The 'Rule Set Based Access Control' (RSBAC) Framework for Linux

Amon Ott
Compuniverse
D-22949 Ammersbek / Germany
Email: ao@compuniverse.de

Simone Fischer-Hübner
Karlstad University
Department of Computer Science
SE-651 66 Karlstad / Sweden
Email: simone.fischer-huebner@kau.se

Abstract:
RSBAC ("Rule Set Based Access Control") is an open source security extension for Linux kernels based on the Generalized Framework for Access Control (GFAC). It is a kernel-based access control scheme, which can be configured with a set of security policies chosen from a provided set of options and which can be used to significantly enhance Linux system security.
In this paper, we present the RSBAC system architecture, the RSBAC security policy components and outline the RSBAC implementation. Besides, we briefly compare RSBAC with other Linux kernel-based access control projects.

Keywords: operating systems security, Linux security, access control models and policies

1. Introduction

1.1 The Problem of Linux Insecurity

Linux systems, as many others in the Unix family, have a well-known lack of access control. First of all, there is the small granularity of discretionary access rights, only dividing between read, write and execute rights for file owner, file group members and all others.
The fact that access control relies on a file owner's discretion already leads to various problems, like the level of trust that has to be put in a user, the vulnerability from malware working on behalf of a user, etc. Also, there is hardly any logging of user activities possible, making it even harder to detect malicious accesses.
The worst problem, however, is the system administrator account 'root'. Many system tasks are only allowed to be done by this user, even many network services have to be started or, worse, run as root. On the other hand, the root account has full access to every object in the system. It is easy to understand why so many Unix family systems have been compromised locally or by remote access.[1]

1.2 RSBAC outline

RSBAC ("Rule Set Based Access Control") is an open source security extension for current Linux kernels [RSBAC], [Ott 2001a], [Ott 2001b]. It is based on the Generalized Framework for Access Control (GFAC) by Abrams, LaPadula et al. ([Abrams et al. 1990], [LaPadula 1995]) and provides a flexible access control system with several modules implementing different security policies.

[1] Section 7.1 compares RSBAC to the Linux Privileges scheme which has been included into the official Linux kernel.

GFAC was introduced as a framework for expressing and integrating multiple policy components. Since in GFAC the access control enforcement facility and access control decision facility are implemented and thus separated in two different components, GFAC makes it feasible to configure and extend a system (more precisely: the access control decision facility) with a combination of security policies chosen from a provided set of options, with confidence that the resulting system´s security policies will be properly enforced. A draft top-level specification, which specifies how the GFAC approach can be implemented in Unix System V was published in [LaPadula 1995]. This draft specification was more elaborated, changed and extended in many respects and then adapted and used for the implementation of the Rule Set Based Access Control (RSBAC) in Linux.
The RSBAC system can bring a significantly higher level of security to the Linux kernel and operating environment.
In this paper, we present the RSBAC system kernel architecture, the RSBAC system components and outline the RSBAC implementation. Section 2 introduces the overall system architecture as derived from the GFAC approach. While Sections 3 and 4 explain the ACI data handling and the interface between enforcement and decision components, section 5 introduces the RSBAC logging system. The system performance is subject of section 6. Section 7 provides a brief comparison to other Linux kernel security extensions. Finally, sections 8 and 9 give an outlook to new development and some final remarks.

2. RSBAC Architecture

According to the GFAC approach, the system's security kernel consists of an access control enforcement facility (AEF), an access control decision facility (ADF) and a ACI module which administrates Access Control Information (ACI, e.g. security attributes). ADF implements the system's mandatory security policies and a metapolicy to decide whether processes' requests satisfy those security policies. AEF uses the ADF-decisions to enforce the operations of system call functions.
Also in the RSBAC system, the access control system of the Linux system kernel is divided into the AEF and ADF components and the ACI-module. All RSBAC framework components are hard-linked into the Linux kernel. Figure 1 shows the interactions between the system components. For each security-relevant system call with which a subject requests to access an object, AEF sends a decision request to ADF. Parameters of the decision request are the request type, describing the desired type of functionality, the identification of the calling process (the subject) and possibly some identification values of the target of access (the object).
Targets of an access can be files, directories, named pipes (fifos), symbolic links, devices, interprocess communication data (ipc), system control data (scd), users, processes or none (see Table 1). This list includes the extra targets FIFO, SymLink and Device, which had been missing in LaPadula's specification in [LaPadula 1995]. Hence, named pipes and symbolic links as well as direct access to devices had previously not been properly covered by access control.

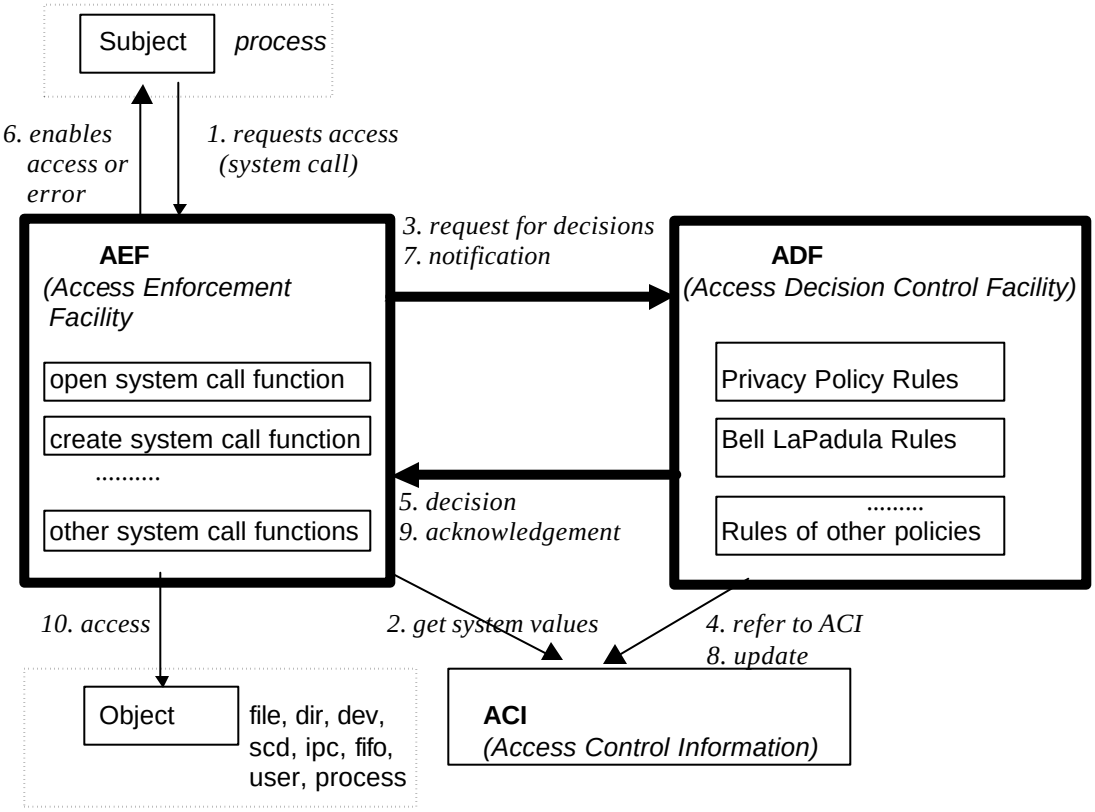

Fig. 1. RSBAC system architecture

ADF evaluates its security policies by using the policy rules for the request type and the ACI needed for these rules. It then evaluates its metapolicy, which uses the decisions of the different security policies, to finally decide about the process´ request. As different policies often return different results, a metapolicy is needed to build the final result. ADF currently has a fixed, restrictive metapolicy, which denies access, if at least one policy returned NOT_GRANTED. Later versions might make the metapolicy configurable as well.

AEF then enforces the decision, by either performing the system call functionality or returning an error to the calling process. In the first case, after successful execution, ADF is notified, so that the attributes can be set accordingly. Finally, control is returned to the process.

In LaPadula´s GFAC specification security attributes were updated by AEF after AEF received information through the decision message from ADF about how security attributes had to be set. In the RSBAC system, however, all policy-dependent functionalities are implemented by ADF and ACI. AEF is designed as a policy-independent component and hence security attributes are set by ADF. After successful execution of a system call, ADF is notified by AEF, so that all decision modules can adjust their attributes accordingly.

Table 1: Targets in RSBAC

FILE	Files identified by device and inode numbers (including device special files)
DIR	Directories identified by device and inode numbers
FIFO	Named pipes (FIFOs), which reside as special files on the file system, also identified by device and inode numbers
SYMLINK	Symbolic link special files, also identified by device and inode numbers
DEV	Devices identified by type (char or block), major and minor numbers
IPC	Inter Process Communication: Semaphores (sem), Messages (msg), Shared Memory (shm), Sockets (sock), identified by different IDs, e.g. object memory location
SCD	System Control Data: Objects affecting the whole system (e.g., system time and date, system log, host names). This target type is the only one with a fixed number of objects, identified by numbers
USER	Users as objects, mostly for access control information (ACI), identified by user ID
PROCESS	Processes as objects, identified by process ID
NONE	No object associated with the request.

3. ACI Module

The ACI component and the ADF are encapsulated into independent modules, which can only be legally accessed by the use of well-defined functions and are thus protected from unauthorized accesses. Because of the monolithic Linux kernel structure, this protection could of course still be violated by direct memory access from within the kernel. Thus all kernel code, especially run-time loadable kernel modules, must be properly checked before inclusion.
The ACI module is responsible for a reliable administration of security attributes of processes (process ACI), of users (user ACI) and of all resources that are needed and controlled by the security policies (object ACI). Besides, it administrates additional access control information needed for the implementation of specific security policies.
User and file ACI are kept in parallel in main and on secondary storage, so that they can be easily recovered after each system start. For this purpose, a kernel daemon regularly checks for updated data and writes it to files in specially protected directories in the file system.
Through the use of the Linux Virtual File System Switch, a file system function virtualization layer, the storage format of user and file ACI on secondary storage is independent from the used file system.

An infrastructure for generic persistent lists is also available, which can be used by additional decision modules. This helps to avoid individual ACI implementations. Existing decision modules are currently moved to this new scheme.

4. AEF and its Interface to ADF

AEF is implemented by extending all security-relevant system call functions with ADF requests. For each security-relevant system call, there is at least one ADF-request that relates to its functionality, but also further requests might be necessary. For example, the open system call has

to be extended with ADF-requests for reading a directory, creating a directory/file, truncating a file and for opening a file in append, read, write or read-write mode.
Before access to a target is granted to a subject, a request function call to the Access Control Decision facility (ADF) is performed. Dependent on the type and the target of a request, ADF decides whether access is granted or denied to a subject. Table 2 lists requests that AEF can send to ADF, valid target types for the requests as well as Linux system calls that are extended to use these requests. Valid target types for a request added to a system call are written in brackets behind the system calls in Table 2. Those system calls that also perform ADF notification are marked with *. The list of requests in Table 2 is an extension and modification of a list of requests in [LaPadula 1995].
In addition to the system calls and requests listed in Table 2, some policy specific system calls and requests were defined.
The logic behind the mapping of system calls to ADF requests should be obvious in most cases. Nevertheless, some of the mapping decisions are now justified.

Some requests only take place under certain conditions or with certain system call parameters, e.g. the EXECUTE request in sys_mmap is only issued, if a file is to be mapped in EXEC mode. This special mode, which is often used for libraries, is not covered by the preceding intercepted read-open.

Table 2: List of Requests from AEF to ADF

Request	**Description**	**Valid Target Types**	**Linux system calls** (valid target types in brackets)
ADD_TO_KERNEL	Add a kernel module	NONE	create_module(NONE), init_module(NONE)
ALTER	Change IPC control information	IPC	msgctl(IPC), shmctl(IPC),
APPEND_OPEN	Open to append	FILE, DEV, IPC	open(FILE,DEV)*, msgsnd(IPC)*, sendto(IPC)*, sendmsg(IPC)*
CHANGE_GROUP	Change active group	IPC, PROCESS, NONE	setgid(PROC), setregid(PROC), setresgid(PROC), setgroups(PROC), setfsgid(NONE) (for DAC only), shmctl(IPC), msgctl(IPC)
CHANGE_OWNER	Change owner	FILE, DIR, IPC, PROCESS, NONE	chown(FILE, DIR, FIFO, SYMLINK), lchown(FILE, DIR, FIFO, SYMLINK), fchown(FILE, DIR, FIFO, SYMLINK), setuid(PROC)*, setreuid(PROC)*, setresuid(PROC)*, setfsuid(NONE) (for DAC only), shmctl(IPC), msgctl(IPC)
CHDIR	Change working directory	DIR	chdir(DIR), fchdir(DIR), chroot(DIR)
CLONE	Fork/clone a process	PROCESS	fork(PROC)*, vfork(PROC)*, clone(PROC)*
CLOSE	Close opened file etc. Should always be granted.	FILE, DIR, FIFO, DEV, IPC	close(FILE, DIR, FIFO, DEV, IPC), shmdt(IPC)*, msgrcv(IPC)*, msgsnd(IPC)*, send(IPC)*, sendto(IPC)*, sendmsg(IPC)*, recv(IPC)*, recvfrom(IPC)*, recvmsg(IPC)*

CREATE	Create object	DIR (where), IPC	creat(DIR, IPC)*, open(DIR, IPC)*, mknod(DIR)*, mkdir(DIR)*, symlink(DIR)*, shmget(IPC)*, msgget(IPC)*, socket(IPC)*, accept(IPC)*
DELETE	Delete object	FILE, DIR, IPC	unlink(FILE, DIR, FIFO, SYMLINK)*, rmdir(DIR)*, msgctl(IPC)*, shmctl(IPC)*, shutdown(IPC)*. close(IPC)*
EXECUTE	Execute file or mmap in EXEC mode	FILE, NONE (mprotect only)	exec(FILE)*, mmap(FILE) (EXEC mode), mprotect(FILE, NONE) (EXEC mode)
GET_PERMISSIONS_DATA	Read Unix permissions (mode)	FILE, DIR, FIFO	access(FILE, DIR, FIFO, SYMLINK)
GET_STATUS_DATA	Get status (stat() etc.)	FILE, DIR, FIFO, IPC, SCD	open_port(SCD) (/dev/kmem etc.), open_kcore(SCD) (/proc/kcore), stat(FILE, DIR, FIFO, SYMLINK, IPC), newstat(FILE, DIR, FIFO, SYMLINK, IPC), lstat(FILE, DIR, FIFO, SYMLINK, IPC), newlstat(FILE, DIR, FIFO, SYMLINK, IPC), fstat(FILE, DIR, FIFO, IPC), newfstat(FILE, DIR, FIFO, SYMLINK, IPC), stat64(FILE, DIR, FIFO, SYMLINK, IPC), lstat64(FILE, DIR, FIFO, SYMLINK, IPC), fstat64(FILE, DIR, FIFO, SYMLINK, IPC), statfs(FILE, DIR, FIFO, SYMLINK), fstatfs(FILE, DIR, FIFO, SYMLINK), rsbac_stats(SCD), rsbac_check(SCD) , rsbac_stats_pm(SCD), rsbac_stats_rc(SCD), rsbac_stats_acl(SCD), rsbac_log(SCD)
LINK_HARD	Hard link	FILE, DIR, FIFO	link(FILE, DIR, FIFO, SYMLINK)
MODIFY_ACCESS_DATA	Change access information, e.g. time, date	FILE, DIR, FIFO	utimes(FILE, DIR, FIFO, SYMLINK)
MODIFY_ATTRIBUTE	Change an RSBAC attribute value	All target types	(specific request needed for various security models)
MODIFY_PERMISSIONS_DATA	Change Unix permissions	FILE, DIR, FIFO, SCD	ioperm(SCD), iopl(SCD), chmod(FILE, DIR, FIFO, SYMLINK) , fchmod(FILE, DIR, FIFO, SYMLINK)
MODIFY_SYSTEM_DATA	Change system settings	SCD	stime(SCD), settimeofday(SCD), adjtimex(SCD), sethostname(SCD), setdomainname(SCD), setrlimit(SCD), syslog(SCD), sysctl(SCD), swapon(SCD), swapoff(SCD), rsbac_log(SCD)
MOUNT	Mount a filesystem	DIR, DEV	mount(DIR, DEV) (separate mount notification for data structures)
READ	Read from object	FILE, DIR, FIFO, DEV, IPC	read(FILE, FIFO, DEV, IPC)*, readv(FILE, FIFO, DEV, IPC)*, pread(FILE, DEV, IPC)*, readdir(DIR), open(DIR)
READ_ATTRIBUTE	Read RSBAC attribute value	All target types	(specific request needed for various security models)

READ_OPEN	Open for read	FILE, FIFO, DEV, IPC	open(FILE, FIFO, DEV, IPC)*, shmat(IPC)*, msgrcv(IPC)*, recv(IPC)*, recvfrom(IPC)*, recvmsg(IPC)*
READ_WRITE_OPEN	Open for read and write	FILE, FIFO, DEV, IPC	open(FILE, FIFO, DEV, IPC)*, shmat(IPC)*, bind(IPC)*, connect(IPC)*, listen(IPC)*
REMOVE_FROM_KERNEL	Remove kernel module	NONE	delete_module(NONE)
RENAME	Rename	FILE, DIR, FIFO	rename(FILE, DIR, FIFO, SYMLINK) (RSBAC identification not changed!)
SEARCH	Lookup in DIR from inside kernel for access with full path, used by many syscalls, follow SYMLINK	DIR, SYMLINK	(internal functions lookup_dentry(DIR) / path_walk(DIR) / lookup_hash(DIR) / follow_link(SYMLINK))
SEND_SIGNAL	Send a signal, including KILL	PROCESS	kill(PROC)
SHUTDOWN	Shutdown/reboot system	NONE	reboot(NONE)
SWITCH_LOG	Change RSBAC log settings	NONE	rsbac_adf_log_switch(NONE)
SWITCH_MODULE	Switch decision module on/off	NONE	rsbac_switch(NONE)
TERMINATE	End of calling process, for ACI cleanup. Should always be granted.	PROCESS	exit(PROC)
TRACE	Trace a process	PROCESS	ptrace(PROC) (architecture dependent)
TRUNCATE	Truncate	FILE	open(FILE)*, truncate(FILE)*, ftruncate(FILE)*, truncate64(FILE)*, ftruncate64(FILE)*
UMOUNT	Umount a filesystem	DIR, DEV	umount(DIR, DEV) (separate umount notification for data structures)
WRITE	Write to object. DIR is used for object moving to target dir.	FILE, DIR, FIFO, DEV, IPC, SCD	write(FILE, FIFO, IPC, DEV)*, writev(FILE, FIFO, IPC, DEV)*, pwrite(FILE, IPC, DEV)*, rename(DIR), rsbac_write(SCD)
WRITE_OPEN	Open for write	FILE, FIFO, DEV, IPC	open(FILE, FIFO, DEV, IPC)*

A distinction is made between *read* and *read-open* and between *write* and *write-open*. Read-open (write-open) enables the process to *read* (*write*) the object, whereas *read* (*write*) actually transfers data from (to) the open object into (from) the memory space of the process. ADF´s security policies for controlling read (write) access can be applied at the read-open (write-open) as well as (optionally) at the read (write).

Leaving out the check of the read (write) access itself reduces the access control overhead significantly, because the number of necessary checks is much reduced. On the other hand, changes in the ACI can then only be enforced at the next open request.

No operations of the ipc mechanisms for message queues and shared memory are, however, equivalent to the file open and close system calls. The *msgget* and *shmget* system calls are

similar to the *creat* and *open* system call, because they return a kernel-chosen descriptor for use in other system calls. Nevertheless, even if a process never did a "get" call, it can access an ipc message mechanism if it guesses the correct ID and if access permissions are suitable.
For consistency with accesses to other object types, the *msgsnd* system call is extended with an append-open and a close ADF-request, although *msgsnd* is actually more analogous to the actual write operation. Similarly, *msgrcv* system call is extended with a read-open and a close ADF request.
Also, socket system calls do not strictly follow the traditional Unix open-read-write-close paradigm. The *connect* system call, which binds a permanent destination to a socket, and the *listen* system call correspond to the open system call. However, sockets used with connectionless datagram services need not be connected before they are used, if a destination is specified for each data transfer. Thus, for consistency the system calls *sendto* and *sendmsg*, which both allow the caller to send a message through an unconnected socket, are extended with append-open and close ADF-requests. Similarly, the system calls *recvfrom* and *recvmsg* are extended with read-open and close ADF-requests.
The *shmat* and *shmdt* system calls for attaching and detaching of shared memory to the virtual address space of a process are in a way analogous to the file *open* and *close* system calls. The shmat system call has to be called before a process can access shared memory, much as the *open* system call has to be executed before a process has access to a file. After attaching a shared memory, it becomes part of the virtual address space of the process and can be accessed. However, in contrast to the *open* system call for files, after the *shmat* system call, no system calls are needed to access data in shared memory, which are accessible in the same way as other virtual addresses are.

5. Security Policies in ADF

The different security policy modules that have been implemented as part of ADF are listed in Table 3. A RSBAC system can be configured with any combinations of these policy components.
In addition to the policy of the traditional Bell LaPadula model, the SIM policy and the FC policy, further security model policies that were developed by the authors have been implemented in the RSBAC system. In the next subsections, some of those policies are described in more detail.

Table 3: Security Policies included in the RSBAC implementation

Policy Module Name:	**Implemented Security Policy:**
MAC	Mandatory Access Control Policy of the Bell LaPadula Model [Bell LaPadula 1973]
FC	Functional Control Policy (see [Abrams et al. 1991], [LaPadula 1995]). A simple role based model that can be used to restrict access to security information to security officers and access to system information to administrators.

SIM	Security Information Modification Policy (see [Abrams et al. 1991], [LaPadula 1995]). Only security administrators are allowed to modify data labeled as security information
PM	Simone Fischer-Hübner's Privacy Model policy (see below)
MS	Malware Scanner policy (see below).
FF	File Flags Policy. The FF policy can be used to set the following access flags for directories and files: execute_only (files), read_only (files and directories), search_only (directories), secure_delete (files), no_execute (files) and add_inherited (files and directories). The flags are checked at every access. Only security officers may modify these flags. If the add_inherited flag is set, the parent directory´s flags are added to the target's own flags. (Example: If no_execute is set on the directory /home, all executables below that directory inherit this flag. Thus no user can execute files from her home directory, unless the flag is removed).
RC	Amon Ott´s Role Compatibility Model policy (see below).
AUTH	Authorization enforcement policy (see below).
ACL	Access Control List policy (see below).

Privacy Policy :

The privacy policy module is enforcing the security policy of a formal task-based privacy model which was introduced in [Fischer-Hübner 1994], [Fischer-Hübner/Ott 1998], [Fischer-Hübner 2001]. The privacy model was designed to protect personal data and can be used to enforce legal privacy requirements such as necessity of personal data collection and processing and purpose binding.

The basic idea of the privacy policy is that subjects (processes) can only access personal data objects by performing a task (e.g., diagnosing, patient admission) that serves a specific purpose. Each personal data object is classified by a class (e.g., medical diagnosis data or accounting data), which has specified purposes for which objects of that class are obtained. The privacy policy can be informally stated as follows:

A process may have access to personal data, if this access is necessary to perform its current task and only, if the user owning the process is authorized to perform this task. The process may only access data in a controlled manner by performing a (well-formed and certified) transformation procedure, for which the process´ current task must be authorized. Besides, the purpose of its current task must correspond to the purposes for which the personal data of this class are obtained or there has to be consent by the data subjects that their personal data may be used for the purpose of the current task.

Note that other known security models (such as the RBAC, the Bell LaPadula model or the Clark Wilson model) are not in all respects adequate to protect personal data. In particular the privacy principle of purpose binding cannot be adequately enforced by other models, which are not modeling purposes and consents of data subjects [Fischer-Hübner 2001].

A complete privacy model description and specification of ACI and ADF-rules needed for implementing the privacy policy are given in [Fischer-Hübner/Ott 1998], [Fischer-Hübner 2001].

Malware-Scanner Policy:
The Malware Scanner (MS) Policy can protect the system against malware (viruses, malicious applets, etc.) infections by preventing the execution, reading and transmission of malicious code (see also [Ott/Fischer-Hübner/Swimmer 1998]). It is enforced by on-access scanner policy rules that are invoked on EXECUTE- and open-ADF requests and can detect and stop the spread of malware. The malware scanner policy is similar to the methodology incorporated in many Antivirus products, except that it is implemented in the kernel and is therefore much better protected from manipulations. Besides, it contains socket-level scanner policy rules that are invoked on READ-requests to network sockets and can stop the transmission of malware into the system by a network connection.

To enforce the on-access-scanner policy rules, the following security attributes grouped into process-ACI and object-ACI are needed and administrated by the ACI-Module.

```
CASE read-open, execute
    SELECT CASE target[input-argument]
        CASE file
            IF type(process) is MS-trusted
            THEN
                return(YES)
            SELECT CASE MS-scan-result(object)
                CASE unscanned
                    IF SCAN(object) is infected
                    THEN
                        return(set-attribute (MS-scan-result(object), rejected); NO)
                    ELSE
                        return(set-attribute (scan-result(object),
                        version-number(scanner));YES)
                CASE scanned
                    IF version-number(scanner) > version-number(object)
                    THEN
                        [IF SCAN(object) is infected
                        THEN
                            return(set-attribute (scan-result(object), infected); NO)
                        ELSE
                            return(set-attribute (scan-result(object), version-
                            number(scanner)); YES)]
                    ELSE
                            return(YES)
                CASE rejected
                            return(NO)
                CASE ELSE
                    return(UNDEFINED)
        CASE ELSE
            return(UNDEFINED);
```

Fig. 2. ADF-On-Access Scanner Policy Rule for read-open and execute requests

For each file, the security attribute *MS-scan-result* is used, which can have “scanned”, “rejected” or “unscanned” as possible values. It has the value ”scanned”, if the file was scanned and no infection was found. In this case, the file has also a version-number as a security attribute that corresponds to the version number of the used malware scanner. If the file was scanned and a

malware infection was detected, the file attribute is set to “rejected”. It is set to “unscanned,” if the file has not been scanned so far, or if the file has been modified by the last access to it. Besides, a new attribute *MS-trusted* is used, which is set for the antivirus or backup programs and processes that are executing those programs. The ACI-module also administrates a database of virus patterns/signatures of a malware scanner, which has a version number attached to it.

The scanner policy rules are invoked at execute or open decision requests sent to ADF by AEF. Infections by file viruses are caused by the execution of infected files, whereas reading an infected document causes infections by macro viruses. Thus, each request from AEF to ADF to execute or to read a file should only be positively decided by ADF, if the file has first been scanned with the latest scanner version and if no infections were found or if the process is MS-trusted

Figure 2 specifies the ADF-On-Access Scanner Policy Rule that is invoked at READ-OPEN or EXECUTE requests. The specification language used should be intuitively understandable to computer scientists, but is also explained in [LaPadula 1995]. SCAN is the scanner function that scans an object and returns a scan result.
For the modifying access request types WRITE-OPEN, APPEND-OPEN and TRUNCATE, no decision rules are defined. However, for these and for READ-WRITE-OPEN, the MS-scan-attribute value is reset to “unscanned”, because the file might have been infected or disinfected.

The on-access scanner policy provides a reliable and tamper-proof protection against known malware in executable files. Increasingly however, malware is brought into the system by a network connection, often executed without ever being saved to a file. Therefore, we extended our approach to include scanning and denying access to data that seems to contain malicious code from a given origin of network connections by controlling UNIX-type sockets (see [Ott/Fischer-Hübner/Swimmer 1998]).
Additional socket-level scanner policy rules control all read accesses to network connection sockets. The rules are invoked by requests sent to ADF for the access type READ and target IPC, which were added to the TCP and UDP read functions. If malware is detected in the data stream, further read access is denied and a new error code "malware-detected" is returned. Depending on the configuration, the connection can be closed by the kernel, or a trusted process can still allow the transmission of the whole data stream.

Role Compatibility (RC) Policy:
The Role-Compatibility Model policy can be used to define roles as a set of access permissions to compatible object types, and is most useful for secure system administration. It is similar to the Domain and Type Enforcement (DTE) approach [Badger et al. 1995].
In the RC model, objects of the target type FILE, DIR, DEV, IPC, SCD, PROCESS can be categorized according to object types. The set of access modes corresponds to the set of request types listed in Table 2. Roles are defined as sets of access permissions to objects of certain types in certain access modes. Each process is currently performing one role. The initial role of a process is determined by a default role that is defined for its owner.
One essential rule of the security policy can be stated as follows:

A process can only access an object in a certain access mode if its current role is authorized to access objects of that type in that mode (we say that the role has to be "compatible" to the object type and access mode).

It is also possible to change the current role of a process if the current role is "compatible" with the new role or if the "forced role" mechanism is used.
For a role R, a set of so-called compatible roles can be defined, to which a process performing role R is authorized to switch. Once the role has been changed, there is only a way back to the first role, if the new role is in turn compatible with the first role. This mechanism allows a process to temporarily work with different privileges.
A forced role is an attribute of an executable stating which role a process gets when running this executable. Forced roles can forbid processes to access data arbitrarily, and allows them to access data only in a constrained way by performing certain well-defined programs. The forced role concept can for instance be used to restrict the privileges of the root account in a Unix system, e.g. to restrict access to authorisation information (/etc/passwd, /etc/shadow) to certain programs with a forced role "Authorisation". Forced roles are also very useful for CGI programs of tightly controlled Web servers.

Access Control List (ACL) Policy:
Because of good experiences with and wide knowledge of the access control list scheme of a popular PC network system, this scheme as been adapted for RSBAC implementation. However, some important changes have been made to overcome known disadvantages (e.g. the omnipotent Supervisor) and to provide some interconnection with the Role Compatibility policy.
The Access Control List policy uses an access control list (ACL) for every object of the target types FILE, DIR, DEV and SCD. If there is no ACL entry for a subject – object pair, the ACL entry of the object parent is inherited. Inheritance is terminated by a default ACL, which is defined for every target type, including USER, PROCESS, IPC.
Every user can define groups of users for access control. A user that defines a user group becomes the owner of that group. All group settings like name or memberships can only be modified by the group owner. If a group is marked as global, its settings can be read and it can be used/set as a subject in access control lists by all users, while the settings of a private group can only be read by the group owner and only the owner can set ACL entries with its private group as a subject .

An ACL entry consists of a pair (Subject, Access Modes). Subjects can be users, RC policy roles or ACL groups. Access Modes is a subset of the set of all request types (so called "normal rights", see table 2) and all so called ACL special rights.
ACL special rights are Forward (set normal rights this user has for others), Access Control (set any normal right for anyone) and Supervisor (contains all rights, can set special rights for anyone).
A process has the right to access an object in a certain mode, if the object´s ACL has an entry with either the user owning the process, one of the user´s groups or the process´ RC role as the first entry component and the access mode as the second component.
An object can be protected against inheritance of rights given from a higher hierarchy level by its inheritance mask. This mask defines, which rights may be inherited. Its default value is of course

the whole set of rights. As this has always been a popular way of locking yourself out, the Supervisor right can only be masked out under special circumstances.

Authorization Policy:
Linux, like many other Unix like systems, has well known deficiencies at user authorization. In fact, every process running as root is allowed to set its owner to any possible user ID. This makes the enforcement of security with access control based on user IDs almost impossible. The Authorization policy was developed to protect against this vulnerability.
The basic idea is simple: No process is allowed to set its owner to another user ID, unless it has been explicitly granted this right called capability. In this authorization policy model, a capability is defined as a range of user IDs (first, last) to which a process running a certain program has the right to change. The capability for all user IDs can also be set by an extra switch 'auth_may_setuid'. Capability sets are defined for executables and inherited by every process running them.
Capabilities may be set by users with system role 'Security Officer'. Capabilities for processes may also be set by other processes with the extra switch 'auth_may_set_cap' turned on. The latter feature can be used to implement authentication daemons, which grant a capability to a process only after proper authentication.
As this model is so important for all other models, MODIFY_ATTRIBUTE decision requests with special SCD targets 'add_file_cap' etc. are enforced whenever a process tries to add or delete a capability. This way, each model can perform its own access control for all capability modifications.

6. The RSBAC logging facility

A facility for extensive and model independent logging has been implemented. It is possible to specify the event to be logged in dependence of the request type, user, executable and target file, fifo, symlink, directory or device objects. All log settings are at the granularity of request types.
Logged items are the request, process ID, program name, real or pseudonymous user ID, target type, target ID, attribute type, attribute value, ADF decision and the names of the modules that made this decision.
General settings for request types can determine whether to log none, denied requests only or all requests of that type. Object based settings have the additional default value 'request based', which means that the request setting should be used.
The user and the executable based logging can only be turned on or off for a specific request. If it is turned off, logging might still be triggered by the object log settings.
All accesses to log settings are controlled. Each model can implement its own access control scheme for logging administration.

7. System Performance

Some benchmarks have been run on a SuSE Linux 7.0 pentium class system with kernel 2.2.18 and with RSBAC version 1.1.0. Three benchmark runs per kernel type, in single user mode right after boot, were performed under a clean kernel, a kernel with RSBAC framework, but without decision modules, and an RSBAC kernel with modules FF, AUTH, RC and ACL.

Each benchmark run consisted of a full compile of the same set of kernel sources.Kernel compile benchmark are widely used, because they combine a lot of file accesses with significant memory pressure and a high system load. The following average times were produced:[2]

Kernel type	Total time	Kernel + User	Kernel time	User/Process time
Clean kernel	1858s	1857s	69s	1788s
RSBAC without modules	1884s (+1.3%)	1877s (+1.1%)	82s (+18.8%)	1795s (+0.4%)
RSBAC with FF, AUTH, RC, ACL modules	1967s (+5.9%)	1959s (+5.5%)	167s (+142%)	1792s (+0.2%)

Based on these numbers and practical experience, we regard the RSBAC overhead as small for real systems.

Although enhanced access control is mostly important on big multi-user server systems, the additional memory usage is still relevant. To reduce the memory burden, many RSBAC features and all decision modules can be left out at kernel configuration and compilation.
Also, attribute data objects are only allocated on demand, and those containing only default values or relating to deleted objects are automatically removed.

Additional RSBAC code and static data increase the uncompress kernel size by approximately 130 to 600 KB, plus the memory consumed for attribute objects.

8. Other Linux security extensions

8.1 Linux Privileges

Recent Linux kernels additionally implement a privilege scheme, which splits the root user's special rights into a set of single rights, called capabilities (see [Linux-Privs 2000]). These rights are given to a process based on the parent process and the executable that is run.
However, while these capabilities can distinguish between some access types, they are mostly ignorant of the object that is to be accessed, e.g. CAP_DAC_OVERRIDE gives full read and write access to all files and devices on the system. As a result, many administration tasks still have to be done with too many access rights. Another disadvantage is the fixed access control model, which cannot easily be changed or replaced.

[2] Times measured via time(1) utility.

8.2 Flask / NSA Security Enhanced Linux (SELinux)

Security-Enhanced Linux (SELinux), which is currently developed by NSA, is a Linux version that is in a similar way as RSBAC also incorporating a flexible mandatory access control in its kernel. In a previous research project, the Flask Security architecture providing policy flexibility [Spencer at al. 1999] was developed and prototyped in the Mach and Fluke research operating systems. Like the RSBAC system architecture, the Flask architecture is also separating the enforcement mechanism from the policy decision mechanism and is consistent with GFAC. It includes a security policy server to make access control decisions and object managers to enforce access control decisions. In the SELinux project, the NSA is integrating the Flask security mechanisms into the Linux kernel and is now working with the NAI Labs in further developing and configuring the security-enhanced Linux system. The Linux prototype implementation of the Flask security server implements a policy that is a combination of Type Enforcement, role-based access control (RBAC), and optionally multi-level security. The SELinux prototype is so far only controlling operations on processes, files, directories and sockets. For the protection of System V IPC objects, there is a preliminary design concept.

While RSBAC is a complete system that runs on different platforms and has already been used for the last year in several security-relevant applications, SELinux is so far a prototype which is still under major development. So far, SELinux only supports the x86 architecture and has only been tested on Red Hat distributions. Besides, in contrast to RSBAC, it does not offer a set with a wide variety of different security policies to which new policy components can be easily added and from which an arbitrary subset can be chosen for system configuration. Although Type Enforcement in combination with RBAC is powerful and capable to enforce a wide range of policies, there are a number of security policies implemented in RSBAC that cannot or at least cannot easily be expressed with Type Enforcement and RBAC.

8.3 Medusa

The Medusa DS9 security system is another security extension for Linux kernels [Medusa]. It consists of a patch to the Linux kernel and a user space daemon that acts as an authorization server that is invoked by the kernel to authorize operations. The authorization server is implemented as an interpreter of its own configuration language and is thus capable to implement different security models. While the user space implementation of the authorization server allows to easier port Medusa to new Linux kernel versions, it is less secure than a kernel-based solution.

8.4 Linux Intrusion Detection System (LIDS)

Linux Intrusion Detection System (LIDS) [LIDS], which has been developed by a Chinese and a French student, is a patch that enhances the Linux kernel's security by implementing a reference monitor with mandatory access control. In addition, it includes a network port scanner detector and a response mechanism that allows to log and to act upon security violations by shutting down a user's session. However, in contrast to RSBAC, LIDS offers only a static access control policy and thus less flexibility.

9. Outlook

At the moment, the network access control is completely redesigned. There will be network target templates, the attributes of which are inherited for every network connection instance.
The templates will contain fields like source address range, target address range, interfaces (to protect against spoofing attacks), Protocols, source ports, target ports and some more.
The templates as well as the connection instances will be treated as normal access objects with a full range of requests, like CREATE, CONNECT, LISTEN, etc. This way, each decision policy can perform full control of all network accesses by all processes.

As an example, think of a local network with well controlled systems. A connection template could contain the following settings and attributes:
- Source Address Range: (interface address for internal net)
- Target Address Range: (local network)
- Interfaces: (local network interface)
- Protocols: All
- Source Ports: 0-65535 (all)
- Target Ports: 0-65535 (all)
- MAC Security Level: Secret
- RC Network Object Type: 4 (Internal Net)

All other addresses at the outer interface of a firewall could have the following template:
- Source Address Range: (External Network address)
- Target Address Range: (All)
- Interfaces: (external network interface)
- Protocols: All
- Source Ports: 0-65535 (all)
- Target Ports: 0-65535 (all)
- MAC Security Level: untrusted
- RC Network Object Type: 0 (untrusted)

There is also work in progress to get a generic access control interface into the official Linux kernel, which would greatly reduce development work.

10. Final Remarks

Linux systems are increasingly used in mission critical environments where a lack of security can be very risky for a company. Hence, access control under Linux will most likely gain more and more relevance and interest in future. In comparison to other kernel-based access control systems for Linux, RSBAC has the advantage that it offers a wide range of security policies and that it easily allows to define and add additional policy modules for almost any new security policy.
The development of RSBAC will be continued. It is used in more and more server systems, and there are currently 30-100 downloads of the latest release per week, with a clear peak after the

announcement of a new version. Also, there is a mailing list for discussion and announcements with more than 90 participants.
The single processor versions as well as the latest version for Symmetric Multiprocessing (SMP) kernels have shown to be very stable and have been used in production environment for at least one year.[3] Benchmark tests and practical experience has also shown that performance impacts are small, unless a lot of logging has to be done.

References

[Abrams et al. 1990] M.Abrams, K.Eggers, L.LaPadula, I.Olson, "A Generalized Framework for Access Control: An Informal Description", Proceedings of the 13th National Computer Security Conference, Washington, October 1990.

[Abrams et al. 1991] M.Abrams, L.LaPadula, M.Lazear, I.Olson, "Reconciling a Formal Model and Prototype Implementation – Lessons Learned in Implementing the ORGCON Policy", Mitre Corporation, Bedford, Mass.01730, November 1991.

[Bell LaPadula 1973] D.E.Bell, L.LaPadula, "Secure Computer Systems: A Mathematical Model", Mitre Corporation, Bedford, Mass.01730, January 1973.

[Fischer-Hübner 1994] S.Fischer"Towards a Privacy-Friendly Design and Use of IT-Security Mechanisms", Proceedings of the 17th National Computer Security Conference, Baltimore MD, October 1994.

[Fischer-1998a] A.Ott, "From a Formal Privacy Model to its Implementation", Proceedings of the 21st National Information Systems Security Conference, Arlington, VA, October 5-8, 1998, http://www.rsbac.org/niss98.htm.

[Ott/Fischer-Hübner/Swimmer 1998b] A.Ott, Fischer-"Approaches to Integrated Malware Detection and Avoidance", Proceedings of the 3rd Nordic Workshop on Secure IT Systems, Trondheim, November 5-6, 1998, http://www.rsbac.org/nordse98.htm.

[Fischer-Hübner 2001] "IT Security and Privacy - Design and Use of Privacy-Enhancing Security Mechanisms", Springer Scientific Publishers, Lecture Notes of Comp uter Science, LNCS 1958, May 2001.

[LaPadula 1995] L.LaPadula, "Rule-Set Modeling of Trusted Computer System", Essay 9 in: M.Abrams, S.Jajodia, H. Podell, "Information Security - An integrated Collection of Essays", IEEE Computer Society Press, 1995.

[LIDS] Linux Intrusion Detection System (LIDS), http://www.lids.org/

[Linux-Privs 2000] Linux Privileges, http://www.kernel.org/pub/linux/security/linux-privs/

[Loscocco et al. 2001] P.Loscocco, S.Smalley, "Integrating Flexible Support for Security Policies into the Linux Operating System", Proceedings of the FREENIX Track of the 2001 USENIX Annual Technical Conference, http://www.nsa.gov/selinux/

[3] E.g. in Compuniverse file server and several customer firewalls. For security reasons, references can only be given on personal request.

[Medusa] Medusa DS9 security system, http://medusa.formax.sk

[Ott 2001a] Amon Ott: "Rule Set Based Access Control (RSBAC), Snow Unix Event / unix.nl congress "Reliable Internet", Waardenburg, 14th of September 2001, http://www.rsbac.org/unix-nl/

[Ott 2001b] Amon Ott: "The Rule Set Based Access Control (RSBAC) Linux Kernel Security Extension, 8th International Linux Kongress, Enschede, 28th to 30th of November 2001, http://www.rsbac.org/linux-kongress/index.html

[RSBAC] Rule Set Based Access Control in Linux, http://www.rsbac.org.

[Spencer et al. 1999] R.Spencer, S.Smalley, P.Loscocco, M.Hibler, D.Andersen, J.Lepreau, "The Flask Security Architecture: System Support for Diverse Security Policies", Proceedings of the 8th USENIX Security Symposium, Washington, D.C., August 23-26, 1999.