmt, ##args)


//FILE *fopen(const char *filename, const char *modes)
FILE* (*old_fopen)(const char *filename, const char *modes);
FILE* new_fopen(const char *filename, const char *modes){
    LOGD("[+] New call fopen.\n");
    if(old_fopen == -1){
        LOGD("error.\n");
    }
    return old_fopen(filename, modes);
}

void* get_module_base(pid_t pid, const char* module_name){
    FILE* fp;
    long addr = 0;
    char* pch;
    char filename[32];
    char line[1024];

    // 格式化字符串得到 "/proc/pid/maps"
    if(pid < 0){
        snprintf(filename, sizeof(filename), "/proc/self/maps");
    }else{
        snprintf(filename, sizeof(filename), "/proc/%d/maps", pid);
    }

    // 打开文件/proc/pid/maps，获取指定pid进程加载的内存模块信息
    fp = fopen(filename, "r");
    if(fp != NULL){
        // 每次一行，读取文件 /proc/pid/maps中内容
        while(fgets(line, sizeof(line), fp)){
            // 查找指定的so模块
```

```
            if(strstr(line, module_name)){
                // 分割字符串
                pch = strtok(line, "-");
                // 字符串转长整形
                addr = strtoul(pch, NULL, 16);

                // 特殊内存地址的处理
                if(addr == 0x8000){
                    addr = 0;
                }
                break;
            }
        }
    }
    fclose(fp);
    return (void*)addr;
}

#define LIB_PATH "/data/app-lib/com.bbk.appstore-2/libvivosgmain.so"
int hook_fopen(){

    // 获取目标pid进程中"/data/app-lib/com.bbk.appstore-2/libvivosgmain.so"模块的加载地址
    void* base_addr = get_module_base(getpid(), LIB_PATH);
    LOGD("[+] libvivosgmain.so address = %p \n", base_addr);

    // 保存被Hook的目标函数的原始调用地址
    old_fopen = fopen;
    LOGD("[+] Orig fopen = %p\n", old_fopen);

    //计算program header table实际地址
    Elf32_Ehdr *header = (Elf32_Ehdr*)(base_addr);
    if (memcmp(header->e_ident, "\177ELF", 4) != 0) {
        return 0;
    }

    Elf32_Phdr* phdr_table = (Elf32_Phdr*)(base_addr + header->e_phoff);
    if (phdr_table == 0)
    {
        LOGD("[+] phdr_table address : 0");
        return 0;
    }
    size_t phdr_count = header->e_phnum;
    LOGD("[+] phdr_count : %d", phdr_count);


    //遍历program header table, ptype等于PT_DYNAMIC即为dynameic, 获取到p_offset
    unsigned long dynamicAddr = 0;
    unsigned int dynamicSize = 0;
    int j = 0;
    for (j = 0; j < phdr_count; j++)
    {
        if (phdr_table[j].p_type == PT_DYNAMIC)
        {
            dynamicAddr = phdr_table[j].p_vaddr + base_addr;
            dynamicSize = phdr_table[j].p_memsz;
            break;
        }
    }
    LOGD("[+] Dynamic Addr : %x",dynamicAddr);
    LOGD("[+] Dynamic Size : %x",dynamicSize);

/*
typedef struct dynamic {
    Elf32_Sword d_tag;
    union {
    Elf32_Sword d_val;
    Elf32_Addr d_ptr;
    } d_un;
} Elf32_Dyn;
*/
    Elf32_Dyn* dynamic_table = (Elf32_Dyn*)(dynamicAddr);
```

```
116     unsigned long jmpRelOff = 0;
117     unsigned long strTabOff = 0;
118     unsigned long pltRelSz = 0;
119     unsigned long symTabOff = 0;
120     int i;
121     for(i=0;i < dynamicSize / 8;i ++)
122     {
123         int val = dynamic_table[i].d_un.d_val;
124         if (dynamic_table[i].d_tag == DT_JMPREL)
125         {
126             jmpRelOff = val;
127         }
128         if (dynamic_table[i].d_tag == DT_STRTAB)
129         {
130             strTabOff = val;
131         }
132         if (dynamic_table[i].d_tag == DT_PLTRELSZ)
133         {
134             pltRelSz = val;
135         }
136         if (dynamic_table[i].d_tag == DT_SYMTAB)
137         {
138             symTabOff = val;
139         }
140     }
141
142     Elf32_Rel* rel_table = (Elf32_Rel*)(jmpRelOff + base_addr);
143     LOGD("[+] jmpRelOff : %x",jmpRelOff);
144     LOGD("[+] strTabOff : %x",strTabOff);
145     LOGD("[+] symTabOff : %x",symTabOff);
146     //遍历查找要hook的导入函数，这里以fopen做示例
147     for(i=0;i < pltRelSz / 8;i++)
148     {
149         int number = (rel_table[i].r_info >> 8) & 0xffffff;
150         Elf32_Sym* symTableIndex = (Elf32_Sym*)(number*16 + symTabOff + base_addr);
151         char* funcName = (char*)(symTableIndex->st_name + strTabOff + base_addr);
152         //LOGD("[+] Func Name : %s",funcName);
153         if(memcmp(funcName, "fopen", 5) == 0)
154         {
155             // 获取当前内存分页的大小
156             uint32_t page_size = getpagesize();
157             // 获取内存分页的起始地址（需要内存对齐）
158             uint32_t mem_page_start = (uint32_t)(((Elf32_Addr)rel_table[i].r_offset + base_addr)) & (~(pa
ge_size - 1));
159             LOGD("[+] mem_page_start = %lx, page size = %lx\n", mem_page_start, page_size);
160             //void* pstart = (void*)MEM_PAGE_START(((Elf32_Addr)rel_table[i].r_offset + base_addr));
161             mprotect((uint32_t)mem_page_start, page_size, PROT_READ | PROT_WRITE | PROT_EXEC);
162             LOGD("[+] r_off : %x",rel_table[i].r_offset + base_addr);
163             LOGD("[+] new_fopen : %x",new_fopen);
164             *(unsigned int*)(rel_table[i].r_offset + base_addr) = new_fopen;
165         }
166     }
167
168     return 0;
169 }
170
171 int hook_entry(char* a){
172     LOGD("[+] Start hooking.\n");
173     hook_fopen();
174     return 0;
175 }
```

运行后的结果为：

```
D/INJECT  ( 4441): [+] Injecting process: 4103
D/INJECT  ( 4452): [+] Injecting process: 4103
D/INJECT  ( 4452): [+]get remote address: local[b6f3c000], remote[400a9000]
D/INJECT  ( 4452): [+] Remote mmap address: 400bbd31
D/INJECT  ( 4452): [+]Calling mmap in target process.
D/INJECT  ( 4452): [+] Target process returned from mmap, return value = 75f59000, pc = 0
D/INJECT  ( 4452): [+]get remote address: local[b6f9f000], remote[4008e000]
```

```
D/INJECT  ( 4452): [+]get remote address: local[b6f9f000], remote[4008e000]
D/INJECT  ( 4452): [+]get remote address: local[b6f9f000], remote[4008e000]
D/INJECT  ( 4452): [+] Get imports: dlopen: 4008ef4d, dlsym: 4008ee9d, dlclose: 4008ee19, dlerror: 4008edc9
D/INJECT  ( 4452): [+]Calling dlopen in target process.
D/INJECT  ( 4452): [+] Target process returned from dlopen, return value = 7602424c, pc = 0
D/INJECT  ( 4452): [+]Calling dlsym in target process.
D/INJECT  ( 4452): [+] Target process returned from dlsym, return value = 75fb92e1, pc = 0
D/INJECT  ( 4452): hooke_entry_addr = 0x75fb92e1
D/INJECT  ( 4452): [+]Calling hook_entry in target process.
D/INJECT  ( 4103): [+] Start hooking.
D/INJECT  ( 4103): [+] libvivosgmain.so address = 0x76018000
D/INJECT  ( 4103): [+] Orig fopen = 0x400bd9d9
D/INJECT  ( 4103): [+] phdr_count : 9
D/INJECT  ( 4103): [+] Dynamic Addr : 76022e28
D/INJECT  ( 4103): [+] Dynamic Size : 128
D/INJECT  ( 4103): [+] jmpRelOff : 2a88
D/INJECT  ( 4103): [+] strTabOff : d8c
D/INJECT  ( 4103): [+] symTabOff : 18c
D/INJECT  ( 4103): [+] mem_page_start = 76022000, page size = 1000
D/INJECT  ( 4103): [+] r_off : 76022fcc
D/INJECT  ( 4103): [+] new_fopen : 75fb8fb5
D/INJECT  ( 4452): [+] Target process returned from hook_entry, return value = 0, pc = 0
```

图3

参考文章:

http://gslab.qq.com/portal.php?mod=view&aid=169

https://blog.csdn.net/u011247544/article/details/78668791

https://blog.csdn.net/qq1084283172/article/details/53942648

分类: Android底层



« 上一篇: Android so注入(inject)和Hook技术学习（一）
» 下一篇: Android so注入(inject)和Hook技术学习（三）——Got表hook之导出表hook

posted @ 2018-07-13 19:54 bamb00 阅读(885) 评论(0) 编辑 收藏