基于 Linux Shell 的安全审计机制

阮 越,秦 锋,周建钦
(安徽工业大学 计算机学院,安徽 马鞍山 243002)

摘 要:用户登录后在 Linux Shell 中留下的历史记录是审计信息的重要来源,但未能包含判断入侵与否的足够信息,且很容易被篡改。文中基于 shell 机制,利用可装入内核模块、/proc 虚拟文件系统和系统调用劫持技术,实现了一个较全面的入侵检测的审计机制,同时给出了一个用其进行安全监测的简单实例以及该方法的优点。

关键词:安全审计;可装入内核模块;/proc 虚拟文件系统;系统调用劫持
中图分类号:TP316;TP393.08 **文献标识码**:A **文章编号**:1673-629X(2007)06-0155-04

Security Auditing Mechanism Based on Linux Shell

RUAN Yue, QIN Feng, ZHOU Jian-qin
(School of Computer Science, Anhui University of Technology, Ma'anshan 243002, China)

Abstract: Command history records which generated by Linux shell after user login is one of the important sources of system auditing information. But command history does not include sufficient information for intrusion detection and the history records can be easily modified. Adopted loadable kernel module technique, /proc virtual file system and system call interception, a full-function security auditing mechanism based on Linux shell is implemented in this paper, and then a simple example is given for security monitoring with the new mechanism. At last a discussion of its advantage is given.

Key words: security audit; loadable kernel module; /proc virtual file system; system call interception

0 引 言

随着 Internet 的发展,网络安全已经成为人们越来越关注的目标。如何及时发现 hacker 的入侵就成为安全防范的第一步,从广义的角度讲,所有的可执行的系统程序和用户程序都能看作 shell 的一部分,那么 shell 也就记录了登录用户在系统上的一切活动,这部分数据就成为发现用户异常行为的最直观和最有用的审计源。但 Linux Shell 保留的记录作为审计源还不够充分:记录的信息不够全面,大多数情况下仅有用户命令;记录的日志文件和历史文件很容易被有经验的 hacker 修改或者删除;有经验的 hacker 也很容易绕过 shell 的审计,比如在 C 程序中执行 system()或 exec()调用。

文中针对 Linux Shell 审计机制的缺陷[1,2],利用系统调用劫持和可装入内核模块技术(LKM,Loadable Kernel Module)实现了一种扩展 shell 审计机制的方法。它不仅可以更加全面地记录用户登录主机后的行为,也可以记录所有经 exec 类系统调用执行的命令。

1 LKM 相关技术

1.1 LKM

Linux 内核是作为单内核体系结构(Monolithic architecture)而实现的,为了获得微内核体系结构(Microkernel architecture)带来的可扩展性和可维护性,Linux 引入了 LKM 机制,藉此来保证内核的紧凑性和单一体系结构的优点——上下文切换速度快。

在 Linux 中,用户(通常需要 root 权限)通过 modutils 软件包中提供的工具,动态地将模块插入、移出内核来扩展和删除内核功能,这个过程不需要编译内核也不需要关机和重启。因为模块运行的环境是内核,因而它具有内核特权,模块编程也就是内核编程,Linux 中绝大多数的设备驱动程序都是通过 LKM 加载进内核的。

实现一个 LKM 最少需要两个基本函数:init_module()和 cleanup_module()。前者在模块装入内核

收稿日期:2006-08-28
基金项目:国家自然科学基金(60473142);安徽省教育厅自然科学研究项目(2006KJ063B);安徽省高等学校青年教师科研资助项目(2007jq1028)
作者简介:阮 越(1972-),男,湖北红安人,硕士,讲师,研究方向为系统安全和嵌入式系统;秦 锋,教授,研究方向为数据挖掘、网络安全。

时执行,后者在模块从内核卸出时执行。将一个模块装入内核包括 4 个任务:(a) 在用户空间编写/编译模块代码,解析未定义的符号,由模块装入器 insmod 完成链接装入过程;(b) 在内核地址空间分配内存;(c) 将模块代码复制到新分配的空间并向内核提供必要信息,以维护此模块;(d) 执行模块初始化例程 init_module()。

LKM 装入后即成为内核的一部分,它具有内核程序所有的权限(核心态运行),可以访问所有的内核资源。利用 LKM 挂接系统调用已经成为 hacker 入侵的一种常用手段[3,4],但这种技术也为系统管理员监测系统活动提供了一种方法。笔者利用 LKM 技术劫持系统调用以扩展 shell 审计机制。

1.2 系统调用劫持

Linux 通过 int 0x80 软中断实现系统调用。Linux 启动时,由 start_kernel()调用 trap_init()函数进行初始化,它的主要工作是通过 set_system_gate()设置 IDT 表中的第 128 号陷阱门(见 arch/i386/kernel/traps.c 文件)。系统调用完成用户态到内核态的模式转换,当用户程序通过 C 函数接口发出系统调用时,这些代码将首先被扩展为含有 int $0x80 汇编指令的汇编代码段,然后通过 int 0x80 软中断陷入到内核态的入口点 system_call(),该函数根据系统调用号找到对应系统调用例程执行。

与系统调用有关的数据结构和响应函数分别是系统调用号和系统调用表 sys_call_table。源文件 include/asm/unistd.h 中为每个系统调用定义了唯一的编号,这个编号就是所谓的系统调用号。系统调用表是一张由指向实现各种系统调用的内核响应函数的地址指针组成的表,系统调用号就是系统调用表中表项的相对偏移量(源文件 arch/i386/kernel/entry.S)。也就是说,系统调用处理程序 system_call()执行时,根据其传入 EAX 寄存器中的系统调用号,确定该响应函数在 sys_call_table 中的准确位置,从而确定该系统调用例程在内存的入口地址。

shell 作为 Linux 系统的外壳,是用户交互使用 Linux 的一个接口,负责解释和执行用户的命令和程序。shell 的基本动作是对用户命令进行语法分析,然后 fork()一个子进程并用 exec()来执行此命令。如果能劫持 exec 系统调用(exec 是一族系统调用,它们最终都通过 execve()来实现,所以只需劫持 execve()),则可以获得关于此命令的重要信息。

根据以上分析,只需重新编写 execve()函数,然后根据系统调用 execve()对应的系统调用号在 sys_call_table 中找到它的入口指针,用新函数指针替换掉老函数的入口地址,就可以实现系统调用劫持。

1.3 proc 文件系统

proc 最早出现在 Linux kernel 0.99.X 的正式发行版中,开始主要应用在网络相关方面,后来为了简化系统管理和调试,逐渐把它应用到其它方面。现在,proc 已经成为 Linux 内核中使用最广泛和最成功的特性之一(比如 ps 命令的实现就利用了/proc)[5]。

proc 借用了文件系统的概念,在内存中建立虚拟的文件节点(跟任何设备都没有对应关系),用户可以直接使用文件系统中的标准系统调用去访问 proc 下的信息,当用户发出访问/proc 下的“文件”请求时,再由系统动态生成。

proc 文件系统的超级块是在内核初始化的过程中由系统生成的,在将其挂载到安装点(mount point)“/proc”上后,就生成一个 proc_dir_entry 结构的树。/proc 下的一个文件节点对应一个 proc_dir_entry 结构,proc_dir_entry 结构中包含了建立对应文件 inode 和 dentry 结构所必需的信息。在具体的文件操作时(主要是 path_walk()函数),才生成具体文件的 dentry 和 inode 结构,逐渐建立起真正意义上的 proc 文件系统。

proc_dir_entry 是 proc 文件系统中最重要的数据结构,它构成了/proc 树形目录结构中的一个节点。通过对内核相关源代码的分析,文件读/写流程最后将落到内核函数 create_proc entry()中的 read_proc()和 write_proc()两个方法上。这里只需填写 proc_dir_entry 结构体,建立/proc 文件系统下的“文件节点”,然后实现 read_proc()和 write_proc()等函数,就可以将在内核态下获取的系统信息发送到用户态。

2 实 现

用运行在内核空间的 LKM 来收集 shell 命令的信息,运行在用户空间的安全监测程序则通过与 LKM 的通信来获取这些信息。本方法的主要思想是先创建一个/proc/cmdlog 的虚拟文件节点,然后在 LKM 中劫持 execve()系统调用并将信息写入/proc 中的虚拟内存页 * page 中。新的 execve()与/proc 文件系统共同维护 * page 页中的一个内核 FIFO 队列:新的 execve()每次取得一条 shell 命令的信息,作为一个节点加入 FIFO 队列;而当有用户进程读/proc/cmdlog 时,读操作处理函数 read_proc()就从 FIFO 队列中取出一个节点并将其复制到用户空间供用户进程使用;若 FIFO 队列中没有数据可用,则阻塞用户进程。

整个模块包含 3 个主要部分:创建/proc 节点、proc 读写和系统调用劫持。

2.1　模块定义

init_module()和 cleanup_module()是 LKM 必需的两个函数。在 init_module()函数中首先调用 create_proc_entry()创建/proc/cmdlog 文件节点(截取到的系统调用将写入这个文件节点),然后设置系统调用入口表中 execve()的入口地址,以截获对 execve()的调用。创建/proc 时需定义一个 file_operations 结构的 cmdlog_handler,用来指定对/proc/cmdlog 文件读写等操作时对应的处理函数(由于只是获取截获的系统调用的信息,所以只需定义读函数)。当有进程对文件实施操作时,自动调用相应的动作函数。cleanup_module()取消注册此设备,并恢复 execve()系统调用原来的入口地址。

```
struct proc_dir_entry * proc_cmdlog;
int init_module()
{
  proc_cmdlog = create_proc_entry("cmdlog", S_IFREG | S_IRUSR, &proc_root);/* 创建/proc 下的 cmdlog 节点 */
  proc_cmdlog->read_proc = read_cmdlog;/* 将我们编写的函数赋给 proc_dir_entry 中的读函数 */
  orig_execve = sys_call_table[SYS_execve]; /* SYS_execve 是 execve 调用号 */
  sys_call_table[SYS_execve] = new_execve; /* 用 new_execve 作为新的处理函数 */
  return(0);
}
int cleanup_module()
{
  sys_call_table[SYS_execve] = orig_execve; /* 恢复原 execve 系统调用 */
  remove_proc_entry("cmdlog", &proc_root); /* 删除/proc 下的 cmdlog 节点 */
  return(0);
}
```

2.2　读取/proc

/proc 文件的读取可以借助标准文件系统的系统调用实现,在内核中仅允许 root 用户在没有冲突时操作(在内核中用! suser()判断)。读操作处理函数牵涉到数据从内核空间向用户空间的复制,可以用 memcpy_tofs()或 copy_to_user()内核函数来实现。当用户进程读/proc/cmdlog 文件时,将自动调用 read_cmdlog()处理函数。

```
struct proc_dir_entry * proc_cmdlog;
int read_cmdlog(char * page, char * * start, off_t off, int count,
int * eof, void * data)
{
  int size=0;
  if(! suser())return(-1); /* 非 root 用户不可访问以保证日志完整性 */
  …
  /* has 指向 FIFO 命令日志链表首若无数据可读则阻塞此进程 */
  if (! has) {interruptible_sleep_on(&wp);signal_pending(current);}
  if (! access_ok(VERIFY_WRITE, buf, buflen)) return(-1);
/* 当前进程是否允许访问 buf */
  size+ = sprintf(page + size, "%d:%d:%d:%d:%s:", has->start_time, has->uid, has->euid, has->gid, has->p_comm");
  );/* 获取日志信息写入内存页 * page 中 */
  copy_to_user(buf, page, buflen); /* 将日志信息复制到用户空间内存缓冲区 buf */
  … /* 释放已读过的日志结点的内核空间 */
}
```

2.3　中断劫持

new_execve 是 execve()系统调用新的处理函数。其主要工作是从内核的任务控制结构中取出必要信息放在输出链表中,供读取/proc 文件的进程使用,再真正执行 shell 命令。

```
int new_execve(struct pt_regs r) /* 用来劫持原 execve()系统调用 */
{
  char * cp, * filename=NULL;
  char ch;
  int ret=0,i=0;j=0;
  lock_kernel();
  /* 为新的日志结点在内核分配空间,用 kmalloc() */
  struct lognode * logp=(struct lognode * )kmalloc((sizeof(struct lognode),GFP_HERNEL);
  logp->cmd = (char * ) kmalloc(MAXNAME, GFP_ KERNEL);
  /* 从用户空间取当前进程名,即 shell 命令字 */
  filename=(char * )getname((char * )r.ebx);logp->next = NULL;
  … /* 日志链表管理及读等待进程队列管理 */
  /* 复制 shell 命令字到日志结点 */
  memcpy(logp->cmd, filename, MAXNAME-1);i = strlen(logp->cmd);
  /* 从用户空间复制命令行参数到内核空间,用内核系统调用 get_user() */
  char * * cmdargv=(char * * )r.ecx;get_user( * cp, + + cmdargv);
  while ((cp! =NULL)&&(i<MAXNAME-2)) {
    logp->cmd[i+ +]='';
    while (1) { /* 每次循环从用户空间取一个参数 */
      get_user(ch, cp+j);
```

```
    if ((ch! =NULL)&&(i<MAXNAME-2)) {logp->
cmd[i++]=ch;j++;}
    else break;
  }
  j=0;get_user(cp,++cmdargv); /* 取下一参数 */
 }
 /* 复制其他信息 */
 memset(logp->p_comm,0,16);strcpy(logp->p_comm,
current->p_opptr->comm,16);
 logp->uid=current->uid;logp->euid=current->euid;
 logp->current->gid=current->gid;…;
 /* current 是一个 task_struct 结构的全局变量，含有当前正
在处理机上执行进程的信息 */
 logp->start_time=CURRENT_TIME; /* 获得时间,参见
linux/timex.h */
 ret=do_execve(filename,(char**)regs.ecx,(char**)regs.
edx,&regs); /*执行 shell 命令 */
 putname(filename);unlock_kernel();return(ret);
}
```

3 安全监测

3.1 审计能力比较

扩展 Shell 安全审计能力的 LKM 生成的审计记录格式为 todaytime:uid:euid:gid:parent:cmd。各项分别为命令开始时间、用户号、有效用户号、组号、父进程名和 shell 命令。这些信息对于安全监测和入侵检测是必要的和方便的。无论是缓冲区溢出入侵攻击,还是黑客留下的后门程序,其最终目的是都是了获得 root shell。我们定义了简单的审计规则,并通过一个简单实例来剖析这种方法的优缺点。

3.2 简单监测规则的描述

审计规则的简单描述:

(1)监测所有在早上 6:00 到晚上 6:00 执行的命令,规则为:

((6 * 60 * 60 < todaytime) && (todaytime < 18 * 60 * 60))

(2)监测所有以 root 权限重新装载 shell 的行为,规则为:

(euid==0) && ((cmd==/bin/bash) || (cmd==/bin/csh) || (cmd==/bin/sh))

(3)监测所有 suid root 程序的执行,监测后门程序和缓冲区溢出攻击,规则为:

(uid! =0) && (euid==0)

3.3 实 例

黑客入侵后留下的后门程序(比如 suid 程序)或者缓冲区溢出攻击程序,这些程序的最后往往要执行 exec 类函数或简单的 system 调用重新装载 shell 以获得 root 权限。表 1 给出了一后门程序执行时两种日志系统能力的比较。后门程序很简单,只是一个 suid 程序,利用 system("/bin/sh")重新装载 shell。

表 1 两种日志系统的能力比较

Executing backdoor programe	Extended log records	Shell records
$ ./kop kop 为后门程序,它可以隐藏在系统的任何一个角落,同时具有 suid 权限 # (becomes root)	…(略) 40214:503:0:100:bash:./kop<--检测规则:(uid! =0) && (euid==0) 40214:503:0:100:kop:/bin/sh 40214:0:0:100:/bin/sh:/bin/sh<--检测规则:cmd=/bin/sh …(略)	./kop 仅仅有一个程序名,不包含任何其他信息

System 函数是通过 fork, exec 和 waitpid 来实现的。首先 fork 一子进程,由该子进程调用 shell 命令解释程序执行诸如 execl(/bin/sh,"sh","-C","/bin/sh",(char *)0)的类似代码。

4 结 论

利用 LKM 扩展 Shell 安全审计功能具有如下优点:

a) 将 shell 的审计能力扩展到所有经 exec 调用执行的命令,可以获得充分的审计信息;

b) /proc 虚拟文件节点只允许 root 操作而保证了审计日志的完整性;

c) 这种审计功能对普通用户和应用是透明的;

d) 对入侵者保持了隐蔽性;

e) 审计记录的获取只需在装入此 LKM 以后,按正常的文件操作方式操作"proc/cmdlog"即可。

参考文献:

[1] IBM, NOVELL, SUSE. Linux Audit - Subsystem Design Documentation for Kernel 2.6[EB/OL]. 2004. http://ftp.isr.ist.utl.pt/pub/MIRRORS/ftp.suse.com/projects/security/laus/sles9/laus/doc/LAuS-Design.pdf.

[2] Durst R, Champion T, Witten B, et al. Testing and evaluating computer intrusion detection systems[J]. Communications of the ACM, 1999,42(7):53-61.

[3] Halflife. Linux TTY hijacking[J]. Phrack Magazine, 1997,7(50):5-5.

[4] Salzman P J, Burian M, Pomerantz O. The Linux Kernel Module Programming Guide[EB/OL]. 2005. http://www.tldp.org/guides.html.

[5] Bowden T. The proc-howto[EB/OL]. 2004-01. http://www.linuxforum.com/linux-filesystem/proc.html.