第 34 卷 第 7 期 计 算 机 工 程 2008 年 4 月
Vol.34 No.7 Computer Engineering April 2008
·安全技术· 文章编号：1000—3428(2008)07—0160—03 文献标识码：A 中图分类号：N945

LSM 框架下可执行程序的强制访问控制机制

刘威鹏[1]，胡 俊[1]，吕辉军[2]，刘 毅[3]
(1. 中国科学院研究生院信息安全国家重点实验室，北京 100039；2. 国防科学技术大学计算机学院，长沙 410000；
3. 解放军信息工程大学电子技术学院，郑州 450052)

摘 要：分析 LSM 框架的基本设计思想和 Linux 系统上原有的可执行程序访问控制机制存在的问题，在此基础上讨论在 LSM 框架下可执行程序强制访问控制机制的设计。作为验证，基于 Linux2.6.11 内核实现了一个可执行程序强制访问控制系统原型，对如何在操作系统中实现可执行程序的强制访问控制具有指导意义。

关键词：LSM 框架；强制访问控制；可执行程序

Mandatory Access Control Mechanism of Executable Program Under LSM

LIU Wei-peng[1], HU Jun[1], LV Hui-jun[2], LIU Yi[3]
(1. State Key Laboratory of Information Security, Graduate University of Chinese Academy of Sciences, Beijing 100039;
2. Computer Institute, University of National Defense Science and Technology, Changsha 410000;
3. Electronic Technology Institute, PLA Information Engineering University, Zhengzhou 450052)

【Abstract】This paper analyses the main design idea of Linux Security Module(LSM) and the problem of the intrinsic access control mechanism of Linux executable program, and discusses the design of Mandatory Access Control(MAC) mechanism of executable program based on LSM. As the demonstration, it implements a MAC system prototype based on Linux kernel 2.6.11. The illumination that how to implement MAC of executable program in operating system is given.

【Key words】Linux Security Module(LSM); Mandatory Access Control(MAC); executable program

访问控制作为一种最为基本和重要的安全机制，对于保护操作系统中资源免受非法访问起着关键的作用。访问控制可以分为自主访问控制(DAC)和强制访问控制(Mandatory Access Control, MAC)，与DAC相比，MAC中的安全策略由安全管理员根据安全威胁和安全假设预先定义，用户或代表用户的进程即使拥有客体也不能修改其安全属性，从而能够有效防止特洛伊木马的攻击，这对于保证整个系统的安全具有重要意义，因此，MAC在高等级安全操作系统的设计中被强制要求。然而，当今主流的操作系统并未提供MAC机制，难以为系统提供充足的安全保证。为此，学术界对如何增强操作系统的MAC机制进行了大量研究。Linux的创始人Linus认为Linux内核需要一个通用的访问控制框架，响应他的号召，DARPA等开展了LSM(Linux Security Module)[1]的研发。LSM是一种灵活的通用访问控制框架，用户可以根据安全需求来选择安全模块，加载到Linux内核中，从而提高其安全访问控制机制的易用性。

1 LSM 框架研究

LSM 的设计思想是在内核对象数据结构中放置透明安全域(opaque security field)作为其安全属性，并在内核源代码中放置钩子(hook)函数，由 hook 函数来完成对内核对象的访问的判断，基本原理如图 1 所示。在 LSM 框架下，用户进程执行系统调用时，将通过原有的内核接口逻辑依次执行功能性错误检查、传统的 DAC 检查，并在访问内核对象之前，通过 hook 函数调用具体的访问控制策略实现模块来决定该访问是否合法。

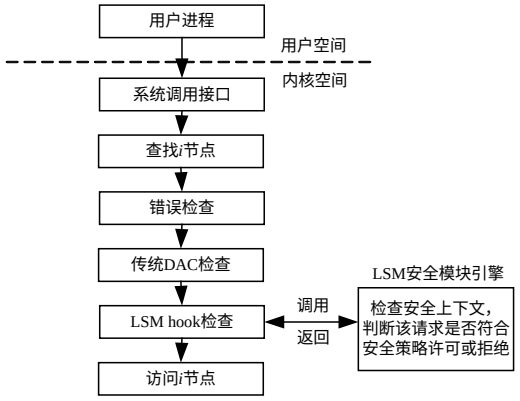

图 1 LSM 的基本原理

LSM实现了一个通用的访问控制框架[2]，为安全模块的开发提供了统一的标准接口，并对内核作了以下 2 处重要修改[3]：(1)为内核数据结构添加透明安全域；(2)在内核代码中的关键访问点插入安全钩子函数。

1.1 透明安全域

任何安全模型要控制主体对客体的访问，首先必须对主和客体进行安全标识。LSM 作为一个通用的访问控制框架，

基金项目：国家“973”计划基金资助项目“信息与网络安全体系结构研究”(G1999035801)
作者简介：刘威鹏(1980－)，男，博士研究生，主研方向：可信计算，安全操作系统；胡 俊、吕辉军、刘 毅，博士研究生
收稿日期：2007-05-09 **E-mail：**wpliu@is.ac.cn

不能为内核对象指定特定的安全标签，而需要指定一个能够适应安全模型变化的标签。因此，LSM 框架通过在内核对象的数据结构中添加透明安全域来实现框架本身的通用性。LSM 在内核中 9 个与访问控制相关的内核对象数据结构中添加了透明安全域，例如：inode(文件)、Linux_bprm(可执行程序)、super_block(文件系统)等。透明安全域本身是一个空指针类型(void*)，该域的具体结构由安全模块定义，通过该域能将安全信息与内核对象关联。

1.2 安全钩子函数

LSM 提供了 2 类 hook 函数：(1)用于分配、释放和管理内核对象的透明安全域；(2)用于控制对内核对象的访问。对这 2 类 hook 函数的调用均是通过全局表 security_ops(即指向 security_opertations 的指针)中的函数指针来实现的，在 security_operations 结构体中定义了 150 多个函数指针，这些指针分为 6 类，分别与进程、文件系统、可执行程序、进程间通信、网络以及系统操作有关，安全开发者通过对其中的 hook 函数进行具体实现来满足实际的安全需求。

2 可执行程序的强制访问控制的设计和实现

2.1 问题分析

在 Linux 系统中的/bin, /sbin 等目录中存在很多与系统管理相关的可执行程序，由于它们涉及到许多重要的操作，例如，通过执行 grub 能够更改系统的启动，通过 insmod, rmmod 能够添加和删除模块，因此系统提供了一套自主访问控制机制来对这些可执行程序进行保护。每个可执行程序根据其所属的“ owner-group-other ”3 个层次各自分为“ 读(r)-写(w)-执行(x)”3 个操作项，系统提供了 chmod 的命令，允许可执行程序的拥有者能够修改操作项，还提供了 chown 命令允许拥有者改变对象的拥有权。在上述目录中，绝大部分可执行程序在“ 其他 ”位上都设置为“ x ”标志，意味着这些程序对于系统中任何用户都是可执行的，而在实际的应用环境中，系统的普通用户只能够执行完成基本任务的一系列可执行程序，而不应有其他特权操作。为方便系统的管理，还设置了 root 用户，该用户具有特权，能够执行任何可执行程序并对其执行其他操作。这种对于可执行程序的保护机制的安全性建立在 root 用户不会被冒充并熟悉系统各项管理等假设的基础之上，因此，由 root 用户全权负责系统的管理在一定程度上能够方便系统的管理，并易于使用。而在实际中，一旦超级用户被攻破或超级用户进程被非法控制，会给系统带来很大的安全威胁和隐患。

2.2 安全策略和设计方案

对于高安全等级的操作系统，强制访问控制是必须的。为了防止用户对可执行程序的滥用，避免不必要的权限扩大给系统带来的安全威胁和安全隐患，需要对用户能够运行的可执行程序进行强制访问控制限制。该强制性是系统安全管理员通过授权管理工具，根据系统的实际安全需求和威胁进行配置的，即使是系统的超级用户或可执行程序的拥有者也无法改变。

在原型系统中，用户分 2 级：0 级为管理员用户，1 级为普通用户；可执行程序也分为 2 级：0 级为特权可执行程序，1 级为普通可执行程序。此外，每个用户具有一个访问控制号(ACL_NUM)，每个可执行程序具有一个访问控制字符串(ACL_STRING)。可执行程序强制访问控制策略为：0 级用户可以执行 0 级和 1 级的可执行程序，1 级用户不能执行 0 级可执行程序，只能执行在全局资源访问控制链表中级别为 1，并且该用户的 ACL_NUM 和链表中可执行程序 ACL_STRING 相匹配的可执行程序。

为便于可执行程序管理，在系统中维护了 3 个可执行程序控制白表：(1)系统最小授权可执行程序白表，定义了每个用户在系统中能够执行的最小的操作集合；(2)用户组最小授权可执行程序白表，在系统中，每个用户隶属于一个用户组，用户组中的每一个用户都继承用户组最小授权可执行程序白表；(3)用户配置可执行程序白表。在通过授权管理工具成功添加一个新用户后，该用户能够继承系统最小授权可执行程序白表和该用户所隶属的用户组授权最小可执行程序白表，而通过授权管理工具，能够对于用户配置可执行程序白表进行添加、删除以及修改操作，也能够对用户组授权最小可执行程序链表进行修改。以上 3 个白表最终合成用户可执行程序白表。在 Linux 安全模块初始化时，系统的链表初始化函数读取用户可执行程序白表文件，在系统内存中构造出全局资源访问控制链表，用于程序访问控制的最终判定使用。

2.3 关键数据结构和主要钩子函数

作为验证，在 Linux2.6.11 内核上实现了可执行程序的强制访问控制系统原型，以下是关键数据结构：

```
//系统核安全状态数据结构
struct Sos_core_sec_state_type {
      struct list_head * fs_list;
      void * user_sec_data; };
```

上述结构是系统安全中枢，记录了系统安全信息。fs_list 为指向文件系统安全状态数据结构队列的指针，在系统中维护一个文件系统安全数据状态结构链表，从而提供对多个不同类型文件系统的支持。user_sec_data 是指向用户安全状态数据结构的指针，这时的系统核安全状态数据结构和一个用户安全状态数据结构相对应。其中，list_head 结构体是 Linux 内核源代码中定义的双向链表，该结构体本身不存取任何信息，只是提供给其他结构作为双向链表的接口。

```
//文件系统安全状态数据结构
struct Sos_fs_sec_state_type {
  struct list_head * file_attr_listhead; };
```

上述结构是添加在内核超级块(super_block)数据结构上的透明安全域，记录了某个文件系统的安全信息，其中 file_attr_listhead 为指向资源(文件或目录)访问控制链表的指针，该队列形成整个文件系统的访问控制信息列表。

```
//索引节点安全状态数据结构
struct Sos_inode_sec_state_type {
  unsigned int inode_mac_level;
      char acl[16];
};
```

上述结构是添加在内核索引节点(inode)数据结构上的透明安全域，记录了某个索引节点的安全信息。其中，inode_mac_level 是某个索引节点的访问控制安全级别，即强制访问控制中客体的安全标记，用于判断用户能否访问可执行程序。acl[16]是一个字符串数组，其中存储了一个 128 b 的二进制数，用于维护索引节点的访问控制信息。

```
//系统用户安全数据结构
struct Sos_user_sec_data_type {
  int user_level;
      int user_acl_no;
};
```

上述结构主要记录用户安全信息。其中，user_level 为用户的安全级别，即主体的安全标记，用于判断用户能否访问

某个可执行程序。user_acl_no 是用户的 acl 号，结合 acl[16]用于判断用户是否可以执行某个可执行程序。所采用的方法如下：如果用户的 acl 号为 $n(0<n<128)$(在添加用户时确定)，并且索引节点的 acl[16]中从右数第 n 位为 1，那么允许用户运行该可执行程序，否则不允许执行。在对用户进行可执行程序授权时，用户可执行程序白表的程序的 acl[16]根据用户的 acl 号置位。

这 4 个数据结构存在一定的关联。系统核安全状态数据结构中包括指向文件系统安全数据结构队列的指针和用户安全数据结构的指针，而在文件系统安全数据结构中又维护着索引节点安全属性队列指针。

Linux 中的 struct linux_binprm 用来记录一些在装入新程序时所需的参数和环境变量。LSM 提供了用于控制可执行程序操作的 hook 函数：

bprm_check_security (struct linux_binprm * bprm)

该函数在目标程序装载之前进行执行权限检查，这个钩子在一次 exec()中可能被调用多次。原型系统中有 2 个关键的钩子函数：

(1)Sos_post_addmount()，作用是在系统挂载新文件系统时，读取该文件系统的访控文件，并根据访控文件在内存中产生系统安全核状态数据结构；

(2)Sos_bprm_check_security()，作用是在用户运行可执行程序前，对用户和可执行程序的安全标签进行比对，并在访问控制链表进行搜索，判断用户是否有权运行该可执行程序，其流程如图 2 所示。

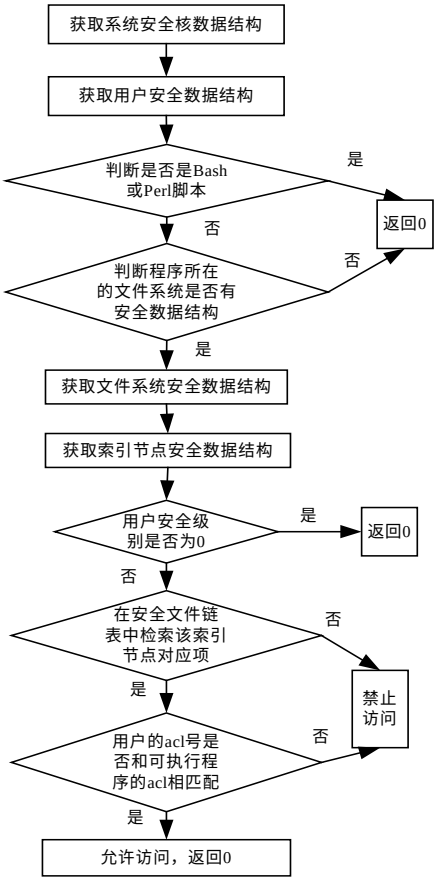

图 2　int Sos_bprm_check_security()流程

其关键代码如下：

```
int Sos_bprm_check_security (struct linux_binprm * bprm){
//根据 inode 在访问控制队列中进行查找
struct list_head * attr_head;
  attr_head = find_elem_with_tag(
  fs_sec_data -> file_attr_listhead, comp_inodeattr_with_ino, &(ino->i_ino));
//对应项不为空
if(attr_head !=NULL) {
//判断用户的 acl 号是否和可执行程序 acl 是否一致，若一致，//则不进行控制，否则拒绝执行
if(os210_acl_test(user_sec_data -> user_acl_no, inode_sec_data -> inode_acl)){
            return 0;}
      else {printk("exec %s denied!\n",bprm->filename);
            return -EPERM;}
//对应项为空，拒绝执行
else{printk(" exec %s denied!\n",bprm->filename);
return -EPERM;}
}
```

在 LSM 框架下编写了完 hook 函数后，需要将 security_operations 结构体内的各个成员(函数指针)指向自己编写的 hook 函数，即

```
struct security_operations Sos_security_ops = {
      .bprm_check_security =Sos_bprm_check_security;
   .sb_post_addmount =Sos_post_addmount;};
```

而后使用 register_security(&Sos_security_ops)在 LSM 框架中注册这些函数。

3　结束语

LSM 是内核级的灵活通用的访问控制框架，它在某种程度上减少了安全模块开发者对于内核代码的关注，使开发者能够关注于安全策略，同时也方便了用户对于安全策略的选择。在 Linux 系统中，超级用户能够执行所有的可执行程序，为了防止用户对于可执行程序的滥用，防止不必要的权限扩大对系统带来的安全隐患，需要对用户能够运行的可执行程序进行限制。本文以 LSM 框架为基础，讨论了可执行程序强制访问控制机制的设计，并在 Linux2.6.11 内核上实现了一个原型系统。作为一个通用访问控制框架，使 LSM 能够支持多策略和灵活安全策略是今后的研究方向。

参考文献

[1] Spencer R, Smalley S. The Flask Security Architecture: System Support for Diverse Security Policies[C]//Proceedings of the 8th USENIX Security Symposium. Washington, D. C., USA: [s. n.], 1999: 123-139.

[2] Loscocco P, Smalley S. Integrating Flexible Support for Security Policies into the Linux Operating System[C]//Proceedings of the FREENIX Track: 2001 USENIX Annual Technical Conference. Berkeley, CA, USA: USENIX Association, 2001: 29-42.

[3] Wright C, Cowan C, Smalley S, et al. Linux Security Modules: General Security Support for the Linux Kernel[C]//Proc. of the 11th USENIX Security Symposium. San Francisco, CA, USA: USENIX Association, 2002.