list_head 数据结构

list_head 是一个双向链表结构，里面没有数据域，没有在链表结构中包含数据，而是在描述数据类型的数据结构中包含链表，从而保证链表的通用性。

```
struct list_head {
    struct list_head *next, *prev;
};
```

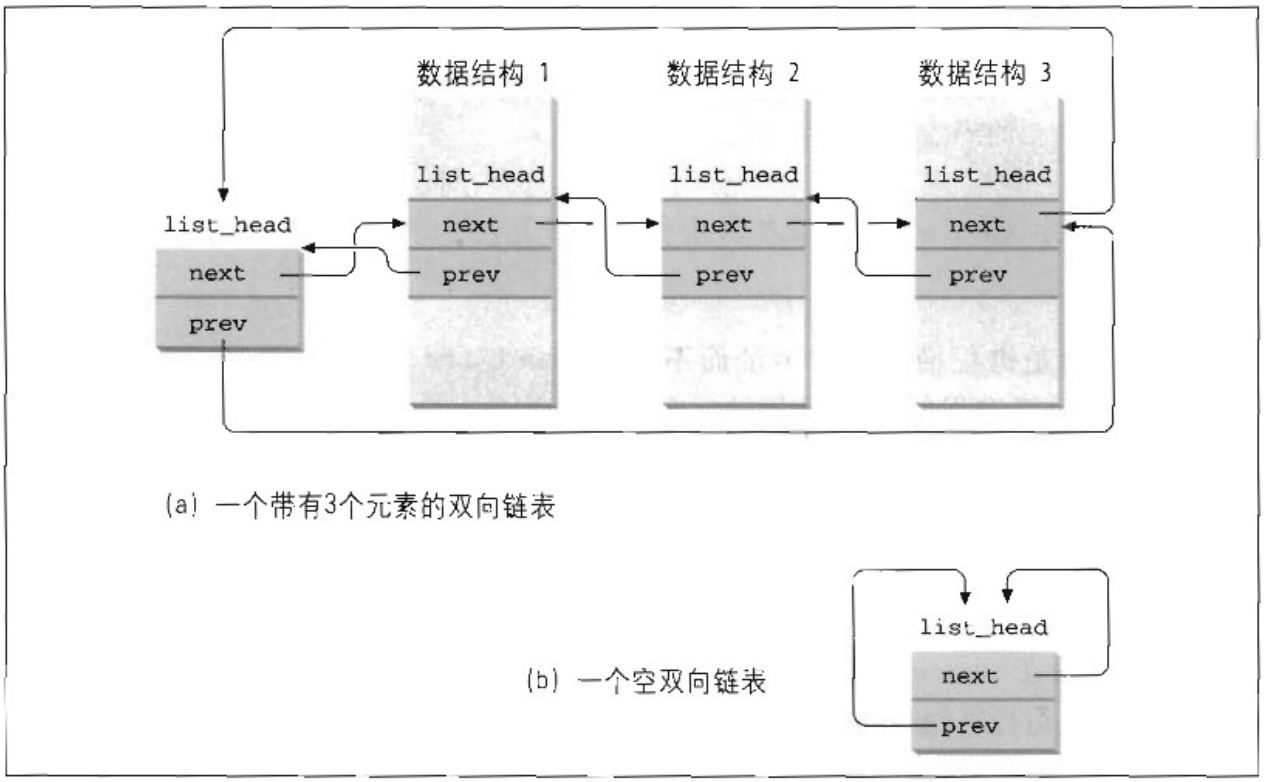


图 3-3：用 list_head 数据结构构造的一个双向链表

以内核中进程描述符结构为例：

```
struct task_struct {
    ...
    struct list_head tasks;
    ...
}
```

结构体初始化

在标准 C 里，数组或结构体变量的初始化值必须以固定的顺序出现，在 GCC 中，通过指定索引或结构域名，则允许初始化值以任意顺序出现。

```
static struct security_operations yama_ops = {
    .name =         "yama",
    .ptrace_access_check =  yama_ptrace_access_check,
```

```
    .ptrace_traceme =   yama_ptrace_traceme,
    .task_prctl =       yama_task_prctl,
    .task_free =        yama_task_free,
};
```

使用这种方式，在结构体定义变化导致元素的偏移位置改变时，仍然可以确保已知元素的正确性，对于未出现在初始化中的元素，初始值为 0。

传统安全钩子函数

在内核中定义了 security_operations 结构体，结构体由一系列的函数指针组成，每个函数指针都代表一个安全控制接口，来确保在内核中必要的阶段进行相应的安全检查。

```
struct security_operations {
    char name[SECURITY_NAME_MAX + 1];
    int (*ptrace_access_check) (struct task_struct *child, unsigned int mode);
    int (*ptrace_traceme) (struct task_struct *parent);
    int (*capable) (const struct cred *cred, struct user_namespace *ns,
            int cap, int audit);
    int (*quotactl) (int cmds, int type, int id, struct super_block *sb);
    int (*quota_on) (struct dentry *dentry);
    int (*syslog) (int type);
    int (*settime) (const struct timespec *ts, const struct timezone *tz);
    int (*vm_enough_memory) (struct mm_struct *mm, long pages);
    ...
}
```

内核中有一个静态的 security_operations 结构体指针 security_ops，用来指向实际启用的安全模块的安全钩子，各个安全钩子模块通过定义自己的一组钩子函数（放在 security_operations 结构体中），然后通过 register_security 函数将自己的 security_operations 结构体地址赋值给 security_ops 变量。

```
static struct security_operations *security_ops;
int __init register_security(struct security_operations *ops)
{
    ...
    security_ops = ops;
    ...
    return 0;
}
```

在系统运行中，需要进行安全检查时，内核就会调用 security_ops 中相应的钩子函数进行安全检查，以 security_ptrace_access_check 函数为例:

```
int security_ptrace_access_check(struct task_struct *child, unsigned int mode)
```

```
{
    ...
    return security_ops->ptrace_access_check(child, mode);
}
```

堆栈式安全架构

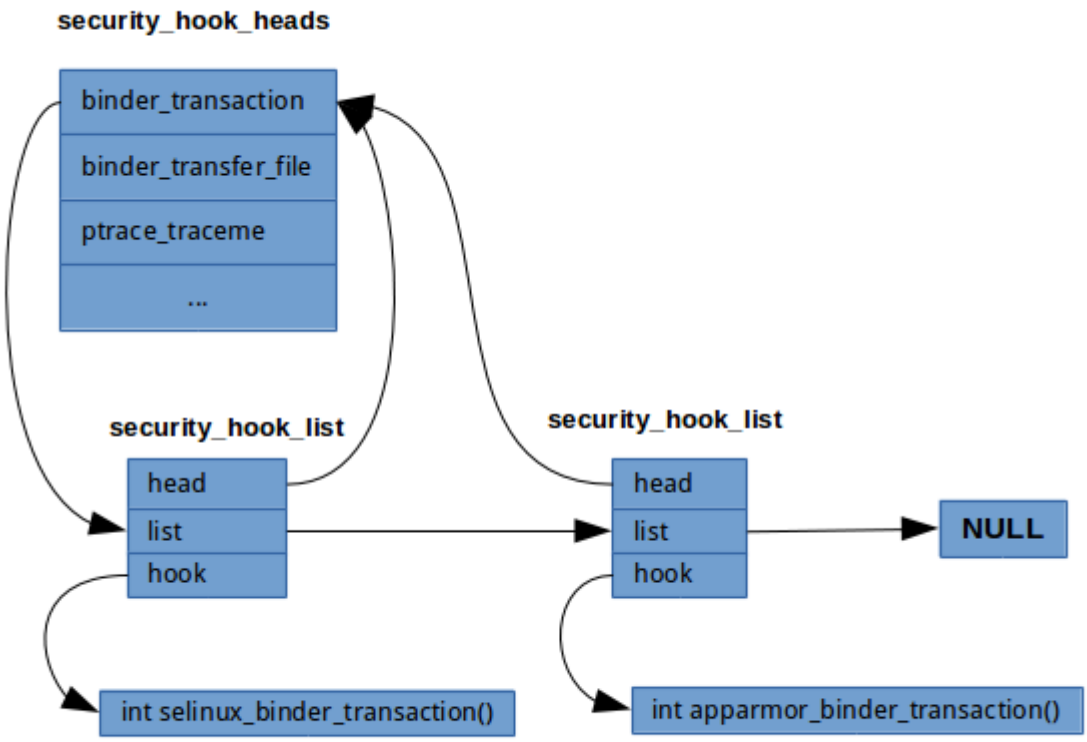
传统安全钩子模块存在以下缺点：只能默认开启一个安全钩子模块，如果要添加并启用额外的安全钩子模块，原来的安全模块就会被替代，内核代码改动大。

新的架构为每个安全控制接口都提供了钩子函数的链表，重要的钩子函数可以先加入链表，默认的安全模块的钩子函数会随后添加到链表中，钩子函数链表中的函数以加入的顺序进行调用，其中任何一个钩子函数返回检查失败时安全控制接口就会停止并返回，空的钩子函数链表意味着直接返回成功。这些安全控制接口的钩子函数链表放置在 security_hook_heads 结构体变量 security_hook_heads 中。

```
struct security_hook_heads {
    struct list_head binder_set_context_mgr;
    struct list_head binder_transaction;
    struct list_head binder_transfer_binder;
    struct list_head binder_transfer_file;
    struct list_head ptrace_access_check;
    struct list_head ptrace_traceme;
    ...
}
struct security_hook_heads security_hook_heads;
```

新的架构下，每个安全模块都需要提供一个钩子函数（security_hook_list 结构）组成的数组，数组中的钩子函数在注册安全模块的时候会添加到每个安全控制接口的钩子函数链表中，每个钩子函数都已 security_hook_list 结构存在。

```
struct security_hook_list {
    struct list_head        list;                   //用于连接到钩子函数链表
    struct list_head        *head;                  //钩子函数链表的头
    union security_list_options hook;               //钩子函数指针
};
```



安全钩子模块添加

每个安全钩子模块中都有一个钩子函数组成的数组，钩子函数的初始化通过 LSM_HOOK_INIT 宏来实现。LSM_HOOK_INIT 宏就是将钩子函数转换成相应的 security_hook_list 结构，并初始化结构体中的链表头 head。

```
static struct security_hook_list selinux_hooks[] = {
    LSM_HOOK_INIT(binder_set_context_mgr, selinux_binder_set_context_mgr),
    LSM_HOOK_INIT(binder_transaction, selinux_binder_transaction),
    LSM_HOOK_INIT(binder_transfer_binder, selinux_binder_transfer_binder),
    LSM_HOOK_INIT(binder_transfer_file, selinux_binder_transfer_file),

    LSM_HOOK_INIT(ptrace_access_check, selinux_ptrace_access_check),
    LSM_HOOK_INIT(ptrace_traceme, selinux_ptrace_traceme)
    ...
};

#define LSM_HOOK_INIT(HEAD, HOOK) \
    { .head = &security_hook_heads.HEAD, .hook = { .HEAD = HOOK } }
```

安全钩子模块的添加通过 security_add_hooks 函数来完成，实际就是对安全模块的钩子函数数组中的钩子函数进行遍历，添加到相应的函数链表中。

```
static inline void security_add_hooks(struct security_hook_list *hooks, int count)
{
    int i;
    for (i = 0; i < count; i++)
        list_add_tail_rcu(&hooks[i].list, hooks[i].head);
}
```

安全钩子函数的调用

钩子函数链表中的函数以加入的顺序进行调用，其中任何一个钩子函数返回检查失败时安全控制接口就会停止并返回，空的钩子函数链表意味着直接返回成功。

```
int security_ptrace_traceme(struct task_struct *parent)
{
    return call_int_hook(ptrace_traceme, 0, parent);
}

#define call_int_hook(FUNC, IRC, ...) ({            \
    int RC = IRC;                       \
    do {                            \
        struct security_hook_list *P;           \
        list_for_each_entry(P, &security_hook_heads.FUNC, list) { \
            RC = P->hook.FUNC(__VA_ARGS__);     \
            if (RC != 0)                \
                break;              \
        }                       \
    } while (0);                        \
    RC;                         \
})
```