tags: OS

HW10

408410094 資工三 葉X勛

1. 撰寫程式碼稱之為myls，在程式碼中使用execve系列的任何libc函數，載入新的執行檔案(ls)。
2. 請問作業系統如何載入執行檔案?
3. 請問作業系統是否立即載入檔案到記憶體?

1.

myls.c > ls()

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

void ls()
{
    char *argv[ ]={"ls", NULL};
    char *envp[ ]={"PATH=/bin", NULL};
    execve("/bin/ls", argv, envp);
}

int main()
{
    printf("%p\n", ls);
    getchar();
    ls();

}
```

2.

(1)首先將中斷點設在do_execve()，追到此函數後發現他只是將字串轉成struct user_arg_ptr，然後呼叫do_execveat_common。

```
int do_execve(struct filename *filename,
    const char __user *const __user *__argv,
    const char __user *const __user *__envp)
{
    struct user_arg_ptr argv = { .ptr.native = __argv };
    struct user_arg_ptr envp = { .ptr.native = __envp };
    return do_execveat_common(AT_FDCWD, filename, argv, envp, 0);
}
```

(2)追進do_execveat_common()，發現他只做了回傳__do_execve_file()的動作。

```
static int do_execveat_common(int fd, struct filename *filename,
                  struct user_arg_ptr argv,
                  struct user_arg_ptr envp,
                  int flags)
{
    return __do_execve_file(fd, filename, argv, envp, flags, NULL);
}
```

(3)追進__do_execve_file()，發現他的程式碼有很多行，以下我將分成好幾部分來解釋這個函數在做什麼。 第一部分:呼叫IS_ERR()判斷檔名是否有誤，有誤的話直接回傳PTR_ERR。

```
*/
static int __do_execve_file(int fd, struct filename *filename,
                struct user_arg_ptr argv,
                struct user_arg_ptr envp,
                int flags, struct file *file)
{
    char *pathbuf = NULL;
    struct linux_binprm *bprm;
    struct files_struct *displaced;
    int retval;

    if (IS_ERR(filename))
        return PTR_ERR(filename);
```

(4)第二部分:註解寫到他們將程序數超過限制的處理從set*uid()搬到了execve()，且沒有對setuid()的回傳值進行檢查了。除此之外還檢查了Process數是否有超過。

```
    /*
     * We move the actual failure in case of RLIMIT_NPROC excess from
     * set*uid() to execve() because too many poorly written programs
     * don't check setuid() return code.  Here we additionally recheck
     * whether NPROC limit is still exceeded.
     */
    if ((current->flags & PF_NPROC_EXCEEDED) &&
        atomic_read(&current_user()->processes) > rlimit(RLIMIT_NPROC)) {
        retval = -EAGAIN;
        goto out_ret;
    }
```

(5)第三部分:呼叫unshare_files()，拷貝當前運行process的fd到displaced中。然後用kzalloc()配置一個bprm(二進位程式結構)的記憶體，再用prepare_bprm_creds()進一步準備此程式的權限相關結構。

```
    /* We're below the limit (still or again), so we don't want to make
     * further execve() calls fail. */
    current->flags &= ~PF_NPROC_EXCEEDED;

    retval = unshare_files(&displaced);
    if (retval)
        goto out_ret;

    retval = -ENOMEM;
    bprm = kzalloc(sizeof(*bprm), GFP_KERNEL);
    if (!bprm)
        goto out_files;

    retval = prepare_bprm_creds(bprm);
    if (retval)
        goto out_free;
```

(6)第四部分:呼叫check_unsafe_exec()檢查bprm安全性，檢查完後標註current process正在execve的狀態。然後呼叫do_open_execat()打開要執行的文件，之後呼叫sched_exec()，做負載平衡的調整。

```
    check_unsafe_exec(bprm);
    current->in_execve = 1;

    if (!file)
        file = do_open_execat(fd, filename, flags);
    retval = PTR_ERR(file);
    if (IS_ERR(file))
        goto out_unmark;

    sched_exec();
```

(7)第五部分:判斷檔案路徑為絕對路徑或是相對路徑，做相應的處理，然後填入bprm內的filename，再將此值再次填入給bprm內的interp。

```
bprm->file = file;
if (!filename) {
    bprm->filename = "none";
} else if (fd == AT_FDCWD || filename->name[0] == '/') {
    bprm->filename = filename->name;
} else {
    if (filename->name[0] == '\0')
        pathbuf = kasprintf(GFP_KERNEL, "/dev/fd/%d", fd);
    else
        pathbuf = kasprintf(GFP_KERNEL, "/dev/fd/%d/%s",
                  fd, filename->name);
    if (!pathbuf) {
        retval = -ENOMEM;
        goto out_unmark;
    }
    /*
     * Record that a name derived from an O_CLOEXEC fd will be
     * inaccessible after exec. Relies on having exclusive access to
     * current->files (due to unshare_files above).
     */
    if (close_on_exec(fd, rcu_dereference_raw(current->files->fdt)))
        bprm->interp_flags |= BINPRM_FLAGS_PATH_INACCESSIBLE;
    bprm->filename = pathbuf;
}
bprm->interp = bprm->filename;
```

(8)第六部分:呼叫bprm_mm_init()配置此執行檔的記憶體，呼叫prepare_arg_pages以及prepare_binprm正式的將bprm設定好，然後根據指定的檔案載入ELF的標頭或是script的直譯器，後面就只是在做字串的copy而已。而

後面呼叫的exec_binprm()只是為了proc檔案系統做資料上的紀錄而已，並非真正執行新程式的地方。

```
    retval = bprm_mm_init(bprm);
    if (retval)
        goto out_unmark;

    retval = prepare_arg_pages(bprm, argv, envp);
    if (retval < 0)
        goto out;

    retval = prepare_binprm(bprm);
    if (retval < 0)
        goto out;

    retval = copy_strings_kernel(1, &bprm->filename, bprm);
    if (retval < 0)
        goto out;

    bprm->exec = bprm->p;
    retval = copy_strings(bprm->envc, envp, bprm);
    if (retval < 0)
        goto out;

    retval = copy_strings(bprm->argc, argv, bprm);
    if (retval < 0)
        goto out;

    would_dump(bprm, bprm->file);

    retval = exec_binprm(bprm);
    if (retval < 0)
        goto out;
```

(9)第七部分:一些善後的程式碼，到這裡這個函數就結束了。

```
    /* execve succeeded */
    current->fs->in_exec = 0;
    current->in_execve = 0;
    membarrier_execve(current);
    rseq_execve(current);
    acct_update_integrals(current);
    task_numa_free(current);
    free_bprm(bprm);
    kfree(pathbuf);
    if (filename)
        putname(filename);
    if (displaced)
        put_files_struct(displaced);
    return retval;

out:
    if (bprm->mm) {
        acct_arg_size(bprm, 0);
        mmput(bprm->mm);
    }

out_unmark:
    current->fs->in_exec = 0;
    current->in_execve = 0;

out_free:
    free_bprm(bprm);
    kfree(pathbuf);

out_files:
    if (displaced)
        reset_files_struct(displaced);
out_ret:
    if (filename)
        putname(filename);
    return retval;
}
```

結論:可以發現作業系統是透過do_execve()、do_execveat_common()以及__do_execve_file()來載入執行檔案的。

3.

以下為我找到在execve()中會對mm_strcut進行動作的函數。 (1)**bprm_mm_init()**:先分配了一個struct mm_struct的記憶體空間，用來存放有關process的相關訊息。

```
static int bprm_mm_init(struct linux_binprm *bprm)
{
    int err;
    struct mm_struct *mm = NULL;

    bprm->mm = mm = mm_alloc();
    err = -ENOMEM;
    if (!mm)
        goto err;

    /* Save current stack limit for all calculations made during exec.
    task_lock(current->group_leader);
    bprm->rlim_stack = current->signal->rlim[RLIMIT_STACK];
    task_unlock(current->group_leader);

    err = __bprm_mm_init(bprm);
    if (err)
        goto err;

    return 0;
```

(2)**exec_binprm()**:在__do_execve_file內後面呼叫了此函數，而此函數又呼叫了search_binary_handler()。

```
static int exec_binprm(struct linux_binprm *bprm)
{
    pid_t old_pid, old_vpid;
    int ret;

    /* Need to fetch pid before load_binary changes it */
    old_pid = current->pid;
    rcu_read_lock();
    old_vpid = task_pid_nr_ns(current, task_active_pid_ns(current->pa
    rcu_read_unlock();

    ret = search_binary_handler(bprm);
    if (ret >= 0) {
        audit_bprm(bprm);
        trace_sched_process_exec(current, old_pid, bprm);
        ptrace_event(PTRACE_EVENT_EXEC, old_vpid);
        proc_exec_connector(current);
    }

    return ret;
}
```

(3)**search_binary_handler()**:在一個list中尋找可識別的可執行文件檔案，找到相對應的文件格式，並調用其load_binary()。

```c
int search_binary_handler(struct linux_binprm *bprm)
{
    bool need_retry = IS_ENABLED(CONFIG_MODULES);
    struct linux_binfmt *fmt;
    int retval;

    /* This allows 4 levels of binfmt rewrites before failing hard. */
    if (bprm->recursion_depth > 5)
        return -ELOOP;

    retval = security_bprm_check(bprm);
    if (retval)
        return retval;

    retval = -ENOENT;
 retry:
    read_lock(&binfmt_lock);
    list_for_each_entry(fmt, &formats, lh) {
        if (!try_module_get(fmt->module))
            continue;
        read_unlock(&binfmt_lock);
        bprm->recursion_depth++;
        retval = fmt->load_binary(bprm);
        read_lock(&binfmt_lock);
        put_binfmt(fmt);
        bprm->recursion_depth--;
        if (retval < 0 && !bprm->mm) {
            /* we got to flush_old_exec() and failed after it */
            read_unlock(&binfmt_lock);
            force_sigsegv(SIGSEGV, current);
            return retval;
        }
```

(4)**load_elf_binary()**:調用到此函數，發現在裡面會修改當前task_struct中的mm_struct內成員的值，因此可確定作業系統並沒有立即載入執行檔案。

```c
static int load_elf_binary(struct linux_binprm *bprm)
{
    struct file *interpreter = NULL; /* to shut gcc up */
    unsigned long load_addr = 0, load_bias = 0;
    int load_addr_set = 0;
    char * elf_interpreter = NULL;
    unsigned long error;
    struct elf_phdr *elf_ppnt, *elf_phdata, *interp_elf_phdata = NULL;
    unsigned long elf_bss, elf_brk;
    int bss_prot = 0;
    int retval, i;
    unsigned long elf_entry;
    unsigned long interp_load_addr = 0;
    unsigned long start_code, end_code, start_data, end_data;
    unsigned long reloc_func_desc __maybe_unused = 0;
    int executable_stack = EXSTACK_DEFAULT;
    struct pt_regs *regs = current_pt_regs();
    struct {
        struct elfhdr elf_ex;
        struct elfhdr interp_elf_ex;
    } *loc;
    struct arch_elf_state arch_state = INIT_ARCH_ELF_STATE;
    loff_t pos;

    loc = kmalloc(sizeof(*loc), GFP_KERNEL);
    if (!loc) {
        retval = -ENOMEM;
        goto out_ret;
    }
    /* N.B. passed_fileno might not be initialized? */
    current->mm->end_code = end_code;
    current->mm->start_code = start_code;
    current->mm->start_data = start_data;
    current->mm->end_data = end_data;
    current->mm->start_stack = bprm->p;
```

結論:否，作業系統載入時只是修改當前task_struct中的mm_struct而已，並沒有立即載入執行檔案。