A COMPARISON BETWEEN ConSA AND CURRENT LINUX SECURITY IMPLEMENTATIONS

Alexandre Hardy
Computer Science, Rand Afrikaans University
PO Box 524, Auckland Park, Johannesburg, South Africa
ah@adam.rau.ac.za

Martin S Olivier
molivier@rkw.rau.ac.za
Computer Science, Rand Afrikaans University
PO Box 524, Auckland Park, Johannesburg, South Africa

Abstract There are many extensions to the Linux security model that are available. ConSA [1] aims to provide a configurable architecture, and should allow many security systems to be implemented. A prototype ConSA system has been implemented in Linux. This paper will examine how ConSA relates to currently available Linux security extensions.

Keywords: Access Control, Security, Security Model

1. INTRODUCTION

Linux is a popular operating system based on Unix [17]. It therefore uses the same security model that Unix uses. The Unix model is well-known and has been thoroughly tested and investigated. Unfortunately a single, fixed security model does not fulfil everyone's needs. This is evidenced by the large number of security models that have been proposed and even implemented for Linux [6, 7, 8, 9, 10, 11, 13, 14, 15, 16]. ConSA [1] is a model that addresses the requirements posed by diverse security needs by providing a high degree of configurability.

The purpose of this paper is to compare ConSA to the diverse range of Linux security implementations to show that ConSA does indeed provide the flexibility that is required as illustrated by these implementations. By comparing ConSA

The original version of this chapter was revised: The copyright line was incorrect. This has been corrected. The Erratum to this chapter is available at DOI: 10.1007/978-0-387-35587-0_24
M. S. Olivier et al. (eds.), *Database and Application Security XV*

to these implementations, the paper also gives an insightful overview of the security projects that are currently underway in the Linux environment.

This paper is structured as follows: Section 2 gives a basic overview of the ConSA model. Sections 3 to 7 discuss these Linux security implementations and how they relate to ConSA.

2. THE CONFIGURABLE SECURITY ARCHITECTURE

The Configurable Security Architecture provides a framework for the implementation and use of a security system. The architecture is illustrated in figure 1. Several modules are identified for implementation:

- Entity Labels — Entity Labels encode the security attributes for a resource in the system.
- Subject Labels — Subject Labels encode the security attributes of a subject or user of the system.
- Information Flow Manager — Information Flow concerns are implemented by this module.
- Authorization Control Module — This module determines the initial rights subjects have to newly created resources.
- Subject Management Module — The Subject Management Module maintains subject information and translates externally presented identifiers into Subject Labels.
- Message Dispatcher — This component is responsible for passing messages to the destination object, and enforcing the security model on those messages. This component is provided by ConSA.
- Protect Table — The Protect Table associates Entity Labels with resources. These resources are protected by the Entity Labels associated with them. This module is also provided by ConSA.

It is clear that the Subject Management Module (SMM), Authorization Control Module (ACM), Information Flow Manager (IFM), Subject Label and Entity Label implement the security model. The other modules will not be considered in the following discussion. These modules are implemented as objects, and each object must implement specific methods. The operation of ConSA as an access control mechanism is as follows:

- The Subject logs onto the system.

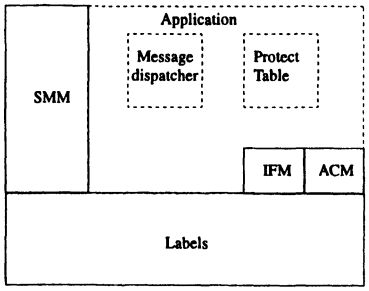

Figure 1. Configurable Security Architecture (from [1])

- A SubjectLabel is assigned to the subject by the Subject Management Module.

- If a subject wishes to access an entity, then the EntityLabel that protects that entity is determined by consulting the Protect Table. The EntityLabel determines if the SubjectLabel of the subject is allowed to access the object. The Information Flow Manager is consulted to make sure that this is a valid information flow. If these checks are passed, then the subject is granted access.

- If a subject creates a new entity, the Authorization Control Module creates an EntityLabel to protect the entity, and associates the EntityLabel with the entity in the ProtectTable. The EntityLabel is initialized to enforce the security policy on that entity.

- The subject logs off the system.

[1] provides a thorough discussion of the methods and requirements of each module. Here we briefly describe a few of these methods to facilitate the following discussion. Note that the definitions here are slightly different to [1].

2.1. `EntityLabel`

- `NoAccess()` — Remove all access from all subjects

- `GrantAccess(SubjectLabel` L_s`)` — Grant access to any subject that may use the label L_s.

- `RevokeAccess(SubjectLabel` L_s`)` — Revoke Access for subjects presenting this label. Subjects may still be able to gain access through other labels.

- `RevokeAccess*(SubjectLabel` L_s`)` — Revoke Access for subjects presenting this label. The subjects from whom access is revoked will not be able to access the entity with any label.

- `CheckAccess(SubjectLabel` L_s`)` — A test to determine if the subject presenting L_s may access the entity protected by this label.

2.2. `InformationFlowManager`

- `ActivateFlow(EntityLabel` L_e`, SubjectLabel` L_s`,` $Parameters$`)` — This method determines if access is allowed to entity e by subject s if s invokes a method of e. $Parameters$ specifies whether parameters are passed or not.

- `ReturnFlow(EntityLabel` L_e`, SubjectLabel` L_s`,` $ReturnVal$`)` — This method determines if access is allowed to entity e by subject s if e returns from a method invocation by s. $ReturnVal$ specifies whether information is returned or not. For languages such as C++ these methods may be evaluated before method invocation.

2.3. `AuthorizationControlModule`

- `EntityCreated()` — This method is invoked if an entity is created to provide an entity label that will protect the entity.

The following sections will analyse other security models, and determine if ConSA can achieve the same results. We begin with the standard Linux security model, since most of the extensions are further restrictions over and above the Linux security model.

3. STANDARD LINUX SECURITY

This section will focus on how the Linux security model may be implemented in ConSA. The Linux security system has no information flow control, and so a ConSA implementation of the Information Flow Manager will allow all operations. The implementation of the Subject Label and Entity Label will also illustrate that the security system is correctly implemented. Next we consider the implementation of each of the Subject Label and Entity Label methods.

3.1. Subject Label

The Subject Label stores the `uid` of the subject, this `uid` is used to obtain access to entities. Method implementations are as follows:

- `InitSubject` — This method will only need one parameter, the `uid` of the subject, which will be stored. For a Subject Label L_s the `uid` will be denoted as $L_s.uid$.

3.2. Entity Label

The entity label for this discussion will protect files. This discussion may be extended for other resources. The Entity Label stores a `uid` (the owner of the file) and a `gid` (the group owner of the file) as well as an ACL (Access Control List) in the form of a bit vector `rwxrwxrwx`. The first `rwx` refers to read, write and execute permissions for the user. The second `rwx` refers to permissions for the group, and the last `rwx` refers to the permissions of all other users. For an Entity Label L_e, $L_e.uid$ denotes the owner of the file, $L_e.gid$ denotes the group that owns the file. The user `rwx` bits will be denoted by $L_e.uid_r$, $L_e.uid_w$, and $L_e.uid_x$ (with values 0 or 1). A similar notation will be used for group bits `rwx`. Lastly, the protection bits for all other users in the system (`rwx`) will be denoted by $L_e.other_r$, $L_e.other_w$, and $L_e.other_x$. *invalid* will denote some invalid identifier, that will not match any identifier in the system. The methods of the Entity Label class can now be implemented as follows:

- `NoAccess()`

$$\begin{aligned} L_e.uid &:= invalid \\ L_e.gid &:= invalid \\ L_e.uid_r &:= 0 \\ L_e.uid_w &:= 0 \\ L_e.uid_x &:= 0 \\ L_e.gid_r &:= 0 \\ L_e.other_r &:= 0 \end{aligned} \tag{1}$$

  Invalid identifiers ensure that the other bit vector is used for access checks, some of the initialization has been omitted due to space constraints.

- `GrantAccess(SubjectLabel` L_s`)`
  `GrantAccess` will be restrictive and may fail, to ensure Linux access semantics. A typical implementation may be as follows:
  - If there is no valid user that owns the file, or $L_s.uid = L_e.uid$ then set the `uid` to that of the Subject Label, and set $L_e.uid_r$, $L_e.uid_w$, and $L_e.uid_x$ according to the supplied mode parameter. Access should be granted by this parameter, and not revoked.
  - Otherwise, if there is no valid group or $L_s.uid \in G(L_e.gid)$ then follow a similar process for group assignment. A group should

be selected according to $L_e.uid$. Usually each user has a group associated with it such that `uid=gid`, and this `gid` may then be used. First set the `uid` rights and then the `gid` rights.

- Lastly, if neither of the above hold then adjust the access to $L_e.other$.

Granting access for mode *mode* (r,w or x) to category c (uid, gid or $other$) is achieved by

$$L_e.c_{mode} := 1$$

- `RevokeAccess(SubjectLabel` L_s`)`
  This method is similar to GrantAccess, and will not be discussed due to space constraints.

- `RevokeAccess*(SubjectLabel` L_s`)`
  This method is similar to GrantAccess, and will not be discussed due to space constraints.

- `CheckAccess(SubjectLabel` L_s`)`
  This method simply consults the bit vectors according to the mode of access. Only read access will be considered. `CheckAccess` returns true for read access if

```
if (uid = 0) then
     return (true);
else if (L_e.uid = L_s.uid) then
     return (L_e.uid_r = 1);
else if (L_s.uid ∈ G(L_e.gid)) then
     return (L_e.gid_r = 1);
else return (L_e.other_r = 1);
```

  Note that the subject must have access to each of the parent directories that a file resides in to have access to that file.

The labels demonstrate that a non standard interpretation can be used, and the process still appears to be logical for the intended security model. The interpretation is inherently different due to the nature of the ACL implemented in Linux. Note also, that if the group can be modified, then users may be added to the ACL as necessary, however all users would have the same access rights. For example, to grant access to a file with gid *gid* to a subject with uid *uid*, simply set $G(gid) := G(gid) \cup uid$. The user with uid *uid* inherits the rights the rest of the group have. Another option to support more complex operations and perhaps to simplify implementation of command such as `chgrp`, is to implement an entity that provides access to labels, with the necessary security precautions. An entity may also be constructed to modify the group database. It should be clear that ConSA implements a superset of the Linux security model.

The Authorization Control Module must first grant access to the user creating the file, and then to a suitable group. A label must also be constructed to control who may grant or revoke access to the file. Naturally the super user is granted access by this label, and so is the owner of the file. Note that grant or revoke may be considered as another mode of access, and the algorithm for granting or revoking access will be different for these modes (since the concept of `uid` and `gid` no longer apply to this type of entity, namely labels). The Authentication Module may easily be written to authenticate users from the `/etc/passwd` file. The Subject Management Module also uses `/etc/passwd` to translate a user name to a uid.

4. LINUX TRUSTEES (ACL) PROJECT

The Linux Trustees(ACL) Project [14] extends the protection bit vector associated with files in Linux, and increases the flexibility of the algorithm used to determine if access is granted or not. The protection bit vector has been extended to the following:

Symbol	Meaning
R	Read files
W	Write files and directories
B	Browse (like UNIX execute for directories)
E	Read directories
X	Execute files
U	Use standard UNIX permissions
Modifiers	
C	Clear permissions instead of setting them
D	Deny access (instead of grant)
!	Complement of subject set (applies to all except user or group)
O	One level (does not apply to subdirectories)

This scheme extends the control over permissions, furthermore directories and files inherit the protection from directories higher up in the file tree, unless explicitly changed. The modified protection bit vector is known as a trustee object.

A ConSA implementation of the Linux Trustees system requires special attention — the inheritance of rights from a parent directory can be difficult to implement. The extension of the Entity Label representation (permission bit vector) is easily extended to include the extra permissions. Now there are a few factors to take into consideration:

- Multiple trustee objects — Multiple trustees can be handled in the same way that multiple ACL entries were handled in the ACL section.
- EntityLabel:CheckAccess — This method needs special attention, it may be implemented as specified in the first section. This includes the building

of the mask. Entity Labels would probably implement other methods for direct communication between Entity Labels.

5. LOW WATER-MARK MANDATORY ACCESS CONTROL (LOMAC)

This project assists in the protection of the integrity of processes and data, by enforcing Low Water-mark Mandatory Access Control policies. This project differs from the previous projects not only because of the policy implemented, but also because the implementation is in the form of a loadable kernel module. This means that the kernel need not be recompiled. The aforementioned projects do require recompilation of the Linux kernel. LOMAC extends the Unix security system, and does not replace it.

A LOMAC policy is made up of a series of levels or classifications $a_1, ..., a_n$ with $a_i \in \mathbf{N_0}$. Each object is assigned a level or classification at the time of creation. Objects include files, pipes, sockets, and semaphores. If an object with level a_i is considered to have lower integrity than an object with level a_j then $a_i < a_j$. Once an object has been assigned a level, that level is never changed. The primary goal of LOMAC is to prevent data from flowing from low level objects to high level objects. In particular viral data should not be allowed to flow from low level to high level objects. Each subject in the system is also assigned a level. The assignment takes place at creation of the subject, but could be altered at any time. The level selected can be fixed to a_k, or can be selected according to the perceived integrity of the subject. A subject's level may change over time. If a subject at level $a_s(t)$ accesses an object with level $a_o < a_s(t)$ then $a_s(t+1) := a_o$, for a time t. In other words, the subject is demoted to the level of the object it reads from. Subjects may not write to objects with a level higher ($a_s < a_o$) than their own. This is not strictly information flow since the data that is read is not tracked. However, data that is read from a low integrity source cannot be written to a high integrity source. Low integrity data may include data received from the network. LOMAC also prevents subjects of low level from interfering with subjects of higher level, and only allows subjects of sufficient level to access certain critical system functions, for example the reboot system call. In LOMAC a subject is a process group. Subjects are prevented from interfering with higher level subjects, by preventing signals from being sent from the lower level subjects to higher level subjects.

5.1. LOMAC in ConSA

To implement LOMAC only the Information Flow Manager need be implemented while using the ConSA implementation of the Linux security system. The following changes are necessary:

- SubjectLabel — The Subject Label needs to store the level of the subject a_s. This parameter will be set by the Subject Management Module.
- EntityLabel — The Entity Label has a new attribute a_o indicating the level of the object to be accessed. `GrantAccess` and `RevokeAccess` may be modified so as to alter the level of the entity associated with the label. Due to the static nature of the assignments, the configuration file technique of LOMAC is preferred.
- Information Flow Manager — Two methods need implementation namely `ActivateFlow` and `ReturnFlow`.
  - `ActivateFlow(EntityLabel` L_e`, SubjectLabel` L_s`,` $Parameters$`)`
    This method will be activated when an object is written to. `ActivateFlow` will return true exclusively if:
    $$L_s.a_s >= L_e.a_o \tag{2}$$
  - `ReturnFlow(EntityLabel` L_e`, SubjectLabel` L_s`,` $ReturnVal$`)`
    This method will be activated when an object is read from. The implementation is trivial:
    $$L_s.a_s := \min(L_s.a_s, L_e.a_o) \tag{3}$$
    and always returns true.

Domain and Type Enforcement (discussed in the next section) implements a similar policy, and can be generally more secure. It also resolves some problems that LOMAC has. Please refer to [16] for more details.

6. DOMAIN AND TYPE ENFORCEMENT (DTE)

Domain and Type Enforcement uses the "least privilege" concept, a subject is granted the least privileges necessary for that subject to complete the authorized task requested of it. To achieve this, mandatory access controls are enforced on all subjects, even the super user. The system is viewed as a collection of active entities (subjects) and passive entities (objects). Now an access control attribute, called the *domain*, is associated with each subject. The domain of a subject does not change. Each object has an attribute associated with it, known as the *type* of the object. A Domain Definition Table (DDT) is maintained that represents the access domains have to types. This table can be viewed as a mapping:

$$DDT(d,t) : D \times T \rightarrow \{true, false\}, d \in D, t \in T \tag{4}$$

where D is the set of domains, and T is the set of types. The mapping normally also represents the type of access, so that the range of the DDT mapping is not a boolean value, but a representation of the allowed access modes such as read or execute. Furthermore there is a table, the Domain Interaction Table (DIT) which lists, for each domain, the access subjects in that domain have to subjects in other domains. Once again this can be represented by a mapping:

$$DIT(d_1, d_2) : D \times D \rightarrow \{true, false\}, d_1, d_2 \in D \quad (5)$$

and usually represents several interactions including signals. A subject is usually a process. The first process started in the system, usually the init process, begins in a specified domain. Applications (executables) are assigned domains according to the roles they must perform. Files and other objects in the system are assigned types. Implicit typing is used, whereby a file inherits the type of the directory in which it is found, unless stated otherwise explicitly. This technique reduces the overhead involved in managing security information in a system with a large number of objects. (ConSA can also implement implicit protection; the Authorization Control may assign one Entity Label to many objects, taking the resource hierarchy into account). Now programs must execute in the correct domain. To allow programs to remain unaltered DTE supports auto domain switching. Programs are started by making use of the `exec()` family of system calls. DTE will determine which domain the program, or entry point, must execute in and will change the domain of that process. DTE also supports new system calls that may be used to query various DTE attributes and may request domain changes. DTE is implemented over and above conventional UNIX security. One of the strengths of the DTE system is the high level configuration files that are used to configure DTE [7]. The reader may refer to [6, 7, 8, 9, 10] for more details of DTE and applications of the system.

6.1. DTE in ConSA

The implementation of DTE in the ConSA system is very similar to that of the LOMAC implementation. New entities may be created to access special DTE services, and possibly for use of the `exec` call. The `exec` call can be easily protected by the ConSA system. The changes to the Linux implementation are as follows:

- SubjectLabel — The Subject Label needs to store the domain of the subject s_d.
- EntityLabel — The Entity Label has a new attribute e_t indicating the type of the object. If the entity refers to a signal or other domain interaction, the attribute s_d will refer to the domain of the target subject. As in LOMAC, `GrantAccess` and `RevokeAccess` and associated methods may be used to modify the DDT and DIT.

- Information Flow Manager — Two methods need implementation namely `ActivateFlow` and `ReturnFlow`.

  - `ActivateFlow(EntityLabel` L_e`, SubjectLabel` L_s`,` *Parameters*`)`
    This method will be activated when an object is written to. The mode of access selected is important; however illustration with the boolean mapping of the DDT and DIT is sufficient to demonstrate the implementation. `ActivateFlow` returns true if

    $$DDT(L_s.s_d, L_e.e_t) = true \tag{6}$$

    for entities that refer to objects and

    $$DIT(L_s.s_d, L_e.s_d) = true \tag{7}$$

    for entities that refer to domains

  - `ReturnFlow(EntityLabel` L_e`, SubjectLabel` L_s`,` *ReturnVal*`)`
    This method will be activated when an object is read from. No distinction is made between the flow types, so the mode is used to determine access types. `ReturnFlow` can therefore be implemented exactly as `ActivateFlow`.

LOMAC and DTE are very simple policies that are extremely configurable and can assist in the security of a system considerably. In particular, they help to control the extent to which a system is system is compromised if compromise occurs as a result of application error.

7. RULE SET BASED ACCESS CONTROL (RSBAC)

Rule Set Based Access Control is an implementation of the Generalized Framework for Access Control [11]. The Generalized Framework for Access Control divides access decisions into four components:

- Access Control Information (ACI)
  The Access Control Information are the security attributes of subjects and objects within the system. This may include, the sensitivity level of files, or the level of trust of a subject.

- Access Control Context (ACC)
  Context is information about the system unrelated to the objects and subjects. For example time, or which subjects are the members of a certain group.

- Access Control Rules (ACR)
  These are the rules that determine if access is granted or not, and define the security policy that is implemented.

- Authorization (ACA)
  The authorization component implements the physical access control mechanism.

RSBAC has two main components to enforce access control:

- Access Enforcement Facility (AEF)
  This component enforces access control on all attempts to access an object. This is implemented by intercepting system calls (by modifying Linux kernel source). This component will then make use of the next component to implement the security model.

- Access Decision Facility (ADF)
  The Access Decision Component implements the decision making process, by consulting the ACC, ACI and ACR. The component consults the rule base, and relays the necessary subject and object attributes as well as the context. This may result in a change of attributes and/or context.

Several models have been implemented, and may work in conjunction, using a *logical and* function to determine the decision. These models include:

- Bell-LaPadula Mandatory Access Control
  The traditional Mandatory Access Control model.

- Functional Control
  A simple role based model, restricting access to security information to security officers, and access to system information to system administrators.

- Role Compatibility
  This model is similar to the DTE model, but it is implemented at subject level. So a subject has a role, and may access certain objects according to that role. The role does not change if a new process is started.

Please Consult [11] for further details.

7.1. RSBAC and ConSA

RSBAC and ConSA have similar goals, to provide a flexible system that may be used to implement several security policies. ConSA may be implemented in RSBAC with suitable access control information, by implementing a component that consults the Information Flow Manager and Entity Label to

determine if access is granted. For ConSA to implement RSBAC, the necessary security attributes must be stored in the Subject Label and Entity Label. The Information Flow Manager should then have enough information to invoke the ADF to implement the access control decision. The AEF component is already implemented by intercepting system calls. RSBAC has the following deficiencies:

- Although RSBAC is a modular security system, the current implementation needs direct modification to support new security models. There is no interface for components to be automatically inserted into the system.
- RSBAC can implement many access control mechanisms, but the framework does not assist in the construction of a security model. The ADF component simply queries the supplied modules, and bases the decision on these components.
- Security Attributes are stored in a common ACI structure, this structure does not have support for arbitrary security attributes. The attributes are selected according to the implemented security models. Subject Labels and Entity Labels perform the same role in ConSA and are determined by the security system implementor. A generic ACI system is available through the modular implementation of these labels.

8. CONCLUSION

This paper has demonstrated theoretically that ConSA can implement a wide variety of existing security models, and that the architecture supports the implementation of security models.

References

[1] M. S. Olivier, *Towards a Configurable Security Architecture*, Data & Knowledge Engineering, To appear.

[2] D. E. Bell and L. J. LaPadula, "Secure computer system: unified exposition and Multics interpretation", *Rep. ESD-TR-75-306*, March 1976, MITRE Corporation

[3] *The Linux-PAM System Administrators' Guide*, Andrew G. Morgan, 1998

[4] *Inside Unix*, Chris Hare, Emmett Dunlaney, George Eckel, Steven Lee, Lee Ray, New Riders Publishing, 1994

[5] *The Linux Kernel book*, Remy Card, Eric Dumas, Frank Mevel, Wiley, 1997

[6] *Confining Root Programs with Domain and Type Enforcement*, USENIX UNIX Security Symposium, 1996

[7] *Practical Domain and Type Enforcement for UNIX*, L. Badger, D. F. Sterne, D. L. Sherman, K. M. Walker, S. A. Haghighat, IEEE Symposium on Security and Privacy, 1995

[8] *A Domain and Type Enforcement UNIX Prototype*, IEEE Symposium on Security and Privacy, 5th USENIX UNIX Security Symposium

[9] *Controlling Network Communication with Domain and Type Enforcement*, L. Badger, D. F. Sterne, D. L. Sherman, K. M. Walker, S. A. Haghighat, S. L. Murphy, Proceedings of the 1995 National Information Systems Security conference.

[10] *The Controlled Application Set Paradigm for Trusted Systems*, D. F. Sterne, Glenn S. Benson, Proceedings of the 1995 National Information Systems Security conference.

[11] *Rule Set Based Access Control as proposed in the 'Generalized Framework for Access Control'*, Amon Ott, Masters Thesis, 1997

[12] *From a Formal Privacy Model to its Implementation*, Simone Fischer-Hübner, Amon Ott, National Information Systems Security Conference, 1998

[13] *Design Specification: An Implementation of Access Control Lists for Linux*, http://students.dwc.edu/frival/acl/acldesign.html

[14] *The Linux Trustees Project*, http://www.braysystems.com/linux/trustees.html

[15] *Group ACL for ext2 in LiVE*, http://aerobee.informatik.uni-bremen.de/acl_eng.html

[16] *LOMAC - Low Water-Mark Mandatory Access Control User's Manual v0.2*, Tim Fraser, NAI Labs, 1999

[17] *The Single UNIXR Specification, Version 2*, The Open Group, 1997, *www.opengroup.org*