操作系统安全
Operating System Security

[第7次课] Rootkit攻击原理及检测技术

授课教师：游瑞邦

授课时间：2024年4月12日

内容概要

01 Rootkit攻击原理

02 Rootkit检测技术

Linux系统启动过程及安全风险

- 上电
- 固定地址加载**BIOS**
- **BIOS**启动，关键硬件初始化，驱动初始化
- **Bootload**引导系统
- 内核初始化
- **Init**进程
- **login**
- 用户认证，权限解析，……

Rootkit概述

- 最早出现在Unix系统上
- 系统入侵者为了获取系统管理员级的root权限，或者为了清除被系统记录的入侵痕迹，会重新汇编一些软件工具
  - 如ps、netstat、w、passwd等系统管理工具

```
[root@izuf6gm25gfdxygdncwbk7z ~]# who
root     pts/0        2017-10-31 09:13 (121.225.111.194)
[root@izuf6gm25gfdxygdncwbk7z ~]# w
 09:31:14 up 37 days, 10:13,  1 user,  load average: 0.00, 0.01, 0.05
USER     TTY      FROM             LOGIN@   IDLE   JCPU   PCPU WHAT
root     pts/0    121.225.111.194  09:13    2.00s  0.03s  0.00s w
```

Rootkit概述

- 攻击者向计算机系统中植入，能够**隐藏自身踪迹**并**保留超级用户权限**的恶意程序，**避免被监控程序检测**
- 在目标系统上隐藏自身及指定的文件、进程、模块、进程信息和网络链接等信息
- 一般结合木马、后门等恶意程序
- 特点：间谍，持久且毫无察觉地驻留在目标计算机系统中
- 目标：隐藏、操作、收集数据
- 定义：是一种用来隐藏自己的踪迹和保留root访问权限的工具

Rootkit分类

- 应用级rootkit
  - 通过替换login、ps、ls、lsof、netstat等系统工具，或者修改一些系统配置文件、脚本来实现隐藏及后门(Ring 3)
  - hosts.equiv,.rhosts等系统配置文件
  - rexec, rcp, rlogin

Rootkit分类

- 内核级rootkit
  - Hook技术
    - 系统调用hook
    - 函数api hook
  - 直接内核对象操作—DKOM
    - 直接对内存中的内核状态进行修改，如结构体或者链表

Rootkit分类

- 硬件级rootkit
  - BIOS rootkit可以在系统加载前获得控制权，通过向磁盘写入文件再由引导程序加载该文件重新获得控制权
  - 替换关键文件

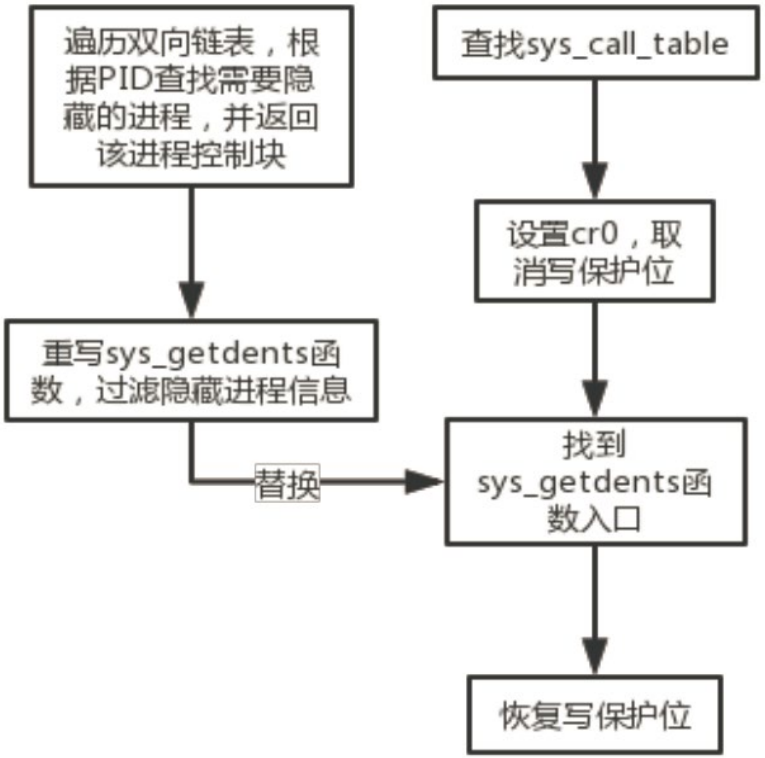
Rootkit分类

- Hypervisor rootkit
  - Intel VT,AMD-V硬件虚拟化 Ring -1
  - 早于虚拟机前加载
  - 不需要对目标的内核进行任何修改
  - 使用虚拟化技术，使整个操作系统运行在rootkit控制之中

- 隐藏文件
  - 原因
    - ls列出当前目录下的文件信息
    - ls 是通过系统调用sys_getdents64获得文件目录
  - 手段
    - 修改sys_getdents64系统调用或更底层的readdir实现隐藏文件及目录信息

- 隐藏进程
  - 原因
    - cat、ps、top和ls等命令读取/proc文件系统下的进程目录获得进程信息
  - 手段
    - 通过隐藏/proc文件系统下的进程目录来隐藏进程
    - 通过hook sys_getdents64和readdir等底层函数实现隐藏进程信息

劫持sys_getdents系统调用隐藏进程



- 隐藏连接
  - 原因
    - 查看网络连接状况主要通过netstat命令
    - 通过读取/proc文件系统下的net/tcp和net/udp文件获得当前链接信息
  - 手段
    - 通过hook sys_read调用实现隐藏连接
    - 通过修改tcp4_seq_show和udp4_seq_show达到隐藏连接目的

- 隐藏模块
  - 原因
    - lsmod通过sys_query_modules系统调用获得模块信息
  - 手段
    - 通过hook sys_query_module系统调用隐藏模块
    - 直接通过将模块从内核链表中摘除从而达到隐藏效果

- 网络嗅探
  - tcpdump
  - 通过libpcap库直接访问链路层，截获数据包
  - 通过linux的netfilter框架在IP层的hook点上截获数据包
  - hook sys_ioctl隐藏网卡的混杂模式

- 密码记录
  - 通过hook sys_read系统调用实现，
  - 通过判断当前运行的进程名或者当前终端是否关闭回显，可以获取用户的输入密码

- 日志擦除
  - 原因
    - 传统的unix日志主要在/var/log/messages，/var/log/lastlog，/var/run/utmp，/var /log/wtmp等文件记录
  - 手段
    - 通过编写相应的工具对日志文件进行修改
    - 将HISTFILE等环境变设为/dev/null隐藏用户的一些操作信息

- 内核后门
  - 本地提权后门
    - 通过对内核模块发送定制命令实现
  - 网络监听后门
    - 在IP层对进入主机的数据包进行监听，发现匹配的指定数据包后立刻启动回连进程

Rootkit主要技术

- LKM注射
  - 一种隐藏内核模块的方法，通过修改内核镜像或内核模块的ELF文件感染系统的内核模块
  - 在不影响原有功能的情况下将rootkit模块链接到系统内核可加载模块中，在模块运行时获得控制权

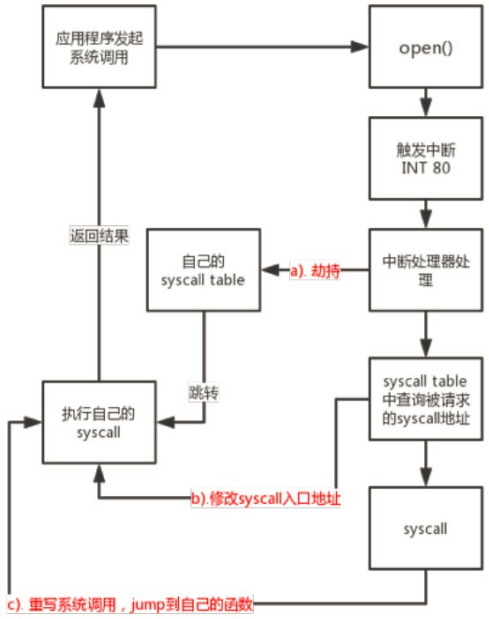
- 模块摘除
  - 将模块从模块链表中摘除从而隐藏模块
  - 最新加载的模块总是在模块链表的表头，因此可以在加载完rootkit模块后再加载一个清除模块将rootkit模块信息从链表中删除
  - 在新版本内核中可以通过判断模块信息后直接调用list_del内核函数

Rootkit主要技术

- 拦截中断
  - 通过sidt指令获得中断调用表的地址
  - 获取中断处理程序的入口地址
  - 修改对应的中断处理程序，如int 0x80等

- 劫持系统调用
  - 对系统调用表进行修改，直接替换原系统调用表
    - 修改系统调用表的入口地址
    - 对0x80中断处理程序进行分析从而获取系统调用表的地址

劫持系统调用


应用程序发起
系统调用
open()
触发中断
INT 80
返回结果
自己的
syscall table
a). 劫持
中断处理器处
理
跳转
执行自己的
syscall
syscall table
中查询被请求
的syscall地址
b).修改syscall入口地址
syscall
c). 重写系统调用，jump到自己的函数

Rootkit主要技术

- inline hook
  - 对内存中的内核函数直接修改
  - 采用跳转的办法，转入rootkit函数
  - 替换operations函数指针

Rootkit主要技术

- 运行时补丁
  - 文件系统、字符设备驱动程序和块设备驱动程序在加载时都会向系统注册 file_operations数据结构实现指定的read、 write等操作
  - 通过修改文件系统的file_operations结构，实现新的read、write等操作

- 端口反弹
  - 为了突破防火墙的限制，在客户端上监听80端口，而在服务器端通过对客户端的80端口进行回连，伪装成一个访问web服务的正常进程从而突破防火墙的限制

Rootkit修改系统调用实例

- 在系统调用挂钩技术中，最简单的方案是修改sys_call_table， 其成员类型为函数指针的一维数组

asmlinkage **const sys_call_ptr_t** sys_call_table[__NR_syscall_max+1] = { /* * Smells like a compiler bug -- it doesn't work * when the & below is removed. */ [0 ... __NR_syscall_max] = &sys_ni_syscall, **#include** <asm/syscalls_64.h> };

Rootkit修改系统调用实例

- 步骤
  - 首先，定位出**sys_call_table**在内存中的地址
  - 其次，去掉**sys_call_table**所在内存的写保护
  - 最后，修改**sys_call_table**函数指针

Rootkit修改系统调用实例

- 获得 sys_call_table 的内存地址
  - 获得方式
    - 暴力猜测
    - 通过/boot/System.map中读取
    - 从使用了sys_call_table的某些未导出函数的机器码里面进行特征搜索

Rootkit修改系统调用实例

获取sys_call_table地址代码

```
unsigned long ** get_sys_call_table(void)
{
    unsigned long **entry = (unsigned long **)PAGE_OFFSET;

    for (;(unsigned long)entry < ULONG_MAX; entry += 1) {
      if (entry[__NR_close] == (unsigned long *)sys_close)
        {
            return entry;
        }
    }

    return NULL;
}
```

* PAGE_OFFSET是内核内存空间的起始地址

* sys_close是导出函数

Rootkit修改系统调用实例

- 关闭sys_call_table写保护
  - **写保护指的是写入只读内存时出错**
  - **CR0寄存器控制写保护开启或者关闭，**
    - 开关在寄存器的第 16位

```
static inline unsigned long read_cr0(void);
static inline void write_cr0(unsigned long x);

 void disable_write_protection(void)
 {
    unsigned long cr0 = read_cr0();
    clear_bit(16, &cr0);
    write_cr0(cr0);
  }
```

Rootkit修改系统调用实例

- 修改sys_call_table
  - 直接修改sys_call_table函数指针数组值

```
disable_write_protection();
real_open = (void *)sys_call_table[__NR_open];
sys_call_table[__NR_open] = (unsigned long*)fake_open;
real_unlink = (void *)sys_call_table[__NR_unlink];
sys_call_table[__NR_unlink] = (unsigned long*)fake_unlink;
real_unlinkat = (void *)sys_call_table[__NR_unlinkat];
sys_call_table[__NR_unlinkat] = (unsigned long*)fake_unlinkat;
enable_write_protection();
```

内容概要

01 Rootkit攻击原理

02 Rootkit检测技术

Rootkit静态检测方法

- 可信任Shell
  - 使用静态编译的二进制文件： ps、 netstat、 w、 passwd、 ls、 lsof、 stat、 strace、 last、 ……
- 检测工具和脚本
  - rkhunter, chkrootkit, OSSEC
- LiveCD
  - DEFT、 Second Look、 Helix

Rootkit静态检测方法

- 动态分析和调试
  - 使用gdb根据System.map和vmlinuz image分析/proc/kcore
- 直接调试裸设备
  - debugFS

Rootkit静态检测方法对比

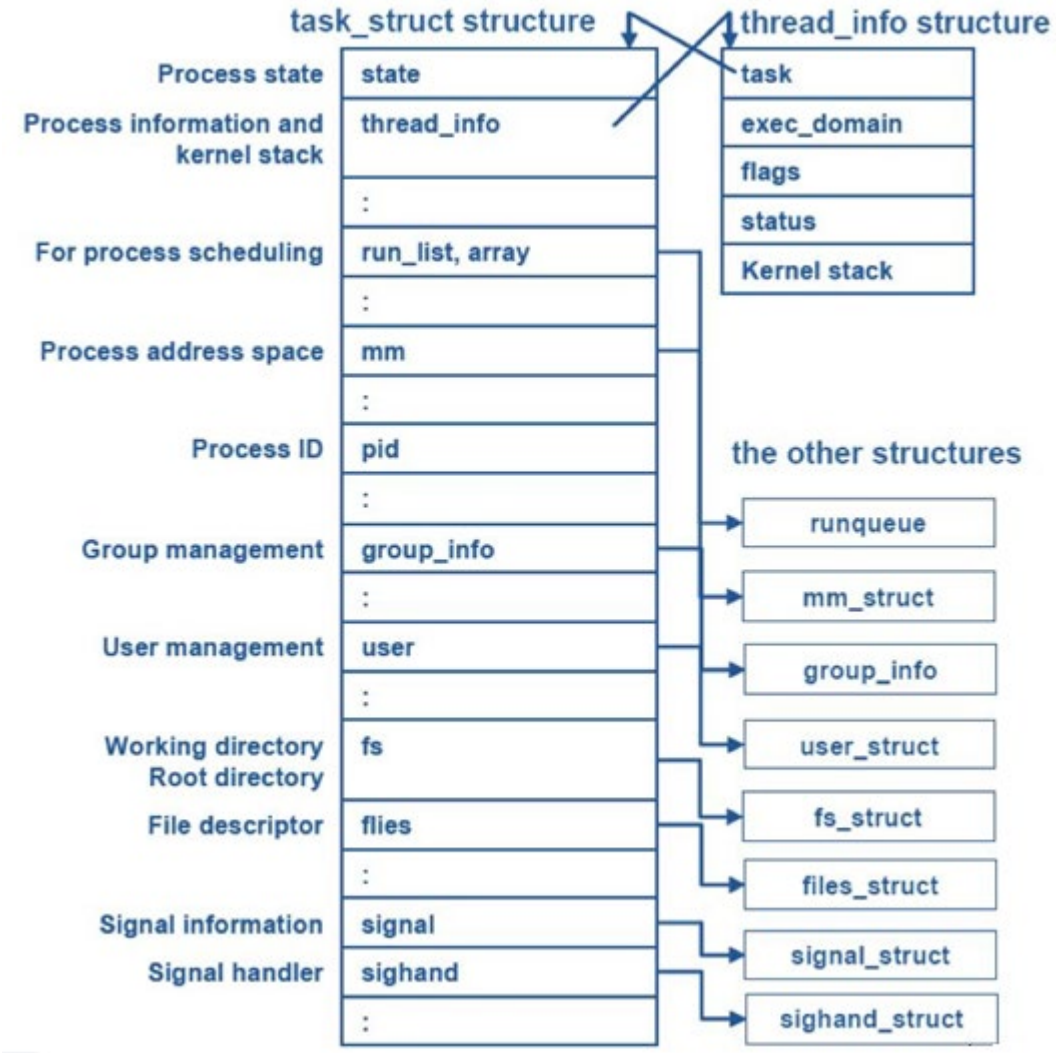
检测方式	局限/缺陷
使用静态编译的二进制文件	工作在用户空间，对Ring0层的Rootkit无效。
工具rkhunter,chkrootkit	扫描已知Rootkit特征，比对文件，检查/proc/modules，效果极为有限。
LiveCD:DEFT	Rootkit活动进程和网络连接等无法看到，只能静态分析。
GDB动态分析调试	调试分析/proc/kcore，门槛略高，较复杂。不适合应急响应。
DebugFS裸设备直接读写	不依赖内核模块，繁琐复杂，仅适合实验室分析。

- 基于内存检测技术
  - Rootkit难以被检测，主要是因为其高度的隐匿特性
    - 一般表现在进程、端口、内核模块和文件等方面的隐藏。
  - 一般情况下Rootkit在内存中会留下“痕迹”
    - 通过dump物理内存，并与内核调试符号信息和内核数据结构来对比解析内存文件
    - 对系统当前的活动状态的真实“描绘”和直接在系统执行命令输出的“伪造”结果做对比，找出信息的差异

基于内存分析检测进程

- 进程隐藏rootkit检测过程
  - 在Linux系统中查看进程通常通过ps –aux命令完成，其本质是通过读取/proc/pid/来获取进程信息
  - 每个进程的相关信息都可以通过其对应task_struct内存地址获取
    - 在内核的task_struct进程结构体中包含进程pid、 创建时间、映像路径等信息
  - 每个task_struct通过next_task和prev_task串起成为一个双向链表，可通过for_each_task宏来遍历进程
  - 通过对PID为0的init_task symobol（祖先进程）的内存地址，进行遍历进程链表输出系统进程信息

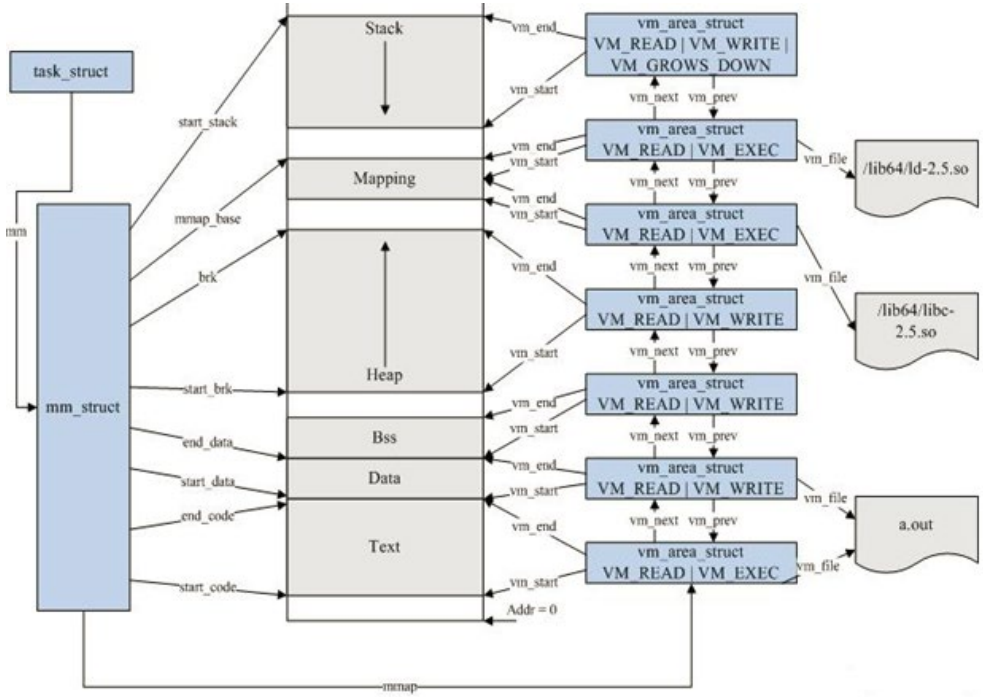
基于tast_struct内核数据结构检测



基于内存分析Process Memory Maps（进程映射）

- 进程映射分析
  - 在task_struct中，mm_struct描述了一个进程的整个虚拟地址空间，进程映射主要存储在vm_area_struct的数据结构中
  - 对比地址空间起止信息、访问权限和映射文件等信息

进程映射分析结构

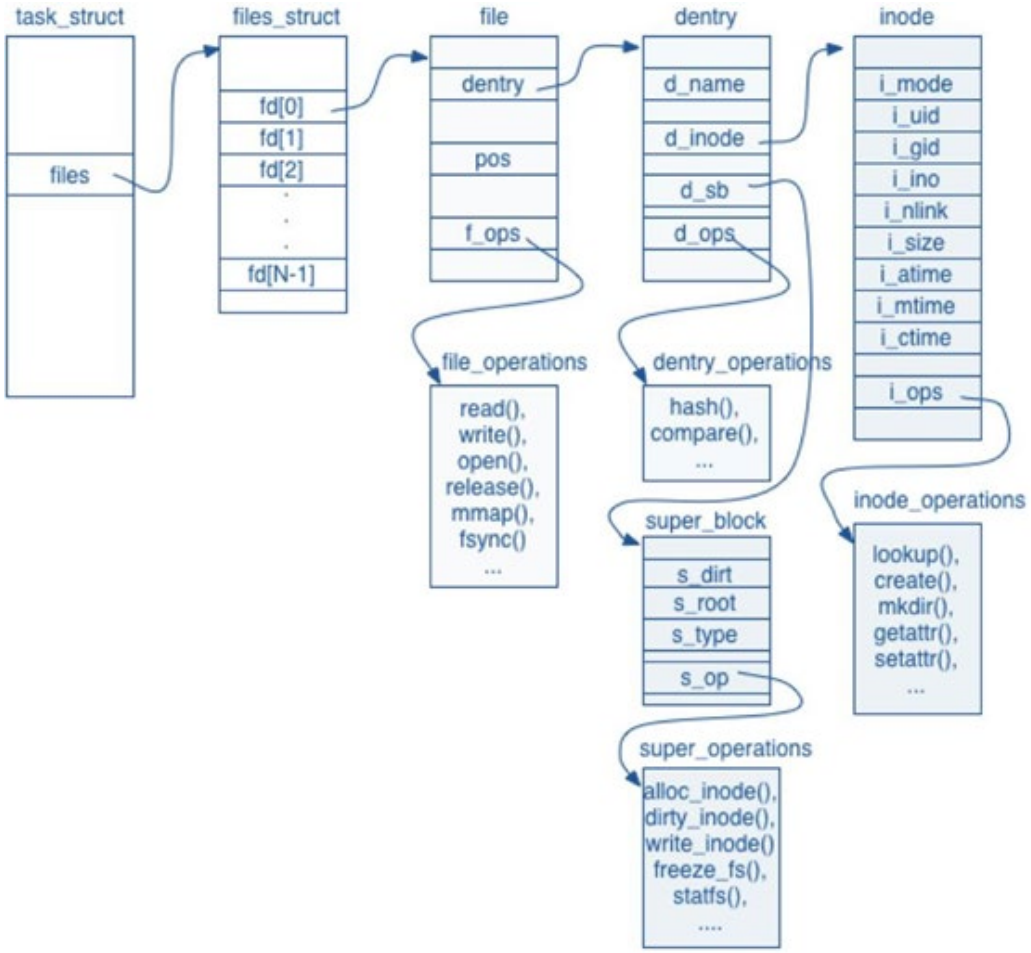

task_struct
mm
mm_struct
start_stack
mmap_base
brk
start_brk
end_data
start_data
end_code
start_code
Stack
Mapping
Heap
Bss
Data
Text
Addr = 0
mmap
vm_end
vm_start
vm_area_struct
VM_READ | VM_WRITE | VM_GROWS_DOWN
vm_next vm_prev
vm_area_struct
VM_READ | VM_EXEC
vm_file
/lib64/ld-2.5.so
vm_area_struct
VM_READ | VM_EXEC
vm_file
/lib64/libc-2.5.so
vm_area_struct
VM_READ | VM_WRITE
vm_area_struct
VM_READ | VM_WRITE
vm_area_struct
VM_READ | VM_WRITE
vm_file
a.out
vm_area_struct
VM_READ | VM_EXEC
vm_file

基于内存分析检测网络连接和打开的文件(lsof)

- List open files分析网络连接状况
  - Linux中的lsof（List Open Files）实质是读取/proc/pid/文件夹中的信息。
  - 打开文件信息根源在task_struct进程数据结构中记录
  - socket特殊的文件

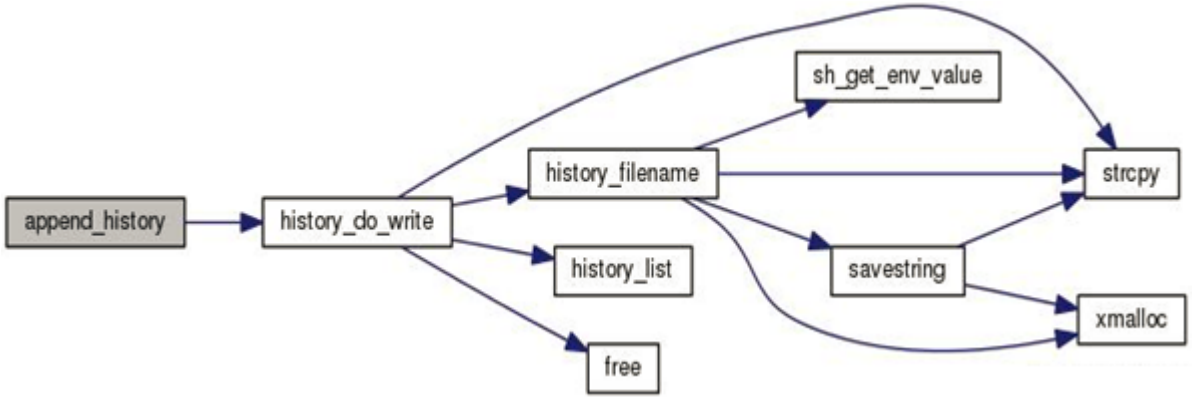
基于内存分析检测网络连接和打开的文件(lsof)

List Open Files分析结构图

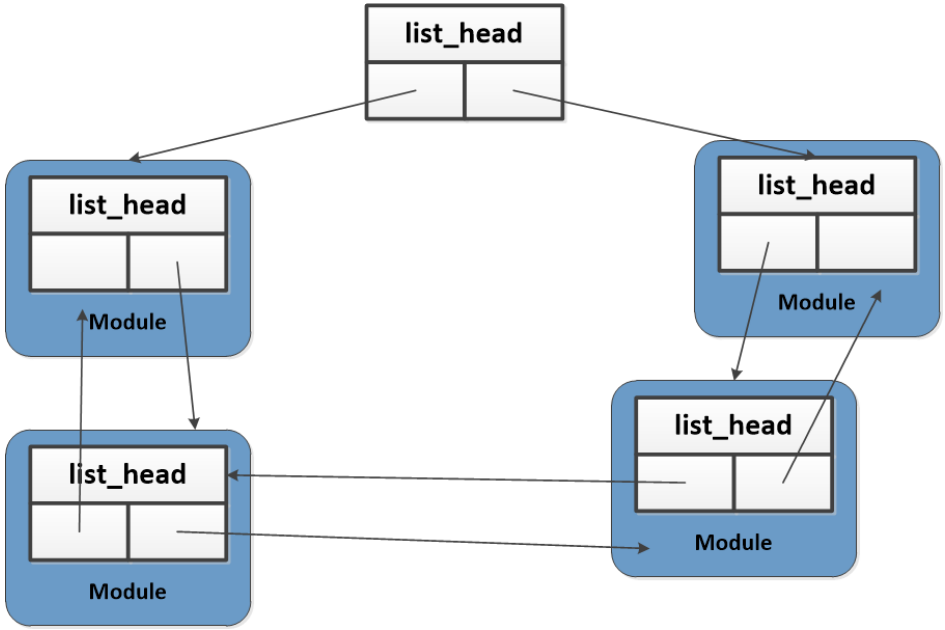


基于内存分析检测bash_history

- Bash_history分析原理
  - history –c命令来清空.bash_history文件的命令历史
  - 配置HISTSIZE = 0 或将HISTFILE = /dev/null
- **bash进程的history也记录在相应的MMAP中（其对应的宏定义为 HISTORY_USE_MMAP），通过history_list()函数相应的mmap数据也可以还原其历史记录**



- 内核模块检测分析原理
  - 通过遍历module list上所有的struct module模拟lsmod命令来检查Rootkit模块
  - Rootkit很难在/sys/module/目录中隐藏，可通过遍历sysfs文件系统来检查隐藏的内核模块



基于内存分析检测process credentials

- process credentials分析检测原理
  - Rootkit可以将用户态的进程通过设置其effective user ID和effective group ID为0（root）进行特权提升。
  - 新kernel把版本引入了 ‘cred’ structure。Rootkit与通过设置同某个root权限进程一样的 ‘cred’ structure 来应对这种改进。由于cred的唯一性，通过检查所有进程的'cred' structure 发现活动的Rootkit

基于内存分析检测总结

- 基于内存分析检测Rootkit的步骤
  - 1 定位内核数据结构
  - 2 通过内核数据结构复原信息
  - 3 管理工具输出信息和复原信息进行比对
  - 4 结果分析

其他检测技术

- 行为检测
  - CPU利用率差异
  - 异常网络流量
  - API调用频率差异
- 内存镜像转储，差异对比
- 防火墙
- 入侵检测系统

- 数字签名认证
- 可信度量检查
- 完整性检查


Q&A