Evaluation of Linux Security Frameworks

Erik Karlsson

April 8, 2010
Master's Thesis in Computing Science, 30 ECTS
Supervisor: Ola Ågren
Examiner: Fredrik Georgsson

Abstract

The number of threats to computers attached to networks continually increases. The focus of preventing security exploits has been on the network, while local exploits has been mostly overlooked. Many security problems in Unix systems stem from the way security is managed; by delegating all security decisions to object owners. There are a number of security frameworks which aim to remedy this in Linux by restricting access to kernel objects, such as files. Ericsson is interested in finding the best possible security frameworks for use with their Linux products.

In this thesis, the available security frameworks are evaluated based on criteria given by Ericsson. First, the theoretical foundation of computer security is explored to serve for an overview of the security frameworks and their properties. Then specific attributes are refined and their values gathered from each framework. These attributes then serve as a basis for selecting two frameworks for further testing.

The selected frameworks are SELinux and AppArmor, based on commerical support, ease of integration, and overall protection measures. Tables with the collected attributes are presented for comparison.

The frameworks TOMOYO and SMACK should have been given more consideration. AppArmor is not useful for the server-centric environment used at Ericsson.

Acronyms

ACI Access Control Information

ACL Access Control List

ACM Access Control Matrix

ACR Access Control Rules

ADF Access Decision Facility

AEF Access Enforcement Facility

API Application Programming Interface

AVC Access Vector Cache

CLI Command Line Interface

COCOMO Constructive Cost Model

CUI Character User Interface

DAC Discretionary Access Control

DARPA Defense Advanced Research Projects Agency

DDT Domain Definition Table

DIT Domain Interaction Table

DSI distributed security infrastructure

DTE Domain and Type Enforcement

DTOS Distributed Trusted Operating System

DoS Denial of Service

EVM Extended Verification Module

FD file descriptor

FLASK Flux Advanced Security Kernel

FSM Finite State Machine

GFAC Generalized Framework for Access Control

GUI Graphical User Interface

HTML Hyper Text Markup Language

IAC Integrity Access Class

IDE Integrated Development Environment

IEC International Electrotechnical Commission

IPC Inter-Process Communication

ISO International Organization for Standardization

KVM Kernel-based Virtual Machine

LIDS Linux Intrusion Detection System

LKM Loadable Kernel Module

LSM Linux Security Modules

MAC Mandatory Access Control

MLS Multilevel Security

N/A not available

NFS Network File System

NSA National Security Agency

POSIX Portable Operating System Interface for uniX

RBAC Role-Based Access Control

RSBAC Rule Set Based Access Control

SAC Security Access Class

SCC Secure Computing Corporation

SELinux Security-Enhanced Linux

SID security identifier

SLES SuSE Linux Enterprise Server

SLIM Simple Linux Integrity Module

SMACK Simplified MAC Kernel

SMP Symmetric Multi-Processing

SPDL Simplified Policy Description Language

SSID source SID

SuSE Software und System-Entwicklung

TCPA Trusted Computing Platform Alliance

TCSEC Trusted Computer System Evaluation Criteria

TE Type Enforcement

TOE Target of Evaluation

TOMOYO Task Oriented Management Obviates Your Onus

TPE Trusted Path Execution

TPM Trusted Platform Module

TSID target SID

U.S. United States

UID user identifier

XML Extensible Markup Language

YAML Yet Another Markup Language

Contents

Chapter 1

Introduction

As the number of security threats in the networked world increase, securing the connected network nodes has become increasingly important. While external threats can be mitigated by firewalls and network infrastructure, there has been an increase in local exploits for which there are few preemptive defenses. This is currently an active field of research [42]. The vulnerability to local exploits and rootkits can be partially ascribed to lack of focus on system security in modern operating systems [25].

The traditional security mechanism in Unix, Discretionary Access Control (DAC), is both limited in scope and, as the name implies, relies on user discretion to impose secure access restrictions. DAC only covers filesystem access and does not impose security restraints for other objects in the system, such as IPC or network sockets.

In order to guarantee the enforcement of a system-wide policy[1], an alternate mechanism, Mandatory Access Control (MAC), can be utilized. MAC does not neccessarily extend the security domain of DAC, but allows system administrators to impose rules which cannot be circumvented by users. In Linux, there is a module framework in place to enable the creation of different implementations of MAC, called Linux Security Modules (LSM), but not all available MAC implementations use this.

By increasing security measures on interactions within the system, the impact of a security breach can be lessened. Although, employing a MAC framework is not enough to guarantee higher security; a proper security policy is also required.

1.1 Problem Statement

The product targeted in this thesis work is developed by Linux Development Center at Ericsson and is based on SuSE Linux Enterprise Server (SLES) 10 which already ships and supports the MAC security framework, henceforth abbreviated to security framework, *AppArmor*. However, there are a number of other security frameworks, with varying security agendas, which could potentially be more suitable alternatives. These frameworks have seen some comprehensive comparisons, most of which focuses on a pair of or a few frameworks under varying conditions [4, 7, 8, 19, 36]. To complicate matters, the frameworks does not present easily comparable features because of their differing security goals.

By comparing these frameworks closer, Ericsson can select one for further testing with its Linux products. This will serve to increase the product availability and information security for network

[1] See Chapter 2 on page 3

nodes in exposed operating environments.

The specific security requirements of the target product, from now on called Target of Evaluation (TOE) as specified in the Common Criteria[2], are not provided with the thesis application and will have to be evaluated as part of this work.

1.2 Document Layout

The remainder of this thesis will be structured as follows: In Chapter 2 we are introduced to the basic concepts of computer security required to describe the security frameworks in Chapter 3. Chapter 4 outlines the intended process of evaluation and Chapter 5 the results of the thesis work. The results are discussed in Chapter 6.

[2]The Common Criteria is the international standard ISO/IEC 15408

Chapter 2

Computer Security

Computer security is a form of information security applied to *computers* and *networks*. Information security is defined as the protection of information from "unauthorized access, use, disclosure, disruption, modification or destruction" by the U.S. House of Representatives [40]. While this means that the scope of computer security can be quite wide, this thesis will mostly address the *access* part of computer security in a computer context.

The security of a computer system is dictated by a *security policy*. A security policy can be defined as a statement, or series of statements, that partitions a system into *secure* and *nonsecure* states [3]. Formally, this means that a secure system is one that, if started in a secure state, only ever transitions to secure states. In reality, policies are rarely this formal and consists of documents in natural language detailing the rules of a system. A computer system may be governed by several security policies with distinct security considerations.

While a security policy might detail *what*, and from *whom*, information should be protected [38], it should not say anything about *how* the policy should be enforced. The enforcement is the task of the security mechanisms of a system, which are more specific implementations of security concepts, which can be said to provide the transitions from one security state to another.

The distinctions made about security in this chapter are not always as clear-cut in real applications and organizations. Likewise, the descriptions are not exhaustive, but selected to illustrate computer security in a structured way.

2.1 Security Models

How a policy should be structured and enforced can be dictated by a *security model*. To enforce its policies, security models specify the use of security mechanisms which individually might not secure a system, but can be effective against specific threats.

Models usually follow one or a combination of *principles* or is backed by a theoretical framework for describing security properties. The most common of these principles are sometimes called the CIA triad, composed from the initials of *confidentiality*, *integrity* and *availability*. Confidentiality means that information will only be divulged to subjects with proper authorization, or clearance. Integrity means that information will not be modified or destroyed without permission, and that non-repudiation and authenticity of said information is ensured. Availability means that information must be available in a timely fashion when requested. Non-repudiation, mentioned as a sub-principle of integrity, means that a party partaking in a transaction should not be able to deny their involvement.

It is generally not possible to satisfy all these principles at once, which is why models often focus on one, or a combination of them.

2.1.1 Access Control Matrix

The Access Control Matrix (ACM) was introduced by Butler W. Lampson of the Xerox Corporation as an attempt of unifying similar security concepts in use at the time [21]. As a generic abstract security model, the ACM can be, and is, used as a formal base for defining policies in other models [22].

Somewhat formally, the model uses three integral sets to describe access rights in a system. Objects, O, are the entities to which access is *restricted*, such as processes or files. Subjects, S, are the entities which *seek* access to objects, such as users or other processes. Rights, R, are the types of accesses available in the system, which can be assigned to pairs of objects and subjects.

These sets are used in defining the Access Control Matrix (ACM), A, with one row per subject and one column per object. The entries contain rights from R governing access between subject and object as follows

$$A(s, o) \subseteq R$$

where $s \in S$ and $o \in O$. The set of secure states in this model is denoted as (S, O, A).

Since a real computer system contains many subjects and objects, most which are not allowed any access to each other, an implementation of the ACM would result in a sparse matrix. In order to alleviate this potential performance hog, implementations tend to store the access rights either column wise or row wise. These modes of storage are called capability list and Access Control List (ACL). Capability lists stores the rights with the subjects, while ACL stores them with the objects.

Table 2.1 shows an example of an access control matrix, where the element (i, j) represents the rights $a(i, j)$ for subject i on object j.

	Object 1	**Object 2**
Subject 1	read, write, execute, own	read
Subject 2	write	read, write, execute, own

Table 2.1: Example of an access control matrix with two subjects and two objects

2.1.2 Bell-La Padula

The Bell-La Padula model was developed David Elliot Bell and Leonard J. La Padula at the MITRE Corporation, and concurrently by Walter et al. at Case Western Reserve University, but only the former has received widespread recognition due to their similarity [22]. The model was conceived as a specialization of the ACM able to incorporate military security policy.

Military security policy is based on the concept of *confidentiality*, where information leakage might damage national security. Since all information is not equally sensitive a distinction in form of different *sensitivity levels* can be made. These are, in order of potiential harm if leaked, `UNCLASSIFIED`, `CONFIDENTIAL`, `SECRET` and `TOP SECRET`. To further limit the number of people with access, information can be divided into *compartments* to which access is granted on a need-to-know basis. Objects are given classifications, or *security levels*, which consists of a sensitivity level and a set of compartments. Similarly, subjects are given *clearance* to access both sensitivity levels and compartments.

The model is defined formally as a Finite State Machine (FSM), but can be described informally by the following properties

1. the *simple security property* says that a subject only has read access to objects of lower or equal classification than the subject's clearance

2. the $*$-*property* says that a subject only has write access to objects of higher or equal classification than the subject's clearance

Originally, the $*$-property was also associated with append operations and included the simple security property. However, in modern use of the model the above variant is the most widely used. The properties are often identified by the simplified prohibitions of "reading up" and "writing down", respectively.

Another principle stated in the model is called the *tranquility principle*, which states that the classification of a subject or object may not change while it is active or in use. This is introduced to protect against information leaks where for example a subject with high clearance reads a classified document, downgrades its own clearance and writes to a document of lower classification than the first. This would in effect be a "write down" and violate the $*$-property.

2.1.3 GFAC

Generalized Framework for Access Control (GFAC) was described by Leonard J. La Padula and Marshall D. Abrams as a model which separates the task of enforcing a policy and the definition of it, to allow different policies to coexist in a single system [20].

The premise of GFAC is that access control ultimately consists of a small set of fundamental concepts. As a result, it describes access control policies as *rules* expressed in terms of *attributes* by *authorities*. Authorities define security policies, identifies security information, and assign values to attributes of controlled resources. Attributes describes properties of subjects and objects. Rules are formalized expressions which defines the security policies of the system.

The model divides the process of access control into two parts; adjudication, which is handled by the Access Decision Facility (ADF), and enforcement, which is handled by the Access Enforcement Facility (AEF). Figure 2.1 illustrates the GFAC access control process. The figure contains numbered labels which are referenced in the following description of the access control process.

When a process, (1), attempts to perform an operation, the AEF queries, (2), the ADF for a policy decision. As a part of the query, the AEF provides the information needed for the ADF to make a decision, denoted Access Control Information (ACI). The ADF then uses this information to look up the relevant access rules, (3), in the Access Control Rules (ACR) database, and returns this, (4), to the AEF. The AEF then enforces the ADF's decisions about which access control requests to deny or allow.

The enforcement facility, AEF, is described as a state machine where each system call is represented by a transition rule, to accurately model the operations of the target system. The rules can also be chosen as generalizations of several operations, using a mapping between system calls and rules.

2.1.4 Domain and Type Enforcement

Domain and Type Enforcement (DTE) was proposed by Lee Badger and Daniel F. Sterne as an alternative to the then pervasive military-style security frameworks with focus on confidentiality [1, 2]. This new work was intended to ease adoption of MAC by providing low system administration overhead, application compatibility, and unobtrusive operation.

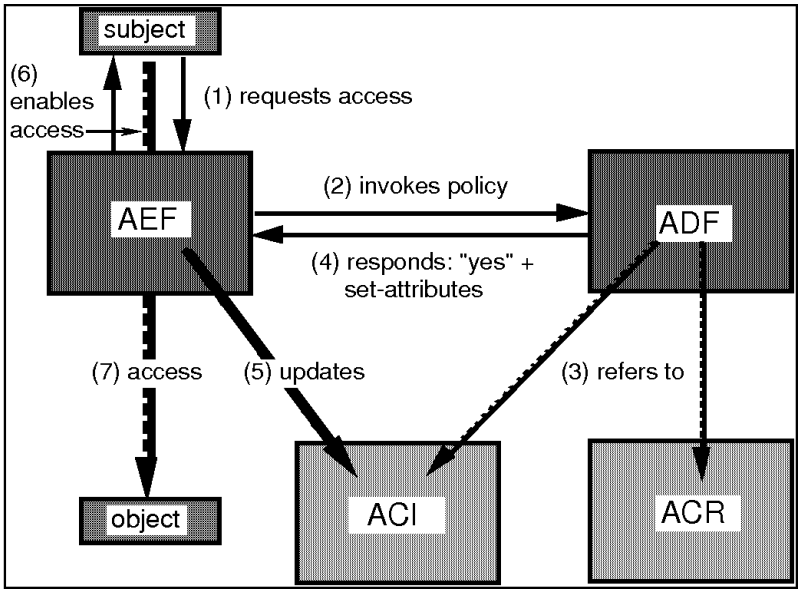

Figure 2.1: Overview of the Generalized Framework for Access Control (GFAC) architecture

To satisfy these requirements, DTE extends on Type Enforcement (TE), a table-oriented MAC mechanism. TE assigns the attribute *domain* to each subject and *type* to each object. The allowed interactions between domains and types are stored in a global table, the Domain Definition Table (DDT), which is an ACM with domains and types in place of subjects and objects. Similarly, interactions between subjects are mediated using the Domain Interaction Table (DIT).

The strictly table-oriented approach in TE has several shortcomings which are addressed in DTE by expressing policy in a high level language and by using *implicit* typing of managed objects.

The Domain and Type Enforcement (DTE) policy language aims to express the information of the DDT and the DIT in a human-friendly form, and adds features such as macros and automatic domain transition rules.

Implicit typing means that the security attributes of subjects and objects are kept in memory instead of being stored on disk. The memory database is created upon system initialization from log files which are updated on file system changes such as file creation or renaming. Attributes are stored sparsely, applying recursively to directory structures where possible.

2.1.5 FLASK

The Flux Advanced Security Kernel (FLASK) model was conceived by Stephen Smalley and others as part of a Defense Advanced Research Projects Agency (DARPA) funded project [34]. It is based on the prototype system Distributed Trusted Operating System (DTOS), which has a similar design, although lacks mechanism flexibility. Superficially, FLASK is consistent with the GFAC model, but additionally attempts to secure non-atomic policy operations.

The main purpose of the FLASK model is to allow for arbitrary policy flexibility to cater to as many security interests as possible. In order to achieve this, the access controls must be fine grained and the propagation of access rights must conform to policy. Additionally, to fully support dynamic policies and policy changes, the model must support revocation of previously granted

access rights. Secondary goals of the model include application transparency, ease of assurance and minimal performance impact.

One way of achieving total policy flexibility is to consider a computer system as a state machine, whose state transitions are atomic operations, and where the policy then can mediate any operation atomically. A real system is too complex to model this way, and in FLASK a compromise is sought by subjecting a security relevant part of the system to such policy control.

Figure 2.2 shows an overview of the architecture of the FLASK model. The architecture is similar to that of GFAC, where the policy enforcement is divided into two parts – *object managers* which enforce security decisions made by *security servers* based on the security policy. The object managers implement a control policy which allows the security policy to govern the object managers' subjugated objects, by defining the actions taken for security decisions made by security servers. They are also required to define a mechanism for assigning labels to their objects.

The labels, or *security contexts*, are strings which can be interpreted by applications or users, but are opaque to object managers. Security contexts are passed onto the security servers, which maps them to security identifiers (SIDs). A SID is a unique identifier used to reference a specific security context, within the security server, where it represents a policy permission statement.

An *access vector* is a set of permissions returned by a security server in return of a access request. Access requests are made using the SIDs of the requesting subject and the target object as parameters, denoted source SID (SSID) and target SID (TSID) respectively. Since access requests are often made multiple times for the same object, the model defines an Access Vector Cache (AVC) which allows object managers to cache access vectors made by the security servers.

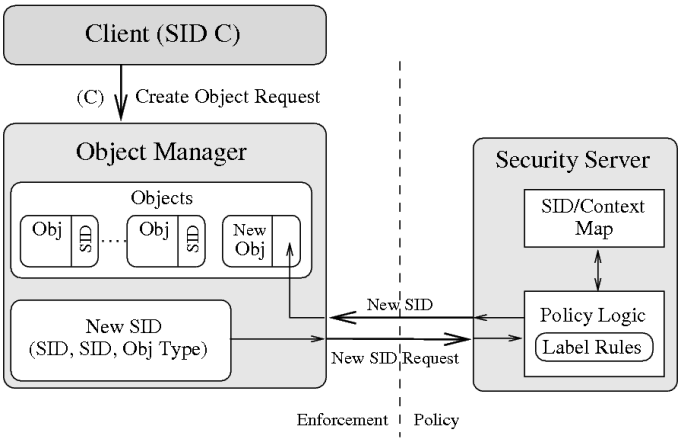

Figure 2.2: Overview of the Flux Advanced Security Kernel (FLASK) architecture

2.1.6 DSI

The distributed security infrastructure (DSI) model was developed by Ericsson Research in Canada for use in Linux clusters [30, 45]. It is intended to extend the scope of MAC from individual nodes to an entire cluster, enforcing a common, distributed, security policy.

The model is divided into *managment* and *service* components. Managment components includes a security server, security managers, and a security communication channel. The security server is responsible for the distributed security policy, the security managers enforce the security

policy at each node, and the security communication channel provides encrypted and authenticated communcations between the managment components.

The service components provide security mechanisms for the security managers and consists of two types; security services and security service providers. The latter which provides services for the former, such as secure key managment.

For access control, the DSI model divides access types into four categories; local, remote, outside, and no cluster access modes. Only the first two are considered by the model. During local accesses, in which the subject and object are on the same node, access permissions are based on the subject's SSID and the target's TSID, much like in the FLASK model. For remote accesses, where the subject and object are on different nodes in the same cluster, the access queries are extended to include node identifiers for both the subject and the object.

2.2 Security Mechanisms

Security mechanisms, or protection mechanisms, are implementations of security concepts which protect a subset of a system. This concept is quite wide, as one may choose to encompass an entire system implementation. Mechanisms are usually considered to lie in one of the following security domains; authentication, authorization, auditing, or encryption. Authentication is the process of verifying the identity of a user to a system. Authorization is the act of determining which resources of a system to make available to a, usually authenticated, user. Auditing is the gathering of security information, such as access violations, for use in security assessments. Encryption, or cryptography, is the practice of hiding information, usually in plain sight, by scrambling it using a reversible mathematical process.

2.2.1 Discretionary Access Control

Discretionary Access Control (DAC) is defined in the Trusted Computer System Evaluation Criteria (TCSEC) [39] to control access between users and objects, and to allow users to specify and control sharing of those objects to other users or groups of users. Many implementations of DAC further restricts this definition by requiring that objects have an *owner*, which has primary control over the owned objects' permissions. It is common to make use of ACLs for specifying the policy used in systems employing DAC.

2.2.2 Mandatory Access Control

As with DAC,Mandatory Access Control (MAC) is also defined in the TCSEC. However, this definition is mainly concerned with Multilevel Security (MLS), primarily used in military systems, which makes it too narrow for general use.

Generally, MAC can be said to enforce policy decisions on access control made by a central authority, for example a system administrator, as opposed to the owner of an object in DAC. The enforcement must be performed reliably, for instance by the operating system's kernel. This ensures that users cannot override the policy.

2.2.3 Role-Based Access Control

Role-Based Access Control (RBAC) enforces access decisions based on the roles of the subjects [38]. Subjects are authorized to assume roles, which represents a set of permissions. By assuming a role, the subject will also gain use of the permissions of that role.

Roles are usually used as a way of describing *responsibility*, where multiple subjects can share the same role, or responsibility. This enables roles to be updated without updating the permissions of individual users. Role hierarchies can be created to more closely reflect the natural structure of an organisation, than other access control mechanisms.

2.2.4 Reference Monitor

The reference monitor describes the necessary and sufficient properties for achieving secure access controls in a system [15]. It is an abstract model which does not require a specific implementation or policy; any specific implementation will enforce a specific policy set.

The model describes three properties of a reference monitor; it must be invoked for *every* access request, be *tamperproof*, and be *small* enough to be tested for completeness. The first property makes sure that the security policy is always invoked when a policy decision is needed, as it would otherwise be possible to circumvent the policy. The mechanism must be tamperproof to be safe from attacks directly against the mechanism, causing a Denial of Service (DoS) for example. Lastly, the mechanism must be verifiably secure, as it could not otherwise guarantee enforcement of the policy.

Reference monitors are often used as the most basic entities in secure systems, and are required to implement some security models, such as FLASK.

2.3 Other Concepts

This section briefly touches on subjects which deal with questions on how to achieve security in real-world systems.

2.3.1 Separation of Mechanism and Policy

This concept is also a design principle, closely related to object abstraction, which states that mechanisms for authentication or resource allocation should not overly restrict which policies it might enforce [24]. In practice, this means that it is desirable to make mechanisms as reusable as possible.

Chapter 3

Linux Security

Security in the Linux Kernel has traditionally consisted of file system security, or Discretionary Access Control (DAC), and network security, handled by netfilter in modern Linux distributions. netfilter is a framework for manipulating network packets in the Linux network stack and is commonly used for firewalling purposes. There are other tools for network security as well, such as OpenSWAN for IPSec. These, however, are beyond the scope of this thesis.

3.1 Linux Security Modules

The Linux Kernel did not previously allow Loadable Kernel Modules (LKMs) to mediate access to kernel objects outside of their scope. Thus, several previous attempts at creating security modules has used *system call interposition* [9,18] which is not suitable for access control, because of the ease with which it can be circumvented [43]. These wrappers may also require patches to the kernel or may even reimplement certain kernel functionality due to the limited scope of LKMs.

During the Linux Kernel Summit in 2001, the National Security Agency (NSA) presented their new security framework Security-Enhanced Linux (SELinux) [44]. They argued for the need of a flexible access control architecture in the Linux kernel, but Linus Torvalds, the kernel maintainer, took heed by instead suggesting a new infrastructure for implementing security using LKMs.

Work was set out to create a lightweight access control framework which would allow a multitude of security models to be implemented as LKMs, resulting in the Linux Security Modules (LSM) framework. The design goal of LSM was to allow modules to answer the question "May a subject `S` perform kernel operation `OP` on the internal kernel object `OBJ`?" Additionally, the design had to be generic, simple, minimally invasive, efficient, and be able to support the existing POSIX.1e[1] capabilities logic.

LSM places hooks in the kernel right before accesses to internal objects are granted, where these accesses may be rescinded by the policy implemented in an LSM module. Because of the simplicity design constraint, LSM hooks are mainly *restrictive*, in that they are only used when the kernel is about to grant access, not when about to deny access. The complete solution, *authoritative* hooks, would have been significantly more complex. A few permissive hooks exist, to allow the implementation of POSIX.1e capabilities as a LSM module.

In order to allow modules to keep track of special security properties of objects, LSM attaches *security fields* to mediated objects. The modules are responsible for managing the data in these fields themselves.

[1] See Section 3.2.1

Extensive automatic testing has been performed on the soundness of the LSM implementation, regarding the placement of hooks [6, 10, 16, 46]. While most studies found exploitable bugs in the placement of hooks in early implementations, they also agreed with the already manually chosen hook placements.

3.2 Available Security Frameworks

Following the inclusion of LSM into the Linux Kernel, a number of projects has developed security modules. There are also frameworks which does not employ LSM, because of technical reasons.

The examples of policy language presented with some of the following frameworks are intended to highlight the configuration syntax complexity and readability.

3.2.1 POSIX.1e Capabilities

The now withdrawn POSIX draft 1003.1e attempted to standardize Unix security measures by introducing them in the Portable Operating System Interface for uniX (POSIX) standard [14]. Linux includes an implementation of a part of this draft, the *capabilities* framework.

The capabilities framework in Linux work by partitioning the all-encompassing root privilege into separate privileges, called capabilities [37]. These capabilities can be assigned to processes, which allows these processes to run under other user identifiers (UIDs) than root, but with some of its privileges.

In a capability-enabled system, each process has three sets of bitmaps describing which capabilites are available to it: inheritable (I), permitted (P), and effective (E). The effective set contains the capabilities a process can use, the permitted set contains the capabilites a process is permitted to put in its effective set, and the inheritable set contains the capabilites which are allowed to be inherited by a program executed by the process.

3.2.2 AppArmor

AppArmor, Application Armor, is an application-centric MAC framework built using LSM. It was first developed by the company Immunix, which used it in its GNU/Linux offering between 1998 and 2003. Immunix was acquired by Novell in 2005 and they maintained AppArmor until 2007, when the AppArmor team was laid off.

The AppArmor security framework has the following design goals: It needs to be *secure* to keep applications in confinement, *fast* to avoid overhead, and *transparent* to allow normal operation of software. Security is achieved by using LSM which is more secure than system call interposition, as described in Section 3.1. Speed is achieved by a simple security model with fast lookup which does not need caching. Transparency is achieved by reusing Unix security semantics on applications instead of users.

AppArmor policies, called *profiles*, consists of sets of POSIX.1e capabilities and file permissions attached to programs. The matching of rules to programs is done via path lookup, which has spawned discussion with primarily SELinux proponents on the relative security merits of this method and other design aspects of AppArmor [23].

Listing 3.1 shows an example of an AppArmor profile with use of capabilities, files and network controls.

Profiles can be automatically generated by first running AppArmor in a so called “learning mode”, where all rule violations are logged, and then analysing these logs interactively. This method can also be used to incrementally improve a profile.

Listing 3.1: Example of an AppArmor profile for `/bin/ping`

```
#include <tunables/global>
/bin/ping {
        #include <abstractions/base>
        #include <abstractions/consoles>
        #include <abstractions/nameservice>

        capability net_raw,
        capability setuid,
        network inet raw,

        /bin/ping mixr,
        /etc/modules.conf r,
}
```

3.2.3 DISEC

DISEC is the collective name for a set of utilities implementing the DSI security model and was developed by Ericsson Research in Canada between 2002 and 2004 [29, 45]. It is now defunct.

The access control enforcement of DSI is implemented using LSM, and the distributed security policy is specified using XML.

Although the model does not specify the mediated object types, DISEC implements access control with process level granularity. This means that it only concerns itself with the spawning of processes, the execution of programs, and the communication between processes on the cluster network.

3.2.4 GRsecurity

GRSecurity is a patch-set for the Linux Kernel which emphasises the security-in-depth principle and enforces MAC through RBAC. It was started by Brad Spengler in 2001 as a port of the Openwall kernel patch from Linux 2.2 to 2.4 and has since become a project of its own. The memory protection patch PaX is included in the GRsecurity patch-set, to allow for broader security.

GRsecurity is implemented using custom hooks in targeted system calls and kernel functions, as opposed to system call interposition or the use of LSM [35]. Among the reasons for not using LSM are that GRsecurity requires more functionality than available through LSM, and that LSM could further enable root-kits rather than hamper them.

The goals of GRsecurity are

- simplicity via configuration free operation
- protection against address space modification bugs
- feature-rich ACL and Auditing
- support for multiple processor architectures

The configuration of GRsecurity is managed with ACLs and the userspace tool `gradm`. The ACLs consists of roles and their subjects, which in turn contains file, capability, socket and computational resource controls. Roles can define transitions into other roles and subject properties can be inherited from parents in the filesystem.

Listing 3.2 shows an example of an ACLs with a role and a subject with file and socket controls for the `sshd` Unix daemon.

Listing 3.2: Example of a GRsecurity ACL

```
role sshd u
   subject /
           /               h
           /var/run/sshd r
           -CAP_ALL
           bind disabled
           connect disabled
```

The `gradm` tool can be used to automatically create ACLs for either the entire system, a specific subject, or a specific role. This learning mode will attempt to create a minimal ACL for the task presented and can take hints on how expressively to generate rules for certain subjects.

3.2.5 LIDS

The Linux Intrusion Detection System (LIDS) is, contrary to its name, a MAC framework but also includes a port scan detector and process protection facilities. The patch-set has been maintained since version 2.2 of the Linux Kernel by Xie Huagang and Philippe Biondi. It has since been ported to the 2.6-series, and is now using LSM.

Mediated objects are files, sockets, directories and POSIX.1e capabilities. Configuration of access rights is handled by the command-line program `lidsconf`. The configuration is stored in text files and can be edited by hand, but the command-line tool is the preferred way of changing the security policy. Listing 3.3 shows an example of the usage of `lidsconf`, which will generate a configuration based on the current filesystem information. This means that the configuration command will have to be re-run when, for instance, a file is overwritten and its inode[2] number changes.

Listing 3.3: Example of a LIDS ruleset

```
/sbin/lidsconf -A -R -o /etc/log -j APPEND
/sbin/lidsconf -A -o /var/log/wtmp -j WRITE
/sbin/lidsconf -A -s /bin/login -o /etc/shadow -j READONLY
/sbin/lidsconf -A -s /bin/su    -o /etc/shadow -j READONLY
/sbin/lidsconf -A -s /bin/login -o CAP_SETUID -j GRANT
```

3.2.6 RSBAC

Rule Set Based Access Control (RSBAC) is a security framework for the Linux Kernel developed by Amon Ott, with similar goals as LSM, but with different design and level of abstraction [17,28]. It is based on the GFAC model, which is described in Section 2.1.3 on page 5.

The framework uses system call interception as its system call mediation method. It avoids LSM because of, for example, limited scope of mediation, lack of concurrent capability support, and stateless calls [27].

In RSBAC, the GFAC model has been modified to include a specific metapolicy for the ADF policy decision process, breaking up the ACI into loadable modules, and further abstract the AEF by not allowing it to update ACI after a successful access control request. The metapolicy is the boolean `AND` operator, with concessions for the different return types of the policy modules. The ACI policy modules have four different return values: *granted*, *not granted*, *don't care*, and *undefined*. Table 3.1 illustrates the results of the metapolicy on two operands.

[2] inode is a filesystem structure containing information about a file

Result 1	Result 2	Final Result
Granted	Granted	Granted
Granted	Don't care	Granted
Not granted	Granted	Not granted
Not granted	Don't care	Not granted
Undefined	*	Undefined

Table 3.1: Policy result combination in RSBAC's ADF

The policy configuration is different for each ACI policy module, and will not be outlined here. The currently available modules are shown in Table 3.2.

Name	Code name	Description
Authenticated User	AUTH	Advanced user authentication
Role Compatibility	RC	Role-Based Access Control
Jail	JAIL	Encapsulation of processes
Linux Capabilities	CAP	Manages Linux capabilities
Pageexec	PAX	Prevents against unwanted code execution
Dazuko	DAZ	On-access anti-virus scanner
File Flags	FF	Per file access control flags
Linux Resources	RES	Manages Linux resources
User Management	UM	In-kernel user management
Access Control Lists	ACL	Extensive Access Control Lists
Privacy Model	PM	Controls data privacy
Mandatory Access Control	MAC	Bell-La Padula with modifications

Table 3.2: Available RSBAC ACI modules

3.2.7 SELinux

Security-Enhanced Linux (SELinux) is an implementation of the FLASK security model in the Linux kernel, initially developed by the NSA and the Secure Computing Corporation (SCC) [26].

The first prototype was built using custom hooks in Linux 2.4, but was the trigger for the creation of the LSM framework. It was the first implementation of LSM to be accepted into the Linux kernel, aside from the POSIX.1e capabilities module. File security contexts are stored in Linux' filesystem extended attributes, for performance reasons.

Like its mother project, FLASK, SELinux's main priority is policy flexibility. As an example, the security server implements a security policy which is a combination of TE, RBAC and MLS. The security context is defined as a string containing a user identity, a role, and optionally a MLS level. Roles are only used for processes, so the role *object_r* is used for all file security contexts.

The policy configuration language is based on TE and contains definitions of domains, types, roles, and their respective interactions. These include domain and role transitions, which specifies which changes in security context may occur at and during program execution. Listing 3.4 shows a snippet of a policy configuration for the `dhcpd` daemon, where the `dhcpd_t` type, or domain, is given access to network domains.

Policy configurations can become quite complex, and a reference policy containing rules for

Listing 3.4: Snippet of a SELinux policy

```
allow dhcpd_t netif_type:netif { tcp_send udp_send rawip_send };
allow dhcpd_t node_type:node { tcp_recv udp_recv rawip_recv };
```

many common services and applications is developed and distributed to ease the use and adoption of SELinux.

3.2.8 SMACK

Simplified MAC Kernel (SMACK) is a MAC framework written for the Linux kernel by Casey Schaufler [33]. It is implemented using LSM and is included in the kernel since 2.6.24.

The security model in SMACK is a mix of DTE and Bell-La Padula. The security attributes of objects are case sensitive strings between 1 and 23 characters in length, and are called *labels*. There are four predefined labels listed in Table 3.3. Subjects, or processes, are called *tasks* and system tasks are given the floor label. Access permissions are enforced by the following rules, in order:

1. Any access requested by a task labeled "*" is *denied*
2. A read or execute requested by a task labeled "^" is *permitted*
3. A read or execute requested on an object labeled "_" is *permitted*
4. Any access requested on an object labeled "*" is *permitted*
5. Any access requested by a task on an object with the *same* label is *permitted*
6. Any access requested that is explicitly defined in the loaded rule set is *permitted*
7. Any other access is *denied*

Label	Pronunciation
_	Floor
^	Hat
*	Star
?	Huh

Table 3.3: Predefined SMACK labels

The policy configuration language consists of one statement:

```
subject-label object-label access-mode
```

Permissions for labels are given using the traditional access modes of Unix – read, write, and execute – with the addition of *append* for some object types.

3.2.9 TCPA

The TCPA [32] framework is not technically a MAC framework, but includes MAC functionality and is therefore included in this thesis. It was developed by IBM but has been abandoned.

The framework consists of the modules Extended Verification Module (EVM) and Simple Linux Integrity Module (SLIM), and provides a chain of trust for applications with the goal of minimizing

security impacts of application vulnerabilities. The modules are implemented using LSM and are a rare example of LSM stacking.

The EVM module is used for validating executables before execution, using a Trusted Platform Module (TPM). The cryptographic verification key is stored in the filesystem's extended attributes, much like SELinux.

The SLIM module is used to constrain privileges for untrusted applications, via MAC. It uses a combination of the Caernarvon and the Low Water-mark models for managing information integrity. All files are given two labels – an Integrity Access Class (IAC) label which contains the classes `SYSTEM`, `USER`, `UNTRUSTED`, and `EXEMPT`; and a Security Access Class (SAC) label which contains the classes `SYSTEM`, `USER`, `PUBLIC`, and `EXEMPT`. Process are given the labels IRAC, IWXAC, SWAC, and SRXAC, where the letter R stands for read, W for write and X for execute.

SLIM defines the following access control rules, based on the previously given classes:

Read:			
	IRAC(process)	<=	IAC(object)
	SRXAC(process)	>=	SAC(object)
Write:			
	IWXAC(process)	>=	IAC(object)
	SWAC(process)	<=	SAC(object)
Execute:			
	IWXAC(process)	<=	IAC(object)
	SRXAC(process)	>=	SAC(object)

Additionally, the module demotes high-classed processes which attempts to read or execute low-classed objects, and vice versa.

3.2.10 TOMOYO Linux

Task Oriented Management Obviates Your Onus (TOMOYO) Linux is a MAC framework for the Linux kernel developed by NTT Corporation [12, 13]. Version 2.2 of the framework is included in the Linux kernel since version 2.6.30. It is implemented using LSM and features an automatic policy generation system. This system is based on process execution history [11].

In Unix, the only way to create new processes is using the scheme "fork-exec", where a process is divided into two identical processes and then differentiated by replacing the running program in one of them. TOMOYO assigns a domain to each process, and when a new process is started, the security context transitions into this new domain from its parent process' domain. Since the domain transition rules considers the entire process lineage, the same program can belong to several domains, depending on the parent processes.

The automatic policy generation simply monitors the process invocations of the system and automatically divides them into domains dependent on their lineage. Other access requests are then transformed into accepting access rules within each domain.

In addition to MAC controls, TOMOYO provides an API which allows applications to relinquish access rights granted by the kernel, to allow for further increase in security.

The policy configuration language is modelled after a user-space view of the kernel, instead of exposing the available internal kernel objects. Permissions are attached to domains, which are string-concatenated process lineage, for example:

```
<kernel> /sbin/init /sbin/getty
```

However, the lineage may be abbreviated to match more instances of a program, like so:

```
<kernel> /sbin/getty
```

This will match more instances since it less specific, whereas the previous example would require that `init` is a parent of `getty`.

Permissions are given via absolute paths prefixed by keywords. Wildcards are allowed in all paths except for domain transition specifications, where they could lead to ambigous transitions. An example configuration snippet is shown in Listing 3.5.

Listing 3.5: Example of a TOMOYO policy

```
<kernel> /usr/sbin/smbd
use_profile 3
allow_read /etc/samba/smb.conf
allow_write /var/log/samba/\*.log
allow_create /var/tmp/smbd.\+
```

The version of TOMOYO currently in the mainline kernel only supports mediating access to files, directories, and domain transitions; while the older, non-LSM version, supports wider variety of kernel objects, such as network access.

3.3 Other Frameworks

There are multitude of other security related products for Linux. In this section we will discuss a few that has an affiliation with the above outlined security frameworks, but which targets other areas of security.

3.3.1 PaX

PaX is a patch for the Linux kernel aimed at preventing security exploits by restricting where code can be executed in memory, and by randomizing placement of stack and heap in virtual memory. It is bundled with GRSecurity and RSBAC but is developed and can be used independently.

3.3.2 sVirt

sVirt is an extension to the Linux virtualization library `libvirt` which allows for MAC frameworks to disallow compromised guests to access other guests [41]. MAC frameworks can be used to protect the system in general from compromised guests, but sVirt further allows these frameworks to label or restrict privileges of virtualized guests. Currently there is support for SELinux and AppArmor.

Chapter 4

Methods

To begin with, the available frameworks are evaluated according to a number of critera. After the result of the evaluation, two frameworks are selected for further analysis. Finally, on of these frameworks are presented as the result of this thesis.

4.1 Evaluation

In order to evaluate the frameworks with bias for the Ericsson Linux system, the TOE, this system's security requirements must first be ascertained. These requirements are then transferred into criteria which are used to evaluate the available frameworks. The set of criteria is augmented with metrics, stemming from the frameworks, some not directly applicable on the target system, but which can indicate the maturity of the project or potential future uses. The criteria are ranked, based on the security requirements, to enable selection of frameworks with respect to their merits relative to the TOE.

The security requirements should be based on Ericsson's information security policies as well as the specific requirements of the TOE.

The following criteria are given in the specification for this thesis.

- Theoretical foundation
- Practical implementation
- Community support
- Regulatory aspects
- Scalability
- Hardware issues
- Configuration complexity
- Capacity impact

After criteria expansion, ranking, and evaluation of the frameworks, two frameworks are selected for further analysis.

4.2 Further Analysis

The individual framework analysis consists of three parts; installation, policy creation and benchmarking. Installation entails modifying the target system to include the intended framework, while policy creation means creating a base policy for normal operation of the target system. Benchmarking is explained further in one of the following sections.

The results of performing these analysis parts are compared while making the final assessment of the selected frameworks.

4.2.1 Policy

In order to perform a fair comparison of the frameworks, an *abstract policy* is first created from which the actual policy is then implemented in the frameworks. The policies will be created with the security practices and demands set by Ericsson mandates and the target system.

4.2.2 Policy Implementation Metrics

The policies are implemented with support from the abstract policy as a least-privilege policy. This means that the size of the resulting policy can be used as a measure of the efficiency of the policy language. However, a subjective measure of the expressiveness of the policy language is also included, since a less expressive language might not enable enforcement of the least possible privilege available.

4.2.3 Performance Metrics

Since an addition to a system is made, the benchmarks will serve to measure slowdown and additional resource utilization compared to an unladen system.

First, the benchmark suite `lmbench` is used to measure the overhead of system calls as well as simple throughput testing.

To measure the memory usage of the frameworks, the Kernel memory, as reported at boot time, of an unmodified and a modified kernel is compared. Policy memory usage is estimated by booting into a minimal environment with and without a policy loaded and comparing the memory usage under these conditions.

If time permits, an application deployment will also be used to benchmark the frameworks, to better measure real system impact. This will most likely require an extended policy to accommodate the application's system access requirements.

4.2.4 Penetration Metrics

In order to determine the security effectiveness of the frameworks a number of kernel exploits and root-kits are exposed to the modified target systems and a tally of increased protection is then established.

Chapter 5

Results

The frameworks available for evaluation, and not obviously defunct, are presented in Section 3.2 on page 12. This chapter will expand on these descriptions with attributes collected during evaluation and analysis of the frameworks.

5.1 Evaluation

This section outlines the results of the evaluation performed on the available frameworks. First, the requirements and attributes are presented, and then the data from evaluation using these attributes.

5.1.1 Security Requirements

Due to security restrictions on information regarding the TOE, its security requirements will not be directly available in this thesis, but in a separate report. However, the initial criteria given in the thesis specification can be seen to represent a subset of the requirements. By extension, the ranking of criteria by security importance for the TOE will also not be present.

5.1.2 Criteria

The initial criteria for evaluation are given in Section 4.1 on page 19. Comparison by these criteria alone would prove difficult since they are are non-specific and thus evokes hard to compare attributes from the evaluation subjects. However, they are good examples of categories for expansion with more specific attributes. These specific attributes are chosen from the security requirements of the TOE, abstract security features, and attributes found to be held in common by several frameworks. The categories and their respective selected attributes are described further below.

The following attributes can be roughly categorized together as belonging to the initial criterion *theoretical foundation*. These attributes indicates what kind of security and flexibility a framework can offer.

Supported models If the framework supports any formal security models, this attribute simply enumerates them for further comparison.

Supported objects This attribute states to which extent access to kernel objects can be mediated. This can include for example Inter-Process Communication (IPC) resources, files, and

sockets. The more protected object types, the more potential security gain. However, a too large object coverage may have an adversarial effect on policy complexity.

Memory protection This attribute determines if the framework can shield important program and/or kernel structures from tampering via memory interfaces such as `mmap(2)`, `/dev/kmem`, or even via stack overruns.

Automatic policy generation By automatically generating a partial or full security policy for a set of normative system operations, the policy implementation process can be sped up and the risk of mistakes decreased.

Security auditing Security auditing can be used to keep clear trace of policy violations, which can in turn be used to assess the security status of the system.

Configurable auditing By configuring the security auditing, a more fine-grained security status may be acquired.

Whitelisting This attribute dictates whether or not a framework works by allowing operations with the default of denying everything, or vice versa. Whitelisting greatly simplifies creation of least-privilege policies, since only the required privileges for normal operation need be specified, instead of attempting to specify all conceivable attack vectors or anomalous operations.

Supports TPE Trusted Path Execution (TPE) is a mechanism for limiting the possibility of executing harmful binaries by only allowing execution in controlled paths.

Next are descriptions of the attributes concerning the frameworks' *practical implementation*. This is of interest from a system integration perspective, where different systems may have different operational requirements on the frameworks. It would, for example, be difficult to use a framework which is based on filesystem security attributes in a system which only uses filesystems without such attributes – for instance in a netboot environment.

Uses LSM Whether or not the framework implements access mediation via LSM or a custom set of hooks. Both has positive and negative aspects, but the interesting corollary is the framework's chance of being included in the mainline kernel, which currently requires an LSM implementation.

Code size A quantitative measure of the complexity of the framework. Size must be correlated with the overall feature set for a fair comparison.

Caches requests Caching of access requests can significantly speed up access control decisions if the number of mediated objects is large. Because access information can be requested several times per object, this can have an impact on overall system performance.

Coexists with other frameworks I can sometimes be beneficial to combine security frameworks which does not have overlapping functionality. This attribute enumerates any such frameworks which are known to work.

Supports virtualization The framework is tested to work with specific virtualization solutions, mainly those involving kernel patches, since the frameworks often are quite invasive kernel patches themselves.

Userspace dependencies This attribute lists the userspace library dependencies that the framework or its utilities might require.

Default policy By providing a default policy, for instance covering standard utilities and daemons, initial deployment of a security framework can be sped up.

Can dynamically load policy It is essential to availability that security policies can be added or modified during runtime. However, policy loading without proper privilege revocation can lead to potential security breaches.

Included in kernel There are only a few frameworks included in the Linux kernel so far, but they fare a potentially better chance of being maintained for an extended period of time, or at least receive official bug fixes from the kernel maintainers.

Since almost all available security frameworks are open source projects, a long term commitment in a business application requires a degree of operational support. At minimal, there should be some forum for discussing issues or to request bug fixes during the duration of deployment, and at the other end of the scale lies commercial support. The attributes concerning these issues can be loosely grouped under the initial criterion *community support*.

Actively maintained Whether or not the framework is seeing addition, porting to new kernel versions, or bug fixes. Abandoned projects require more upkeep and lacks the crucial expertise of the original developers if picked up by a third party.

Support channels This attribute lists the support channels through which the framework is developed and supported.

Security technologies can often be laden with export or patent restrictions, however these are usually cryptographic products, which does not directly apply on pure MAC frameworks. As mentioned when introducing the concept of TOE in the introduction, there are ISO standards for evaluation of security of products, in which MAC plays a role. There is only one attribute related to these *regulatory aspects*, because of the informal nature of Linux security frameworks.

Security certifications Any security certifications of the framework directly or as a part of another product. Common certifications are the, now deprecated, Orange Book levels and the superseding Common Criteria.

The attributes for the criteria *scalability* and *hardware issues* can be combined since they consider the same problem, hardware, but from different perspectives.

Scales with SMP systems When subjects situated on different processors contend for related information there is risk for slowdown if locking of security attributes is not properly implemented.

Requires TPM TPM is a hardware device used to store encryption keys and hash values for system verification. This attribute shows whether the framework requires such a device for normal operation.

Implementation of security policy can come to be a large part of integrating a new security system, such as these frameworks. The policy languages or methods cannot always be directly compared due to their differing natures and the differing underlying properties of the frameworks, such as available objects for mediation. The category *configuration complexity* contains the attributes regarding this process.

Configuration type This attribute considers the main method of describing the security policy in the framework, such as a policy language or a pure binary interface operated by utilities. Little heed is given to the complexity or features of the language, since these would require more work to compare thoroughly.

Alternative configuration Some frameworks offer different methods of configuration, which usually consists of abstractions and Graphical User Interfaces (GUIs) on top of the standard policy language.

Object selection This attribute queries the method by which the framework looks up, and sometimes stores, security attributes to mediated objects. Some security frameworks stores the security attributes of mediated files as filesystem attributes, in the filesystem inodes, which limits the number of filesystems which can be used in those frameworks.

The final initial criteria, *capacity impact*, covers some of the metrics used in the further analysis section below, but as provided by either third parties or the framework creators themselves.

System startup overhead This attribute shows any overhead associated with starting the system, brought on by the framework.

System call overhead This attribute is somewhat indicative of overall system slowdown by employing the framework. Different system calls may have different overhead, and since their usage is not uniform performance degradation in turn is seldom uniform.

Memory usage In most modern systems, main memory is not a premium, but for embedded systems the potential secyrity gains must be weighed against the memory usage.

5.1.3 Resulting Data

The attributes and their respective values for each framework are stored using the markup language Yet Another Markup Language (YAML) and compiled into a comparison chart in HTML with a simple Python script. YAML was chosen for its lax syntactic structure and because parsing and data conversation to Python data structures is trivial. The attributes can be seen in short form with the accompanying YAML data type in which they are represented in Appendix A.

The attributes which yielded significant results are listed in Table 5.1, Table 5.2, and Table 5.3. SMP scaling, startup overhead, and memory usage was omitted due to the lackluster data gathered for these attributes. Code size is also omitted.

The results can be interpreted differently depending on the requirements of a planned deployment.

5.2 Further Analysis

Based on the results in the evaluation, the frameworks SELinux and AppArmor were selected for further testing. These frameworks were selected based on the following attributes:

Corporate Support Both SELinux and AppArmor are included in SLES 11.2 and later, which are expected to be used in later versions of the TOE. This is essential to minimize the amount of effort for integration and for speedy support. Altough SELinux is not supported by Novell at this time, it is still a large product and the pre-packaging is beneficial for a speedy deployment. This means that TOMOYO could be considered as well, since it is included in the mainline Linux kernel and the userspace application is considerably smaller than those of SELinux. However, the version of TOMOYO currently included in the Linux kernel only supports file access controls.

LSM Both the selected frameworks are implemented using the LSM Application Programming Interface (API). This is related to the previous attribute, whereas frameworks is required to

use this API to be allowed inclusion in the Linux kernel. Of the selected frameworks, only SELinux is currently included in the Linux kernel.

Other frameworks implemented using LSM are for example TOMOYO and SMACK, both included in the Linux kernel.

File System Indenpendence The TOE keeps its root filesystem in `tmpfs` and additional cluster resources are mounted via Network File System (NFS). Both of these filesystems lack support for extended attributes, making deployment of a filesystem-independent security framework impossible. This disqualifies SELinux, but it was still selected since there is support being developed for extended attributes in NFS [31]. Also, the filesystems in use could be changed to accommodate security practices. The TOMOYO framework, like AppArmor, does not rely on extended attributes.

Extensive Object Support Since the TOE exists in a networked environment and may host a range of client applications, extensive object mediation is essential. Ideally, a security framework should protect the main data transfer mechanisms in a Unix system: files, IPC, and sockets. While SELinux supports these objects, AppArmor does not yet support IPC mediation.

Configuration The TOE is mostly administered via remote, which makes it useful to be able to configure a security framework via simple means, such as a text editor. Configuration could be done either via a remote shell, or on a remote computer, copying the configuration to the target node. Both the selected frameworks have configuration formats which allows for remote configuration. However, SELinux requires considerably more userspace tools for compiling the policy than any other framework.

Another aspect of configuration is the complexity, where SELinux has one of the most complex configuration languages of the compared frameworks.

5.2.1 Implementation

In order to further test the selected frameworks, the TOE was modified to contain each of the frameworks. This was done by replacing and adding a number of packages. Most packages were built from sources used in the OpenSuSE 11.2 Linux distribution. The new packages and their versions are listed in Table 5.4. The package names in monospace font are given as used in OpenSuSE.

The implementation was brought to a runnable state, but there was no time to develop a suitable security policy for further benchmarking and testing. Thus, no further testing was performed.

Name	SELinux	AppArmor	GRSecurity	SMACK	RSBAC	TOMOYO	LIDS	TCPA
Version	N/A	2.3	2.1.12	N/A	1.4.2	2.2	2.2.3rc7	3.2.0
Supported models	• FLASK • MLS	N/A	N/A	N/A	RBAC	N/A	N/A	N/A
Supported objects	• IPC • filesystem • capability • directory • FD • file • socket	• capability • file • socket	• capability • file • resource • socket	• IPC • file • socket	• IPC • device • file • socket • user	file	• capability • file • socket	• file • socket
Memory protection	• heap • stack	N/A	PaX	N/A	PaX	N/A	N/A	N/A
Automatic policy generation	False	True	True	False	True	True	True	False
Security auditing	True	True	True	True	True	True	True	False
Config-urable auditing	True	False	True	True	True	False	False	False
Whitelisting	True	False	True	True	False	True	False	True
Supports TPE	ish	False	True	False	True	ish	True	True
Uses LSM	True	True	False	True	False	True	True	True

Table 5.1: First part of the attribute values for the evaluated frameworks

Name	SELinux	AppArmor	GRSecurity	SMACK	RSBAC	TOMOYO	LIDS	TCPA
Caches requests	True	False	True	False	True	False	False	True
Coexists with other frameworks	PAX	N/A	PAX	N/A	• Dazuko • PAX	N/A	N/A	N/A
Supports virtualization	• KVM • Xen • sVirt	• Xen • sVirt	KVM	N/A	Xen	KVM	N/A	N/A
Userspace dependencies	See Table 5.4	• DBUS (opt.) • PAM (opt.)	PAM (opt.)	N/A	PAM (opt.)	`ncurses`	N/A	N/A
Default policy	True	False	True	True	False	False	False	False
Can dynamically load policy	True	True	True	True	True	True	True	False
Included in kernel	2.6.0	N/A	N/A	2.6.25	N/A	2.6.30	N/A	N/A
Actively maintained	True	True	True	True	True	True	False	False
Support channels	mailing list	• corporate • mailing list	• forum • mailing list	N/A	mailing list	mailing list	• forum • mailing list	N/A

Table 5.2: Sencond part of the attribute values for the evaluated frameworks

Name	SELinux	AppArmor	GRSecurity	SMACK	RSBAC	TOMOYO	LIDS	TCPA
Security certifications	EAL4+ CAPP, LSPP [5]	N/A	N/A	N/A	N/A	N/A	N/A	N/A
Requires TPM	False	False	False	False	False	False	False	True
Configuration type	C-like	ACL	ACL	DIT	CLI	CUI	iptables-like	fixed
Alternative configuration	• IDE • SPDL	N/A	N/A	N/A	N/A	• Eclipse • text file	N/A	N/A
Object selection	inode	filename	filename	inode (follow symlinks)	inode	process tree	inode	inode
System call overhead	7%	2%	<1%	5-10%	N/A	N/A	$O(n)$	N/A
Extra	Remote policy editing	N/A	• `/proc` protection • IP port randomization • PID randomization • enhanced chroot	N/A	• resource limits • user management	N/A	N/A	N/A

Table 5.3: Third and last part of the attribute values for the evaluated frameworks

Name	Version	Framework
Linux Kernel	2.6.31.6	SELinux
selinux-policy	2.20081230	SELinux
selinux-policy-refpolicy-standard	2.20081230	SELinux
libselinux1	2.0.80	SELinux
libsemanage1	2.0.31	SELinux
selinux-tools	2.0.80	SELinux
libustr	1.0	SELinux
policycoreutils	2.0.62	SELinux
checkpolicy	2.0.19	SELinux
python-selinux	2.0.80	SELinux
audit-libs-python	1.2.9	SELinux
setools-libs	3.3.5	SELinux
setools-console	3.3.5	SELinux
libcap2	2.16	SELinux
libsepol1	2.0.36	SELinux
OpenSuSE 11.2 AppArmor-patches	2.4	AppArmor
apparmor-parser	2.3.1	AppArmor
apparmor-profiles	2.3	AppArmor
apparmor-utils	2.3.1	AppArmor
libapparmor1	2.3	AppArmor
perl-libapparmor	2.3	AppArmor

Table 5.4: Replaced and added packages in the TOE

Chapter 6

Conclusions

GRsecurity, RSBAC, LIDS, and TCPA were excluded from serious consideration quite early, due to variety of reasons. GRsecurity and RSBAC does not utilize the LSM API, which means that these projects will not be able to be merged into the mainline kernel without major rewrites. This means that any user of these will have to package and keep up with new releases himself. LIDS and TCPA are not actively developed and has many policy restrictions, such as hard coded policy rules in TCPA and limited amount of security labels in LIDS.

SELinux was chosen for further evaluation due to number of reasons, but it is not currently usable with the TOE because of its reliance on filesystem extended attributes. Why was it selected then? As stated before, there is work being done on incorporating extended attributes and security labels in NFS. When this project reaches fruition, the usability of SELinux in networked environments, and among them the TOE, will increase.

Another drawback with SELinux is the complexity of the configuration language. This complexity can also be regarded as a boon, since it is a symptom of TE:s policy universality, wherein most security policies should be able to be expressed. The time and cost of developing and maintaining a security policy in SELinux is quite high, but there should be some overlap with other organizations as shown by the extensive reference policy.

AppArmor was the second selected framework for further evaluation. Unfortunately, the intended purpose of AppArmor is not that of whole-system encapsulation, but instead application firewalling. Since the TOE is not focused on applications in the same way for example a desktop environment is, this approach is less than ideal. If AppArmor would introduce some way of automating policy generation for the full system state, it would be useful in applications such as the TOE. AppArmor should not have been selected for further testing.

SMACK is a simpler alternative to SELinux, but the policy is fixed and the labels are limited. It suffers from the same issue as SELinux where it requires file system support for enforcing security. Additionally, it requires the administrator to manually label each file with desired labels, a task which is automated in SELinux. Due to the limited size of the TOE, this might be feasible, but would probably require separate housekeeping of the file to label-mappings, something which SELinux has built-in in its policy language.

On the other hand, SMACK has support for many desired kernel objects, has a small footprint, and is included in the Linux kernel. It may have been prematurely discarded.

TOMOYO has the whole-system approach lacking from AppArmor, and does not rely on filesystem attributes like SELinux and SMACK. It is also built around automatic policy generation, which allows policies to be created with less effort than for example SELinux. The major drawback is that the object mediation is limited to files in the version of TOMOYO currently included in the Linux

kernel. There is support coming for IPC and sockets in future versions, which hopefully will be included in the kernel.

In conclusion, the selected frameworks were not ideal, and two other frameworks, SMACK and TOMOYO, potentially has better properties for inclusion in the TOE.

6.1 Evaluation

The evaluation was restricted to purely functional attributes, which are great for functional comparison, but less useful for targeted evaluation. To achieve a more specific evaluation, the requirements should be further developed and tuned to the TOE. This could for instance be done by using *use-cases*, where normal system operations are exemplified and the expected behaviour is specified with regard to user behaviour.

6.2 Further Work

This thesis has merely scratched the surface while comparing security frameworks. First of all, an extended variant of this thesis would compare system performance metrics using similar security policies across all applicable security frameworks.

It would be interesting to compare the policy languages of all frameworks with regard to expressivity, ease of use, and policy implementation speed. These could potentially be hard properties to compare since they are somewhat subjective, but given small enough sample policy targets the noise should be manageable. There is also an issue of the diversity of sample policy targets.

For deployment, an in-depth cost analysis of all the frameworks would be beneficial. It is uncertain if conventional cost models such as Constructive Cost Model (COCOMO) are useful, but it is currently quite hard to estimate the cost of deploying different security frameworks in Linux.

Chapter 7

Acknowledgements

I would like to thank my thesis supervisor Ola Ågren, for putting up with my stalling with an even temper. Zackarias Andersson and Jonas Eriksson at Ericsson has been great help and endless resources of wisdom.

A big thanks to everyone at Linux Development Center at Ericsson for putting up with me and answering my silly questions. Lunchtime was especially appreciated, thank you.

Chapter 8

References

[1] Badger, L., et al. A Domain and Type Enforcement UNIX Prototype. In *Proceedings of the Fifth USENIX UNIX Security Symposium* (1995).

[2] Badger, L., et al. Practical Domain and Type Enforcement for UNIX. In *Proceedings of the 1995 IEEE Symposium on Security and Privacy* (1995), pp. 66–77.

[3] Bishop, M. *Computer Security: Art and Science*. Addison-Wesley, 2004.

[4] Chen, H., et al. Analyzing and Comparing the Protection Quality of Security Enhanced Operating Systems. In *Proceedings of the 16th Annual Network & Distributed System Security Symposium* (2009).

[5] Common Criteria Evaluation and Validation Scheme for IT Security (CCEVS). Validated Product - Red Hat Enterprise Linux Version 5.1. `http://www.niap-ccevs.org/cc-scheme/st/vid10286/`, accessed 2010-03-15.

[6] Edwards, A., et al. Runtime Verification of Authorization Hook Placement for the Linux Security Modules Framework. In *Proceedings of the 9th ACM Conference on Computer and Communications Security* (2002), pp. 225–234.

[7] Fox, M., et al. SELinux and grsecurity: A Case Study Comparing Linux Security Kernel Enhancements. `http://www.cs.virginia.edu/~jcg8f/GrsecuritySELinuxCaseStudy.pdf`, accessed 2009-09-17.

[8] Fox, M., et al. SELinux and grsecurity: A Side-by-Side Comparison of Mandatory Access Control and Access Control List Implementations. `http://www.cs.virginia.edu/~jcg8f/SELinux%20grsecurity%20paper.pdf`, accessed 2009-09-17.

[9] Fraser, T. LOMAC: MAC You Can Live With. In *Proceedings of the FREENIX Track: 2001 USENIX Annual Technical Conference* (2001).

[10] Ganapathy, V., et al. Automatic Placement of Authorization Hooks in the Linux Security Modules Framework. In *Proceedings of the 12th ACM Conference on Computer and Communications Security* (2005), pp. 330–339.

[11] Harada, T., et al. Access policy generation system based on process execution history. In *Proceedings of Network Security Forum 2003* (2003).

[12] Harada, T., et al. Task Oriented Management Obviates Your Onus on Linux. In *Proceedings of Linux Conference 2004* (2004).

[13] Harada, T., et al. Towards a Manageable Linux Security. In *Proceedings of Linux Conference 2005* (2005).

[14] IEEE. *IEEE 1003.1e: Draft Standard for Information Technology–Portable Operating System Interface (POSIX)–Part 1: System Application Program Interface (API), draft 17 (withdrawn)*, Oct. 1997. `http://wt.xpilot.org/publications/posix.1e/`.

[15] Irvine, C. E. The Reference Monitor Concept as a Unifying Principle in Computer Security Education. In *Proceedings of the First World Conference on Information Systems Security Education* (1999), pp. 27–37.

[16] Jaeger, T., et al. Consistency Analysis of Authorization Hook Placement in the Linux Security Modules Framework. *ACM Transactions on Information and System Security 7*, 2 (2004), 175–205.

[17] Jawurek, M. RSBAC - A framework for enhanced Linux system security. In *Proceedings of Conference Seminar on Dependable Distributed Systems 2005* (2005), Laboratory of Dependable Distributed Systems at RWTH Aachen University.

[18] Jim, T., et al. System Call Monitoring Using Authenticated System Calls. *IEEE Transactions on Dependable and Secure Computing 3*, 3 (2006), 216–229.

[19] Khan, S., et al. A Comprehensive Analysis of MAC Enhancements for Leveraging Distributed MAC. In *Proceedings of the International MultiConference of Engineers and Computer Scientists 2008* (2008).

[20] La Padula, L. J. Rule-Set Modeling of a Trusted Computing System. In *Information Security: An Integrated Collection of Essays*. IEEE Computer Society Press, 1995.

[21] Lampson, B. W. Protection. In *Proceedings of the Fifth Princeton Conference on Information Sciences and Systems* (1971).

[22] Landwehr, C. E. Formal Models for Computer Security. *ACM Computing Surveys 13*, 3 (1981), 247–278.

[23] Leitner, A. Counterpoint: Novell and RedHat security experts face off on AppArmor and SELinux. *Linux Magazine 69* (2006).

[24] Levin, R., et al. Policy/Mechanism Separation in Hydra. In *Proceedings of the Fifth ACM Symposium on Operating Systems Principles* (1975), pp. 132–140.

[25] Loscocco, P. A., et al. The Inevitability of Failure: The Flawed Assumptions of Security in Modern Computing Environments. In *Proceedings of the 21th National Information Systems Security Conference* (1998).

[26] Loscocco, P. A., and Smalley, S. Integrating Flexible Support for Security Policies into the Linux Operating System. In *Proceedings of the FREENIX Track: 2001 USENIX Annual Technical Conference* (2001), pp. 29–42.

[27] Ott, A. RSBAC and LSM. `http://www.rsbac.org/documentation/why_rsbac_does_not_use_lsm`, accessed 2010-01-16.

[28] Ott, A., and Fischer-Hübner, S. *The 'Rule Set Based Access Control' (RSBAC) Framework for Linux*. Division for Information Technology, Department of Computer Science, Univ., Karlstad, 2001.

[29] Pourzandi, M. A New Distributed Security Model for Linux Clusters. In *Proceedings of the FREENIX Track: 2004 USENIX Annual Technical Conference* (2004), pp. 231–236.

[30] Pourzandi, M., et al. Clusters and security: Distributed security for distributed systems. *IEEE International Symposium on Cluster Computing and the Grid 1* (2005), 96–104.

[31] Quigley, D. Labeled NFS. `http://selinuxproject.org/page/Labeled_NFS`, accessed 2010-03-09.

[32] Safford, D., et al. A Trusted Linux Client (TLC). `www.research.ibm.com/gsal/tcpa/tlc.pdf`, accessed 2010-01-18.

[33] Schaufler, C. Simplified Mandatory Access Control Kernel. `http://www.schaufler-ca.com/data/SmackWhitePaper.pdf`, accessed 2009-09-18.

[34] Spencer, R., et al. The Flask Security Architecture: System Support for Diverse Security Policies. In *Proceedings of the 8th conference on USENIX Security Symposium* (1999).

[35] Spengler, B. Why doesn't grsecurity use LSM? `http://www.grsecurity.net/lsm.php`, accessed 2009-11-17.

[36] Teo, L., et al. Supporting Access Control Policies Across Multiple Operating Systems. In *Proceedings of the 43rd Annual Southeast Regional Conference* (2005), pp. 288–293.

[37] Tobotras, B. Linux Capabilities FAQ 0.2. `http://ftp.kernel.org/pub/linux/libs/security/linux-privs/kernel-2.4/capfaq-0.2.txt`, accessed 2010-01-15.

[38] U.S. Department of Commerce. *Assessment of Access Control Systems*, 2006. National Institute of Standards and Technology Interagency Report 7316.

[39] U.S. Department of Defense. *Department of Defense Trusted Computer System Evaluation Criteria*, Dec. 1985. DOD 5200.28-STD (supersedes CSC-STD-001-83).

[40] U.S. House of Representatives. US CODE: Title 44, 3542. Definitions. `http://www.law.cornell.edu/uscode/44/3542.html`, Jan. 2008.

[41] Walsh, D. Secure Virtualization Using SELinux (sVirt). `http://danwalsh.livejournal.com/30565.html`, accessed 2010-03-15.

[42] Wang, Z., et al. Countering Kernel Rootkits with Lightweight Hook Protection. In *Proceedings of the 16th ACM Conference on Computer and Communications Security* (2009).

[43] Watson, R. N. M. Exploiting Concurrency Vulnerabilities in System Call Wrappers. *First USENIX Workshop on Offensive Technologies* (2007).

[44] Wright, C., et al. Linux Security Modules: General Security Support for the Linux Kernel. In *Proceedings of the 11th USENIX Security Symposium* (2002).

[45] Zakrzewski, M. Mandatory Access Control for Linux Clustered Servers. In *Proceedings of the Ottawa Linux Symposium 2002* (2002), pp. 618–631.

[46] Zhang, X., et al. Using CQUAL for Static Analysis of Authorization Hook Placement. In *Proceedings of the 11th USENIX Security Symposium* (2002), pp. 33–48.

Appendix A

Evaluation Criteria

Name	Type	Description
Models supported	`list`	Security model(s) implemented (MAC implicit)
Objects supported	`list`	Kernel objects that can be mediated via policy
Memory protection	`list`	Types of memory protection offered
Automatic policy generation	`bool`	-
Security auditing	`bool`	Supports auditing of policy violations?
Configurable auditing	`bool`	-
Uses whitelisting	`bool`	Denies access if not in policy
Supports TPE	`bool`	Allows trusted executables?
Uses LSM hooks	`bool`	Implemented using LSM or other way
Code size	`scalar`	Code size of kernel- and userspace source code
Caches requests	`bool`	Employs caching to increase performance?
Coexists with	`list`	Compatible kernel security solutions
Supports virtualization	`list`	Compatible virtualization frameworks
Userspace dependencies	`list`	Dependencies for building tools, etc.
Has a default policy	`bool`	Has a sane default policy?
Can load policies	`bool`	Can policies be updated during runtime?
Has been released	`bool`	Is the framework experimental?
Included since kernel	`scalar`	-
Actively maintained	`bool`	Is forward ported or developed?
Support channels	`list`	Ways of getting support
Certifications	`list`	-
Scales badly	`bool`	Has poor SMP performance?
Requires TPM	`bool`	-
Configuration type	`scalar`	Method of creating policy
Alternative configuration	`list`	Alternative policy editing
Object selection method	`scalar`	How objects are identified inside the kernel
Startup overhead	`scalar`	-
Syscall overhead	`scalar`	-
Memory usage	`scalar`	Framework and policy memory footprint
Extra features	`list`	Features that are not directly related to access control