Reload executable files to achieve efficient inline-hook

Preface

In many cases, efficiency optimization is basically time for space and space for time. This solution is to implement a universal and efficient hook based on the traditional inline-hook by spending a little more memory space . The core principle is the overloading of executable modules.

Traditional inline-hook

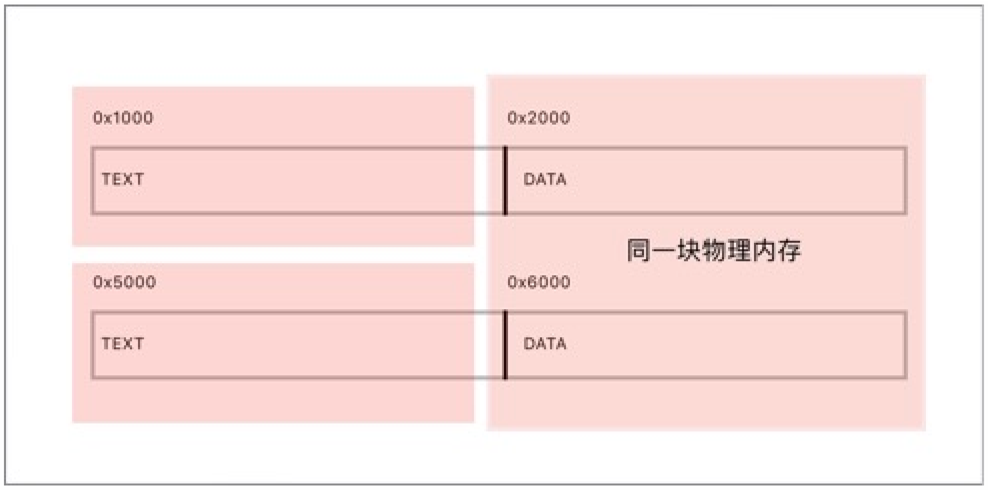
The traditional inline hook is to back up the first n bytes of the target function first , and by modifying the first n bytes of the target function , replacing it with a few instructions, the instruction realizes the function that jumps to the proxy. There are two ways to call the original function. One is to restore the modified bytes before calling, and to hook again after returning . The second is to back up the original bytes to a section of executable memory, jump to the next normal instruction of the code where the original function is modified from the end of the memory, and directly call the code in the memory here. The disadvantage of the first method is low efficiency, and special processing is required for concurrency. The disadvantage of the second method is that when the current n bytes contain a local jump instruction, if the offset is not corrected, it will cause an execution error. It is troublesome and difficult to correct the offset, and different processor architectures need to be dealt with separately.

New hook scheme

The new hook scheme has the following advantages: high efficiency, no additional processing is required when calling the original function; simple and general, no modification instructions are required, and no separate processing of the processor architecture is required. The disadvantage is that it takes up more memory. The amount of memory occupied depends on the size of the hooked module. If some optimizations are performed, this memory footprint can still be reduced. The environmental requirements for the new hook scheme are as follows: support mmap , mprotect , and support shared memory. The requirement for shared

memory is not absolutely necessary, as long as the same segment of memory can be mapped to different memory addresses.

Implementation details



Executable files are usually mapped to memory. Executable files generally have several sections, such as text code section and data data section. The code segment is readable and executable, and the data segment is readable and writable. The addressing of the code in the text section to the data in the data section is relative addressing. Before hooking in this program , the memory image of the entire executable file will be cloned to another memory. The new code segment is an ordinary writable and executable code segment, and the new data segment is somewhat special. The new data segment is a remap of the old data segment . The relationship is that although the old and new data segments are at different memory addresses, they point to the same physical memory by some means. Intuitively speaking, the reading and writing of the new and old data segments actually present the same content. The purpose is to synchronize global variables.

When hooking , just insert the code needed for the inline hook into the old code . When you need to call the original function, just call the original function in the new code segment. In this process, there is basically no loss in execution speed.

Code

This example uses shared memory to map a section of memory to two different addresses

```
#define SET_PROXY(c,f) *((uint64_t*)(((uint8_t*)(c))+2)) = (uint64_t)f

void *ih_hook(void *oldFunc, void *newFunc, void **save) {
    unsigned char hook_stub_code_rax[] = {
        0x48, 0xB8, 0x88, 0x77, 0x66, 0x55, 0x44, 0x33, 0x22, 0x11, //movabsq
0x1122334455667788, %rax  10
        0xff, 0xe0,   //jmp *%rax 2
        0x90,
    };

    unsigned char hook_stub_code_r11[] = {
        0x49, 0xBB, 0x88, 0x77, 0x66, 0x55, 0x44, 0x33, 0x22, 0x11, //movabsq
0x1122334455667788, %r11  10
        0x41, 0xff, 0xe3,    //jmp *%11 3
    };

    void *image_ptr = NULL;
    size_t image_size = 0;
    void *data_ptr = NULL;
    size_t data_size = 0;

// 获取可执行文件的信息
 if (get_addr_info(oldFunc, &image_ptr, &image_size, &data_ptr, &data_size) != 0){
        return  NULL;
    }

    // 创建共享内存对象
    const char *shm_file = "/tmp/hook.shm";
    int fd = shm_open(shm_file, O_RDWR | O_CREAT, S_IRUSR | S_IWUSR);
    if (fd < 0){
        printf("Error: shm_open\n");
        return NULL;
    }
    ftruncate(fd, image_size);

    // 创建新镜像内存
    void *new_image_ptr = mmap(NULL, image_size, PROT_READ | PROT_WRITE, MAP_ANON |
MAP_PRIVATE, -1, 0);
    if (new_image_ptr == MAP_FAILED){
        shm_unlink(shm_file);
        printf("Error: mmap for new_image_ptr: %s\n", strerror(errno));
        return  NULL;
    }

    // 拷贝数据到新内存
    memcpy(new_image_ptr, image_ptr, image_size);
    if (mprotect(new_image_ptr, image_size, PROT_EXEC|PROT_READ) != 0){
        munmap(new_image_ptr, image_size);
        printf("Error: mprotect: %s\n", strerror(errno));
        return NULL;
    }

    // 将旧数据段覆盖为共享内存
    void *shm1 = mmap(data_ptr, data_size, PROT_READ | PROT_WRITE,
MAP_SHARED|MAP_FILE|MAP_FIXED, fd, 0);
    my_memcpy(shm1, (uint8_t*)new_image_ptr + (data_ptr - image_ptr), data_size);

    // 将新数据段覆盖为共享内存
    void *shm2 = mmap((uint8_t*)new_image_ptr + (data_ptr - image_ptr), data_size,
PROT_READ | PROT_WRITE, MAP_SHARED|MAP_FILE|MAP_FIXED, fd, 0);
    my_memcpy(shm2, data_ptr, data_size);

    // 做内联hook
    SET_PROXY(hook_stub_code_r11, newFunc);
    write_memory(oldFunc, hook_stub_code_r11, sizeof(hook_stub_code_r11));

    // 计算原函数地址，原函数位于新代码段
    *save = (uint8_t*)new_image_ptr + (oldFunc - image_ptr);;

//
//    munmap(new_image_ptr, image_size);
//    //    munmap(shm1, data_size);
//    munmap(shm2, data_size);
//
//    shm_unlink(shm_file);
    return NULL;
}

int (*org_puts)(const char *s);

int proxy_puts(const char *s) {
    printf("Hook successful\n");
    org_puts(s);
    return 0;
}

int main(int argc, const char * argv[]) {

    ih_hook(puts, proxy_puts, (void**)&org_puts);
    puts("Normal");
    printf("Hello, World!\n");
    return 0;
}
```

```
Hook successful
Normal
Hello, World!
```