[T3-4-2]热补丁

- 热补丁是一种在程序运行时动态修补安全漏洞的技术，这种修补不需要重启操作系统或应用程序，因此能够大大增强系统的可用性。根据修补对象的不同，热补丁技术可以分为应用程序热补丁和系统内核热补丁两类，它们的技术原理是类似的，但具体的实现细节有所不同.

[T3-4-2]热补丁技术一般三个步骤

- 首先，对程序中存在的安全漏洞进行详细的分析，明确漏洞成因，在此基础上编写相应的代码，并编译出可动态加载的补丁文件。
- 其次，通过加载程序将第一步得到的补丁文件加载到目标程序的内存空间，对于同一个系统，加载程序可以是通用的，补丁文件则因安全漏洞而异。
- 最后，修改程序的执行流程，把存在安全漏洞的代码替换为新的代码，完成热补丁的修补。

[T3-4-2]热补丁

- 热补丁技术的关键在于补丁文件的加载和程序执行流程的修改，工程上通常借助钩子技术（hook）来实现。钩子技术通过拦截系统调用、消息或事件，得到对系统进程或消息的控制权，进而改变或增强程序的行为。主流操作系统，如Windows和Linux，都提供了hook的相应机制，并已被广泛运用到热补丁及代码调试等场景中。

A应用程序热补丁-Preload Hook

- Preload Hook是指利用操作系统对预加载（preload）的支持，将外部程序模块自动注入到指定的进程中的一种钩子技术—动态链接的程序有用，静态的无效，优先加载。
- Preload Hook有两种常见的用法：一种是配置环境变量LD_PRELOAD（SUID或SGID位被置1，加载的时候会忽略LD_PRELOAD），另一种是配置文件/etc/ld.so.preload。对于配置环境变量LD_PRELOAD，通过命令行指定LD_PRELOAD将仅影响当前新进程及其子进程，写入全局环境变量则将影响所有新进程，但新进程的父进程可以控制子进程的环境变量从而取消preload。

样例

- 初始程序源码

original.c

```c
#include <stdio.h>
#include <unistd.h>

int main()
{
    puts("A sample of preload hook.");
    sleep(2);
    puts("The end.");
    return 0;
}
```

- 补丁源码

patch.c

```c
#include <stdio.h>

int sleep(int t)
{
    puts("Your sleep() is hook by xxx.");
}
```

样例

补丁文件编译为：

```
root@DESKTOP-C6VM2Q8:/home/project/pre# gcc -m32 -fPIC --shared patch.c -o patch.so
```

通过命令行进行Preload Hook：

```
root@DESKTOP-C6VM2Q8:/home/project/pre# LD_PRELOAD=./patch.so ./original
A sample of preload hook.
Your sleep() is hook by xxx.
The end.
root@DESKTOP-C6VM2Q8:/home/project/pre#
```

可以看到已经成功地将系统的sleep函数修改成了自定义的功能。原理：loader在进行动态链接的时候，会将有相同符号名的符号覆盖成LD_PRELOAD指定的so文件中的符号。换句话说，可以用自己的so库中的函数替换原来库里有的函数，从而达到hook的目的。这和Windows下通过修改import table来hook API很类似。相比较之下，LD_PRELOAD更方便了，不用自己写代码了，系统的loader会直接调用。通过这个可以启动恶意代码，或劫持、监控软件等。

热补丁的完整实现

- 利用Preload Hook虽然可以进行补丁修补，但它还不能算是真正的热补丁，因为对于已经处于运行状态的应用程序，这种方法是无法生效的。真正的热补丁必须通过专门的加载程序，利用动态的hook机制来实现补丁文件的加载和程序执行流程的修改。下面以Linux系统为例，介绍加载程序的编写及热补丁的完整实现。

original.c

```c
#include <stdio.h>
#include <unistd.h>
#include <time.h>

int main()
{
    while(1){
        sleep(5);
        printf("original: %ld\n", time(0));
    }
    return 0;
}
```

补丁源码

patch.c

```c
#include <stdio.h>

int newprintf()
{
    puts("My student number is xxx.");
    return 0;
}
```

补丁文件编译

```
root@DESKTOP-C6VM2Q8:/home/project/hotfix# gcc -m32 -fPIC --shared patch.c -o patch.so
```

热补丁完整实现--第一步：关联

- Linux系统提供了一种专门用于程序调试的系统调用ptrace，热补丁的加载程序可以借助ptrace对运行状态的应用程序进行hook，并最终实现热补丁修补。具体分为以下五个步骤。
- 第一步：加载程序通过ptrace关联（attach）到需要修补的进程上，并将该进程的寄存器及内存数据保存下来，代码如下：

```c
/* 关联到进程 */
void ptrace_attach(int pid)
{
    if(ptrace(PTRACE_ATTACH, pid, NULL, NULL) < 0) {
        perror("ptrace_attach");
        exit(-1);
    }

    waitpid(pid, NULL, /*WUNTRACED*/0);

    ptrace_readreg(pid, &oldregs);
}
```

第二步-定位：ELF文件及link_map

- 找到需要补丁的可执行程序的elf文件（是一种用于二进制文件、可执行文件、目标代码、共享库和核心转储格式文件。是UNIX系统实验室（USL）作为应用程序二进制接口（Application Binary Interface，ABI）而开发和发布的，也是Linux的主要可执行文件格式。https://blog.csdn.net/kang___xi/article/details/79571137）
- 再找到指向elf文件的link_map链表的指针，并通过遍历link_map中的符号表，找到需要修补的（比如：printf）函数及用于将补丁文件加载到进程内存地址空间的__libc_dlopen_mode函数的地址。根据elf文件的结构信息，首先从elf文件的开头开始读取信息，找到头部表（program header table），再根据头部表找到elf文件的全局偏移量表（global offset table，GOT表）。GOT表中的每一项都是一个32bit的Elf32_Addr地址，其中前两项是两个特殊的数据结构的地址：
- GOT[0]为PT_DYNAMIC的起始地址
- GOT[1]为link_map结构体的地址
- 由此可以得到指向link_map链表的指针，代码如下：

格式代码如下：

```
#define EI_NIDENT 16
typedef struct{
    unsigned char e_ident[EI_NIDENT];
    Elf32_Half e_type;
    Elf32_Half e_machine;
    Elf32_Word e_version;
    Elf32_Addr e_entry;
    Elf32_Off e_phoff;
    Elf32_Off e_shoff;
    Elf32_Word e_flags;
    Elf32_Half e_ehsize;
    Elf32_Half e_phentsize;
    Elf32_Half e_phnum;
    Elf32_Half e_shentsize;
    Elf32_Half e_shnum;
    Elf32_Half e_shstrndx;
} Elf32_Ehdr;
```

第二步：找到被补的可执行成的函数

```c
/*
得到指向link_map链表首项的指针
*/
struct link_map *get_linkmap(int pid)
{
    Elf32_Ehdr *ehdr = (Elf32_Ehdr *) malloc(sizeof(Elf32_Ehdr));
    Elf32_Phdr *phdr = (Elf32_Phdr *) malloc(sizeof(Elf32_Phdr));
    Elf32_Dyn  *dyn =  (Elf32_Dyn *) malloc(sizeof(Elf32_Dyn));
    Elf32_Word got;
    struct link_map *map = (struct link_map *)malloc(sizeof(struct link_map));
    int i = 1;
    unsigned long tmpaddr;

    ptrace_read(pid, IMAGE_ADDR, ehdr, sizeof(Elf32_Ehdr));
    phdr_addr = IMAGE_ADDR + ehdr->e_phoff;
    printf("phdr_addr\t %p\n", phdr_addr);

    ptrace_read(pid, phdr_addr, phdr, sizeof(Elf32_Phdr));
    while(phdr->p_type != PT_DYNAMIC)
        ptrace_read(pid, phdr_addr += sizeof(Elf32_Phdr), phdr,sizeof(Elf32_Phdr));
    dyn_addr = phdr->p_vaddr;
    printf("dyn_addr\t %p\n", dyn_addr);

    ptrace_read(pid, dyn_addr, dyn, sizeof(Elf32_Dyn));
    while(dyn->d_tag != DT_PLTGOT) {
        tmpaddr = dyn_addr + i * sizeof(Elf32_Dyn);
        //printf("get_linkmap tmpaddr = %x\n",tmpaddr);
        ptrace_read(pid,tmpaddr, dyn, sizeof(Elf32_Dyn));
        i++;
    }

    got = (Elf32_Word)dyn->d_un.d_ptr;
    got += 4;
    //printf("GOT\t\t %p\n", got);

    ptrace_read(pid, got, &map_addr, 4);
    printf("map_addr\t %p\n", map_addr);
    map = map_addr;
    //ptrace_read(pid, map_addr, map, sizeof(struct link_map));

    free(ehdr);
    free(phdr);
    free(dyn);

    return map;
}
```

遍历link_map链表，依次对每一个link_map调用find_symbol_in_linkmap函数：

find_symbol_in_linkmap函数负责在指定的link_map中查找所需要的函数的地址：

```c
/*
解析指定符号
*/
unsigned long find_symbol(int pid, struct link_map *map, char *sym_name)
{
    struct link_map *lm = map;
    unsigned long sym_addr;
    char *str;
    unsigned long tmp;

    sym_addr = find_symbol_in_linkmap(pid, lm, sym_name);
    while(!sym_addr ) {
        ptrace_read(pid, (char *)lm+12, &tmp, 4);   //获取下一个库的link_map地址
        if(tmp == 0)
            return 0;
        lm = tmp;

        if ((sym_addr = find_symbol_in_linkmap(pid, lm, sym_name)))
            break;
    }

    return sym_addr;
}
```

第三步—加载补丁程序

调用__libc_dlopen_mode函数，将补丁文件加载到需要修补的进程的内存空间中，并再一次遍历link_map中的符号表，找到新加载的补丁文件中的新函数newprintf的地址，代码如下：

```
/*
在指定的link_map所指向的符号表中查找符号
*/
unsigned long find_symbol_in_linkmap(int pid, struct link_map *lm, char *sym_name)
{
    Elf32_Sym *sym = (Elf32_Sym *) malloc(sizeof(Elf32_Sym));
    int i = 0;
    char *str;
    unsigned long ret;
    int flags = 0;

    get_sym_info(pid, lm);

    do{
        if(ptrace_read(pid, symtab + i * sizeof(Elf32_Sym), sym, sizeof(Elf32_Sym)))
            return 0;
        i++;
        if (!sym->st_name && !sym->st_size && !sym->st_value)   //全为0是符号表的第一项
            continue;
        str = (char *) ptrace_readstr(pid, strtab + sym->st_name);
        if (strcmp(str, sym_name) == 0) {
            printf("\nfind_symbol_in_linkmap str = %s\n",str);
            printf("\nfind_symbol_in_linkmap sym->st_value = %x\n",sym->st_value);
            free(str);
            if(sym->st_value == 0)  //值为0代表这个符号本身就是重定向的内容
                continue;
            flags = 1;
            break;
        }
        free(str);
    }while(1);

    if (flags != 1)
        ret = 0;
    else
        ret =  link_addr + sym->st_value;

    free(sym);

    return ret;
}
```

第四步—将被补函数的地址改为补丁函数的地址

```
/* 发现__libc_dlopen_mode，并调用它 */
sym_addr = find_symbol(pid, map, "__libc_dlopen_mode");        /* call _dl_open */
printf("found __libc_dlopen_mode at addr %p\n", sym_addr);
if(sym_addr == 0)
    goto detach;
call__libc_dlopen_mode(pid, sym_addr,libpath);    /* 注意装载的库地址 */
waitpid(pid,&status,0);
/* 找到新函数的地址 */
strcpy(sym_name, newfunname);                    /* intercept */
sym_addr = find_symbol(pid, map, sym_name);
printf("%s addr\t %p\n", sym_name, sym_addr);
if(sym_addr == 0)
    goto detach;
```

找到要修补的函数printf的重定向地址，在该地址填入补丁文件中的新函数newprintf的地址代码如上

```
/* 找到旧函数在重定向表的地址 */
strcpy(sym_name, oldfunname);
rel_addr = find_sym_in_rel(pid, sym_name);
printf("%s rel addr\t %p\n", sym_name, rel_addr);
if(rel_addr == 0)
    goto detach;

/* 函数重定向 */
puts("intercept...");                    /* intercept */
if(modifyflag == 2)
    sym_addr = sym_addr - rel_addr - 4;
printf("main modify sym_addr = %x\n",sym_addr);
```

第五步-收尾

- 完成patch，恢复现场，脱离需要修补的进程代码如下。

```
    ptrace_write(pid, rel_addr, &sym_addr, sizeof(sym_addr));
    puts("patch ok");
detach:
    printf("prepare to detach\n");
    ptrace_detach(pid);

    return 0;
```

初始程序的执行效果：

```
root@DESKTOP-C6VM2Q8:/home/project/hotfix# ./original
original: 1622280172
original: 1622280177
```

通过加载程序打上热补丁后的执行效果：

```
root@DESKTOP-C6VM2Q8:/home/project/hotfix# ./original
original: 1622281125
original: 1622281130
original: 1622281135
original: 1622281140
patch successed
My student number is xxx.
My student number is xxx.
My student number is xxx.
My student number is xxx.
```

应用热补丁

热补丁加载程序运行情况：

```
root@DESKTOP-C6VM2Q8:/home/project/hotfix# ./hotfix 11331 ./patch.so printf newprintf
main pid = 11331
main libpath : ./patch.so
main oldfunname : printf
main newfunname : newprintf
phdr_addr    0x8048034
dyn_addr     0x8049f14
map_addr     0xf7f82940

find_symbol_in_linkmap str = printf
find_symbol_in_linkmap sym->st_value = 50c60
found printf at addr 0xf7dbcc60
get_sym_info exit

find_symbol_in_linkmap str = __libc_dlopen_mode
find_symbol_in_linkmap sym->st_value = 131a90
found __libc_dlopen_mode at addr 0xf7e9da90
get_sym_info exit

find_symbol_in_linkmap str = newprintf
find_symbol_in_linkmap sym->st_value = 4bd
newprintf addr   0xf7f514bd
get_sym_info exit

printf rel addr  0x804a00c
intercept...
main modify sym_addr = f7f514bd
patch ok
prepare to detach
```

内核级热补丁

- 与应用程序热补丁相比，系统内核热补丁需要考虑的因素更多，对稳定性和可靠性的要求也更高。因此，各大操作系统厂商都推出了自己的系统内核热补丁方案。下面介绍几种常见的Linux内核热补丁方案。

内核级热补丁

- **Ksplice**是MIT的Jeff Arnold在2008年发明的，是第一个成熟的Linux内核热补丁方案，并于2011年被Oracle收购。它的核心原理是构建新、老两个系统内核，并在此基础上比较二进制代码的差异。对于存在差异的函数，Ksplice会调用stop_machine_run函数来短暂地停止其所在的进程及其线程，并迅速地在原始函数的开头处插入一段跳转代码，将其指向修补后的新函数。之后，Linux 内核将执行修补后的新函数。
- **Kgraft**由SUSE在2014年提出。它的原理与Ksplice相近，都是通过差异比较来确定内核中需要修补的函数。但是，与Ksplice不同，Kgraft不会调用stop_machine_run函数来暂时停止进程或线程，而是使用了处理器间非可屏蔽中断机制来完成新、老内核的切换。同时，Kgraft能够为用户进程、内核线程和中断处理程序提供始终一致的视图，这样，老的内核模块将始终调用另一个老的内核模块，而修补后的新内核模块将始终调用另一个新内核模块，从而避免系统在热补丁修补过程中出现混乱。
- **Kpatch**是Redhat在2014年提出的系统内核热补丁方案，它基于ftrace来hook原始函数的mcount调用指令，从而将对原始函数的调用重定向到新的函数。使用Kpatch的系统内核在编译时会在每个函数的入口处保留若干字节，这样，在进行热补丁修补时只需要将待修补的函数入口处保留的字节替换为跳转指令，跳转到Kpatch的相关流程中，即可实现函数级别的执行流程在线替换，最终进入修补后的新函数的执行流程。

补丁的方法

- **硬盘上PE文件的Characteristics和FirstThunk指向2个IMAGE_THUNK_DATA结构，当加载到内存后，PE装载器通过OriginalFirstThunk找到IMAGE_THUNK_DATA再找到IMAGE_IMPORT_BY_NAME，**
- **根据shell32dll.shellaboutA信息，计算出函数的真实地址， 把真实地址填写到FirstThunk->IMAGE_THUNK_DATA中。 PS:** 函数的真实地址 =dll加载基地址+函数RAV. 这个地址是PE加载器算出来的，PE文件不保存函数的RAV.
- EXE程序加载后，FirstThunk指向的为函数的真实地址。
- APIHOOK替换地址就是 *FirstThunk， CALL DWORD ptr[FirstThunk]
- IMAGE_IMPORT_DESCRIPTOR STRUCT 20个字节
- union Characteristics DWORD ? //固定的IMAGE_THUNK_DATA结构的地址
- OriginalFirstThunk DWORD ?
- ends TimeDateStamp DWORD ?
- ForwarderChain DWORD ? Name DWORD ?
- FirstThunk DWORD ? //会被PE加载器重写的MAGE_THUNK_DATA地址，PE装载器加载exe后，把这个IMAGE_THUNK_DATA改写dll里 函数的真实地址IMAGE_IMPORT_DESCRIPTOR ENDS IMAGE_THUNK_DATA STRUC 4个字节 union u1 ForwarderString DWORD ? ; 指向一个转向者字符串的RVA Function DWORD ? ; 被输入的函数的内存地址 Ordinal DWORD ? ; 被输入的API 的序数值 AddressOfData DWORD ? ; 指向IMAGE_IMPORT_BY_NAME ends IMAGE_THUNK_DATA ENDS