LinSec - Linux Security Protection System

Author: Boris Dragovic
Supervisor: Graham Knight

This report is submitted as part requirement for the BSc Degree in Computer Science at University College London. It is substantially the result of my own work except where explicitly indicated in the text.

April, 2002.

Contents

Chapter 1

Abstract

Host security represents one of the most attacked links in the Internet security chain. Large proportion of the efforts to improve host security has gone into following the flawed assumption that adequate security can be provided solely in the application layer. Practice and research have shown that security mechanisms, in order to be effective, have to be implemented in the operating system layer.

Discretionary Access Control (DAC) model, implemented for access control and privilege delegation purposes in most *UNIX* systems, represents the most frequent cause of the host security breaches in the Internet environment. *LinSec* project is aimed at designing and implementing a Mandatory Access Control (MAC) model, as opposed to the existing DAC model, in the *Linux* operating system.

The envisaged MAC model is based on a combination of the existing and novel security mechanisms such as: capabilities, file system access domains and IP labeling. The *Linux* specific *LinSec* design and implementation is original in all its aspects except for the capability model which is a substantial extension of the basic POSIX 1003.6 model implemented in Linux.

LinSec was implemented in about 5,000 lines of *Linux* kernel code over a 16 week period. The preliminary test and benchmark results show that

the implemented MAC model is both efficient and effective. Furthermore, *LinSec* is easily integratable in existing *Linux* systems and does not substantially affect the target system's usability and performance.

Chapter 2

Introduction

2.1 Motivation

Numerous Internet security incidents have shown that cryptography based network tools, although indispensable for ensuring data integrity and confidentiality as well as for authentication purposes, are not sufficient to actively defend against various types of security threats. The other side of the coin is host security which usually represents "the weakest link" in the chain and is often the prime target of attackers.

Most of the current efforts aimed at improving host security rely on the flawed assumption that adequate security can be provided solely in the application layer with existing operating system security features laying below not being altered [23]. A number of well documented examples have shown that support from secure operating systems is paramount to fighting threats posed by modern computing environments [23, 30, 24]. For example, *Linux* community has been trying to eliminate the buffer overflow[1] threat for years by auditing source code of the programs available for the platform. Nonetheless, it was not before the advent of, so called, *Openwall* [3] *Linux* kernel patch that the threat was successfully fought. The application layer

[1] Please refer to Subsection 3.2.2 for explanation of *buffer overflow*.

exclusive security provision approach, as stated in [10], results in a *fortress built upon sand.*

Most of the security attacks on *UNIX* systems today rely, irrespective of the actual exploit mechanism employed, on the way in which access control and privilege delegation mechanisms are implemented in the underlying operating system. The model that *UNIX* systems follow for the purpose is, so called, *Discretionary Access Control* (DAC) model. Therefore, the DAC model is to blame for large proportion of security breaches in *UNIX* environments. When first *UNIX* systems, and the DAC mechanisms, were developed they were perfectly suited for the environment it was envisaged they would serve in. However, the environment most of the existing *UNIX* systems operate in today is far from anything people could have envisaged more than three decades ago. Research and development in the field of operating system security is constantly failing to meet the pace at which it is being challenged. Unfortunately, many operating system mechanisms, known to be seriously flawed and/or inapt for use in the new environments, such as DAC, are still widely present in the operating system design and implementation.

2.2 Project Aims

LinSec is an attempt to improve *Linux* [13] Operating System security by replacing the existing DAC model with a *Mandatory Access Control* (MAC) model and by introducing, through the MAC model, the principle of *least privilege.* Not only should the MAC approach help prevent possible security breaches but it should also enable the confinement of any successful breaches and thus lessen their impact on the overall system functioning.

There are currently several ongoing projects, like [34, 28, 9], trying to address the same issue but their acceptance has not been wide for the following reasons:

- MAC models too complex to administer.
- Substantial semantic changes to *Linux* behavior, raising the application compatibility issues.
- Difficulty of integration into the existing, running *Linux* systems.

LIDS project, for example, one of the first to provide MAC for *Linux*, suffers badly from "code rot". Numerous changes and alterations have resulted in a system difficult to understand whose behavior is full of unexpected and undocumented side effects. Once popular, today the number of people supporting LIDS is steadily decreasing.

SELinux project, on the other hand, implements a very effective MAC model and is well managed and maintained. However, it is impossible to integrate into a running *Linux* system. For many *Linux* servers on the Internet it is simply unacceptable to go offline for lengthy reinstallation and setup periods.

Apart from its academic value, one of the most important aims of *LinSec* is to be highly practical, efficient system and accepted by the Open Source community as such. The project is envisaged as a system that will easily integrate into the existing *Linux* platforms providing the highest possible level of transparency to the existing users and services by fully supporting *POSIX* standards and traditional *Linux* behavior, as long as system mandatory security policy is obeyed. The mandatory security policy itself needs to be highly flexible, reflected through its configurability.

The envisaged MAC model is largely based on modification of ideas existing in the operating systems research world for a long time but which have either never been implemented[2] at all (eg. IP Labeling, etc.) or have not been implemented in a similar setting (eg. file system access domains).

[2]as far as the author is informed.

The scope of the individual final year project is *only* the kernel portion of the overall *LinSec* project. Userspace configuration and administrative tools are external to the project and should be developed by obeying the interface provided by the kernel code.

Linux was chosen primarily because of the widely available kernel source code and because of its widespread use in the Internet environment.

2.3 Project Outcomes

LinSec development, including analysis, design and implementation phases, took roughly 16 weeks. With all of the features specified in Chapter 4 the implementation, in the form of a *Linux* kernel patch, occupies approximately 5,000 lines of kernel code (written in C) as detailed in Chapter 5. The resulting system conforms in all aspects to the envisaged model.

At the time of writing this report, *LinSec* has been presented at *Linux FEST '02*[3], Belgrade, where it has prompted considerable interest. *LinSec* is already employed at the Computer Centre, Faculty of Electrical Engineering, Belgrade. A paper is to submitted for presentation at the *ETRAN*[4] conference this June.

Userspace administrative tools were developed, according to the detailed specification, by Mr Bosko Radivojevic and Mr Veselin Mijuskovic at the *Faculty of Electrical Engineering, University of Belgrade* due to the time constraints. The resulting system is fully employable in existing *Linux* platforms with minimal disruptions due to the setup time. *LinSec* is released to the Open Source community under the terms and conditions of GNU GPL

[3] An event organized by the Open Source Network of Yugoslavia aimed at promoting the Open Source software development model and *Linux* operating system.

[4] Society for Electronics, Telecommunications, Computers, Automation and Nuclear Engineering from Belgrade, Yugoslavia, annual conference in held collaboration with the Yugoslav section of IEEE and the government of the Republic of Serbia.

license.

2.4 Conventions

The following is a list of typographical conventions used in this report:

- Sans serif: used to indicate filenames.
- *Emphasized*: used for *LinSec* specific terms and otherwise important terms.
- `Typewriter`: used for C code.
- In formulas, + sign is used to denote *union*.

2.5 Report Layout

The rest of the report is organized as follows:

- Chapter 3 analyses various security breaches widespread in the Internet environment and traces their causes down to operating system mechanisms that fail to respond to them. This Chapter gives rationale behind the decision to replace DAC with MAC.
- *LinSec* design is presented, on a rather high level which can be applied to various variants of *UNIX*, in Chapter 4.
- The actual implementation of the specified design in *Linux* kernel is explained in Chapter 5. The chapter also describes issues that arose during the implementation due to the attempt to modify some of the core kernel mechanisms and ways in which they were resolved.
- Performance overhead of *LinSec* was assessed in a series of benchmarks as presented in Chapter 7.

- Because of the *LinSec* nature, approach taken for testing the implementation differs from standard, prescribed, ways and is put forward in Chapter 6.

- Finally, Chapter 8 concludes the report with a short summary, envisaged future of the project and a retrospective view on the project.

Chapter 3

OS Security Background

3.1 Introduction

First step in designing security mechanisms is to study potential threats the system in question might be, or is, exposed to. Next, the "features" of the system in question enabling particular types of threats need to be identified. Lastly, the identified "features" need to be redesigned or removed so as to thwart the security attacks.

This section is an analysis of the security threats affecting the *Linux* operating system (OS), mechanisms by which they operate and design points in the *Linux* OS that provide scope for the attacks. General solutions to the problems are presented as well. Although the focus of the Chapter is on the *Linux* OS, as it is the OS of the choice for the project, all details mentioned apply to most of the commercial and server side operating systems available on the market today.

3.2 Attacks - Facts

The past several years have seen tremendous increase in the number of host based security attacks and breaches. This effect should be attributed to the

expansion in the size of the Internet reflected in the number of hosts exposed to the worldwide community.

3.2.1 Facts

Several organizations have been established during the past years to focus on the security threats arising in the Internet environment. Some of them are *The Security Group at the Carnegie Melon Software Engineering Centre* (CERT), *System Administration, Networking and Security Institute* (SANS) and *U.S. DoE Computer Incident Advisory Capability* (CIAC). One of the interests of the agencies were statistics related to the security attacks that take place daily.

To illustrate the rate of increase in the number of attacks relevant statistics originating from *CERT's* web site [8] is included in Table 3.2.1.

Year	*1990*	*1991*	*1992*	*1993*	*1994*	*1995*	*1996*
Incidents	252	406	773	1,334	2,340	2,412	2,573
Year	*1997*	*1998*	*1999*	*2000*	*2001*	*2002*	*2003*
Incidents	2,134	3,734	9,859	21,756	52,658	n/a	n/a

Table 3.1: Number of Incidents per Year

To give meaning to the striking numbers in Table 3.2.1 the following list names the top seven vulnerabilities exploited in 2001 on *UNIX* platforms, according to the *SANS Institute* [17]:

1. Buffer Overflows in RPC Services
2. Sendmail Vulnerabilities
3. BIND Weaknesses
4. R Commands (rsh, rlogin, etc.)
5. LPD (remote printing protocol daemon)

6. sadmind and mountd

7. Default SNMP Strings

The proportion of the numbers from Table 3.2.1 that can be attributed to the above vulnerabilities is roughly proportional to the percentage of *UNIX* servers on the Internet. And the number of affected *Linux* systems is proportional to the number of *Linux* servers among other *UNIX* servers. Emphasis should be placed on the fact that the number of reported incidents is very different from the actual number of incidents that occurred but were not reported or were not detected.

3.2.2 Types of Attacks

The attacks, including the ones mentioned in Section 3.2.1, can be categorized in several groups by the nature of their functioning [9]:

- *Buffer Overflows*: a problem endemic to C programs that provide poor bounds checking on input received from outside environment resulting in subverting the intended flow of program and thus forcing execution of attacker supplied code[1]. Five out of the seven vulnerabilities mentioned in Section 3.2.1 belong to this category.

- *Race Conditions*: intended functioning of a program can be subverted so that it produces side effects that compromise system security. Some of "R Commands" vulnerabilities from Section 3.2.1 fall into this category.

[1] In C, both function return address and local variables are kept on stack. By supplying a function argument longer than the local variable it is subsequently copied to, an attacker may overwrite contents of the stack, including the return address. If carefully chosen, the overwritten return address can point to custom supplied piece of code eg. code for executing system shell.

- *Special Character Processing*: rely on "fooling" character processing programs by providing user input that causes the program to relinquish control to the attacker. Programs affected are usually CGI scripts running in privileged mode and accessible by everyone.

As can be seen, by far the most frequent are attacks that are based on the buffer overflow technique.

3.2.3 Attack Anatomy

All host based attacks, including *Denial of Service* (DOS) and *Distributed Denial of Service* (DDOS) in a slightly different sense, irrelevant of the category (Section 3.2.2) they belong to, are aimed at gaining attacker privileges under which the attacked program is running. The applications attacked are usually the ones labeled as "trusted" by the system administrator. A "trusted" program is a program that is running with privileges that enable it to perform sensitive system operations. In most cases, "trusted" programs run under *superuser* privileges. Once such a program is subverted attacker gains all of the privileges of the program, which usually turns disastrous for the system.

3.3 OS Background

Once the threats are identified the question remains of the right system layer to introduce appropriate security measures to:

- Application layer, *or*
- Operating System layer

Since the application level is the one being directly attacked, one might think that it is the appropriate point to thwart the attacks at. However, to assure the complete absence of security vulnerabilities expensive manual

verification of every single application has to be carried out. Taking into account the sheer size of the application space and the rate of its growth reveals the impossibility of such an audit.

The short analysis of host based security attacks in Section 3.2 pointed at the common aim of most of the exploits — to gain the privileges of the "trusted" applications. The fact that overall result is the same no matter which application is exploited, or the way in which it was exploited, points at a problem at the layer below the application layer - the Operating System layer [9, 23]. This conclusion is not intended to eliminate security concerns when developing applications. Although addressing security problems solely in the application space does not suffice, it is still a valuable aspect of the overall system security.

The following mechanisms and aspects of the most commercial and server side Operating Systems enable the described attack behavior [9, 23, 24, 22, 30]:

- *Discretionary Access Controls* (DAC)
- *Authorizations*
- *Default OS configuration*

3.3.1 Discretionary Access Controls

Discretionary Access Control (DAC) means that the owner of an object can manage permissions for the object at his own discretion. In effect, owner of an object can decide who to grant permissions to access and use the object to without the decision being questioned by the OS. An example of this is permissions associated with *UNIX* files which the owner of the file is allowed to modify with no restrictions. It is *DAC* that is to blame for gaining-the-privileges portion of the attack mechanisms. Once the attacker gets hold of "flow of execution" of the attacked program it may manipulate any of

the objects owned by the uid running the process [30, 23, 22]. Individual attempts have been made[2] at configuring the DAC in a fine-grained enough manner to minimize the described effect but all of them yielded complex and bulky solutions that were impossible to maintain and control.

3.3.2 Authorizations

Usually, only two major user categories are supported by *DAC*:

- *superuser, and*
- *the rest of the world*

DAC model, by its definition, dictates that all requests made on behalf of the superuser[3] must be granted and that their legitimacy is never questioned. Superuser, as in DAC model, owns all system objects and can, at his own discretion, grant or refuse access privileges to any of them to "the rest of the world". Notion of the superuser, as such, represents a single point of vulnerability[4] in a system.

All other users in a system, popularly named "the rest of the world", undergo full DAC checks on every request made on their behalf. Non-superuser users can grant or refuse access privileges to system objects, owned by them, to other users, apart from the superuser to whom all access requests shall always be granted.

As can be suspected, most of the attacks are aimed at programs running under *superuser* privileges as subverting them means obtaining unlimited access to the host system.

Probably the most widely exploited program, over the past several years, in *Linux* environment, has been *Sendmail*, *Mail Transport Agent* (MTA).

[2]These are documented on various *UNIX* admin web sites and in related mail forums.

[3]by processes owned by root.

[4]if superuser account on a system is compromised an attacker can claim full control of the system.

Sendmail, by default, runs under superuser privileges. Buffer overflow exploit scripts are available on the Internet, for various versions of *Sendmail*, that cause the attacked *Sendmail* to relinquish control to the attacker usually by launching a root shell.

3.3.3 Default OS configuration

Listed as number 1 in SANS's top twenty most critical Internet security vulnerabilities affecting all systems [17] is:

> Default installs of Operating Systems and applications

Most of the Operating System distributions offer *user-friendly* installation procedures and scripts whose main aim is to get the system up and running as fast as possible with the administrator having to perform least amount of work. These types of installation and setup need to cater for various end users and thus install much more software than needed in any particular case. From vendors' point of view it is always better to enable functions that are not needed than to make the user install additional functions separately. In the end, users are not even aware of all the software installed on their system and fail to maintain it and patch promptly as security threats are discovered. Furthermore, many system services and "trusted" programs run with coarse grained privileges that far exceed their actual requirements. A security flaw in any of these enables an attacker to gain *superuser* privileges.

3.4 Securing OS

The *NSA Orange Book* [30] is the most quoted source with respect to operating system security requirements and evaluation criteria. It defines five secure levels for operating systems along with their functional requirements (increasingly more secure):

- *C2*: Authentication[5], DAC[6].
- *B1*: *Mandatory Access Control*[7] (MAC), Audit[8]
- *B2*: Structured Security[9], Elimination of Storage Covert Channels
- *B3*: Minimized Trusted Computing Base[10] (TCB), Elimination of Timing Covert Channels
- *A1*: Proven Security[11] (non functional requirement)

Most of the commercial and server side operating systems fall into *C2* category and so does *Linux*. To advance from *C2* category the crucial functional requirement is *Mandatory Access Controls*. *MAC* mechanisms are aimed directly at eliminating the problems described so far and attributed largely to *DAC*. *MAC* model relies heavily on *least privilege* approach to system privilege allocation.

3.4.1 Mandatory Access Control (MAC)

In general terms, mandatory security policy represents any security policy that is defined strictly by system security policy administrator along with any policy attributes associated [23]. Mandatory security policy can be divided into subpolicies, all mandatory by their nature, demonstrating the recursive nature of the definition.

Mandatory Access Control (MAC) can be viewed as a subpolicy of mandatory security policy as well as the mandatory security policy as a whole if no other mandatory policies are implemented in the system. *MAC*

[5]Enables identification of the users making system requests.

[6]Users define control over their objects at their own discretion.

[7]System administrators define system access control policy, not users.

[8]System source code audit to identify sources and means of attacks and eliminate them

[9]Multi-layer security.

[10]Minimize the amount of security-relevant code in the system.

[11]Proven in practice.

policy specifies how certain subjects can access operating system objects and services [23]. There are two fundamental implications of the *MAC* approach [23]:

- Users can no longer manipulate access control attributes of the objects they own at their own discretion, *and*
- Privileges associated with a process are determined by appropriate *MAC* mechanisms, based on relevant mandatory security policy settings, on per task basis.

Since the inception of *MAC* numerous mechanisms of implementing it have been researched, some of which are: *type enforcement* and *domain type enforcement* [22, 24], *role based access control* [11], *SubDomains* [9], *capabilities* [5, 18, 19, 32] etc. Several attempts were even made at providing *MAC* through *DAC* [23] but they failed due to complexity incurred. Standards like *IEEE POSIX 1003.6* were also developed to support *MAC*.

3.4.2 Least Privilege principle

In every system a number of applications require special privileges in order to perform some system task eg. system services in *DAC* run with *superuser* privileges. If the sets of privileges associated with such applications could be made fine-grained enough, as close to minimal needed for the task as possible, unlike in the *DAC* example above, a damage resulting from a possible vulnerability exploit would be confined to only a portion of the system accessible by the privileges thus obtained. Therefore, the mandatory security mechanisms of an operating system should obey the principle of *least privilege* which states that any process in the system should be allocated only the *absolutely minimal* set of privileges needed to successfully perform the desired task. Any additional, not needed, privilege possessed by a process increases the damage incurred if a vulnerability in the program is

exploited. Therefore, any mandatory security system should provide scope for implementation of the *least privilege* principle.

3.5 Conclusion

By the careful analysis of the most wide spread attacks compromising host security in the Internet environment, presented in the Chapter, it was shown that existing operating system access control and privilege management mechanisms need to be redesigned to be able to survive in the increasingly insecure environment. The analysis exposed *Discretionary Access Control* model as the most security critical in current operating systems. Research in the field of computer security, as well as standards developed in the past several years, have recognized *Mandatory Security Policies* and *Mandatory Access Control*, in particular, as a must replacement for existing *DAC*. *MAC*, in combination with the *Least Privilege principle*, can not only provide security breach prevention but also breach and damage confinement within a system.

Chapter 4

LinSec Security Architecture

4.1 Introduction

4.1.1 LinSec Design Aims

The main aim of the *LinSec* project is to develop a *MAC* model for *Linux* operating system based on *capabilities* and file system *access domains* and thus provide *B1* security level, described in Chapter 3. The fact that the desired system should be easily integratable in already running *Linux* installations, with minimum disruptions, has influenced the design.

4.1.2 Chapter Layout

This Chapter provides description of the overall *LinSec* design and architecture. Implementation of the architecture in the *Linux* kernel is topic of Chapter 5. Firstly, the notion of *capabilities* is presented followed by all of the aspects of the *LinSec Capability* model (Section 4.2). Secondly, *File System Access Domains* (Section 4.3) are described in the same manner. Thirdly, a special form of mandatory network access control, named *IP Labeling* (Section 4.4), is explained. And finally, *LinSec MAC* security policy based on *capabilities* and file system *access domains* is put forward binding

the contents of the previous sections (Section 4.5).

4.2 LinSec Capability Model

4.2.1 Definition and Background

A *Capability* is a token possessed by an operating system subject granting access to one or more operating system objects. Subjects are considered to be active entities within an running operating system eg. a process. Subjects may also be regarded as objects for some operations eg. a process (subject) sending signal to another process (object). Objects are entities on which an operation is performed.

Whereas *Access Control List* (ACL) access control model bases its decisions on identity of the subject requesting access to an object (as with each object a list of subjects and allowed access modes is associated), *Capability* model bases its decisions on possession of the appropriate token by the subject irrespective of its identity.

As of version 2.2.0 a limited support for *POSIX* capabilities is implemented, through a very basic[1] form of *Process Capabilities*, in *Linux* kernel. *LinSec* extends it to support *User Capabilities* (Subsection 4.2.5) and *Executable File Capabilities* (Subsection 4.2.4), which were neither supported by *Linux* kernel nor was there a intention, among *Linux* community, to support them.

[1] Just about enough to claim POSIX compliance.

4.2.2 POSIX 1003.6 and Capabilities

POSIX 1003.6[2] defines, among other security related features, so called, *Process Capabilities* in the following manner (quoted from the Linux Capability FAQ [33]):

> A process has three sets of bitmaps called the inheritable(I), permitted(P), and effective(E) capabilities. Each capability is implemented as a bit in each of these bitmaps which is either set or unset. When a process tries to do a privileged operation, the operating system will check the appropriate bit in the effective set of the process (instead of checking whether the effective uid of the process i 0 as is normally done).
>
> The permitted set of the process indicates the capabilities the process can use. The process can have capabilities set in the permitted set that are not in the effective set. This indicates that the process has temporarily disabled this capability. A process is allowed to set a bit in its effective set only if it is available in the permitted set. The distinction between effective and permitted exists so that processes can "bracket" operations that need privilege.
>
> The inheritable capabilities are the capabilities of the current process that should be inherited by a program executed by the current process. The permitted set of a process is masked against the inheritable set during exec(). Nothing special happens during fork() or clone(). Child processes and threads are given an exact copy of

[2]POSIX 1003.6 has been dropped recently after ten years of development and is to be super seeded by a new document. Capability definition, however, is not expected to change so the reference to the standard is still valuable. POSIX 1003.6 was formed from POSIX 1003.1e and POSIX 1003.2c.

```
the capabilities of the parent process.
```

POSIX 1003.6, as such, does not define neither the notion of *Executable File Capabilities* or the notion of *User Capabilities*.

4.2.3 Overview

LinSec retains, from the *Linux POSIX* capability framework, the the notion of capabilities being solely unsigned integer values. Groups of capabilities can therefore be represented as bitmaps in which each bit represents a separate capability depending on its position within the bitmap. Manipulation of capability sets defined this way can be accomplished by simple arithmetic operations. The performance improvement over alternative solutions thus obtained is the primary reason for adoption of the simple representation.

All processes in *Linux*, except for the process 0[3], are running images of executable files and are owned by a user. Therefore, *LinSec* supports *Executable File Capabilities* (Subsection 4.2.4) and *User Capabilities* (Subsection 4.2.5) which are used together to compute (Subsection 4.2.9) *Process Capabilities* (Subsection 4.2.7) of a process created when the executable file is run by the user. *Process Capabilities* are used for access control checks.

LinSec also uses capabilities to implement system *boot phase protection* (Subsection 4.2.10), various types of *process protection* (Subsection 4.2.11) and *INET socket protection* (Subsection 4.2.12).

To accomplish all of the design aims, several new, *LinSec* specific, capabilities had to be introduced (Subsection 4.2.13) in addition to the existing ones.

4.2.4 Executable File Capabilities

Although there were some discussions about *Executable File Capabilities* on *Linux* development forums, no actual work has been ever done on them.

[3]Process with PID 0 represents an image of, otherwise not runnable, *Linux* kernel.

With each executable file in the system there are three sets of capabilities, corresponding to the three capability sets defined for processes by POSIX 1003.6, associated:

- *Allowed set* (fA): capabilities that can be inherited from the process that executes this executable.
- *Forced set* (fF): capabilities that must be contained in the permitted/effective (Subsection 4.2.7) set of a process running this executable.
- *Effective set*: capabilities that will be copied from permitted to effective capability set (Subsection 4.2.7) of a process that invoked this executable.

4.2.5 User Capabilities

Unlike the *Process Capabilities*, *User Capabilities* are an idea completely novel to *Linux*. With each user in the system there are two capability sets associated:

- *User Permitted set* (uP): capability set used to reflect user privileges in the system (via processes run under the user's uid).
- *User Bounding set* (uB): maximum capability set that a process running under the user's uid can obtain during its lifetime.

4.2.6 User Capability Groups

Capability groups represent an idea analogous to user groups in standard *UNIX* implementations. Each capability group is made of one or more capabilities and each user may be a member of one or more capability groups. Capability group 0 is denoted the default for all users.

The reason for introducing capability groups (including the default capability group) is the ease of *LinSec* setup and maintenance as in most systems users can be naturally grouped in several categories with respect to required system privileges and trust. Existence of capability groups reflects this trend and saves administrators from having to specify *uP* (Subsection 4.2.5) for each user individually.

Capability groups take part in the computation of *Process Capabilities* (Subsections 4.2.7 and 4.2.9).

4.2.7 Process Capabilities

Linux kernel, as of version 2.2, supports *Process Capabilities*, as defined by POSIX 1003.6. With each process in a system three capability sets are associated:

- *Inheritable set* (pI): set of capabilities that can be inherited by a new process after a binary is executed by the running process.
- *Permitted set* (pP): maximum set of capabilities that a process may acquire during its lifetime i.e.. that can be in pE (below).
- *Effective set* (pE): set of capabilities that are currently used for access control.

LinSec Process Capabilities build on a slightly modified[4], still POSIX 1003.6 compliant, version of the *Process Capabilities* implemented in the *Linux* kernel.

Process capabilities are computed (Subsection 4.2.9), by *LinSec* specific algorithm, from File Capabilities (Subsection 4.2.5) and User Capabilities (Subsection 4.2.5). It is process capabilities that are used for access control

[4]As stated in Section 5.8.

checks in LinSec as they reflect both owning user's and application's privileges. When a process issues a access request for an capability protected object its *effective* capability set is checked for the required capabilities.

4.2.8 Global Bounds

To be able to limit privileges of any process in the system a global bounding capability set is introduced and denoted as *gB*. gB has system wide effect and describes the maximum possible set of privileges any process can reach during its lifetime in the system. By no means can a process posses, in any of its capability sets (Subsection 4.2.7), a capability which is not in gB. Exactly how gB works can be seen in the Subsection 4.2.9.

4.2.9 Capability Inheritance Algorithm

LinSec Capability Inheritance Algorithm is used to compute process capabilities and it builds on the algorithm existing in the *Linux* kernel, as part of the *Process Capabilities* support.

Capability Inheritance Algorithm implemented in *Linux* kernel has three steps (quoted from `fs/exec.c` where it is implemented):

```
      pI' = pI
(***) pP' = (fP & X) | (fI & pI)
      pE' = pP' & fE          [NB. fE is 0 or ~0]

I=Inheritable, P=Permitted, E=Effective // p=process, f=file
' indicates post-exec(), and X is the global 'cap_bset'.
```

As there is no support for executable file capabilities in existing *Linux* kernel, the variables in the algorithm corresponding to them are hardcoded

as either maximal possible (for superuser processes) or as zero (for non superuser processes). This causes the resulting effective capability set to be either full capability set (for the superuser owned processes) or empty one (for non superuser owned processes).

The *LinSec* specific algorithm also has three steps and is an evolved version of the above, *Linux* implemented, algorithm:

1. $pI^{\star} = pI$
2. $pP^{\star} = (fF|(fA\&(pI^{\star}|uP^{\circ})))\&uB\&gB$
3. $pE^{\star} = (pP^{\star}\&fE)$

Where:

- pI, pE and pP are user capability sets as specified in Subsection 4.2.5.
- $pI^{\star}$, $pP^{\star}$ and $pE^{\star}$ are capability sets that replace pI, pP and pE respectively for the current process after the execution of the algorithm.
- fA, fF and fE are the executable file's capability sets as specified in Subsection 4.2.4.
- uP° is user permitted set obtained by the formula: $uP^{\circ} = uP|\gamma_m P|...|\gamma_n P$, where uP is user permitted capability set as specified in Subsection 4.2.6 and $\gamma_m P$ to $\gamma_n P$ are capability sets representing capability groups the users is a member of as specified in Subsection 4.2.6.
- uB is user bounding capability set as specified in Subsection 4.2.5.
- gB is global bounding set as specified in Subsection 4.2.8.

The first step of the *LinSec* algorithm is left unchanged (from the *Linux* implementation) as neither *User Capabilities* or *Executable File Capabilities* are designed to affect a process' inheritable capability set.

The second step has suffered most alterations. The thinking behind computing the new permitted ($pP^\star$) capability set is (starting by the innermost brackets):

- $pP^\star$ needs to reflect both the capabilities inherited from the existing process (dictated by POSIX 1003.6) and capabilities contained in permitted capability set of the process owner (to reflect privileges of the user), *but*
- only to the extent allowed for the newly executed binary (logical AND with fA), to ensure least privilege principle is followed.
- However, $pP^\star$ needs to contain capabilities necessary for the executed binary to perform its task correctly (logical OR with fF).
- Finally, no capabilities representing privileges that would exceed maximum allowed for the process owner (uB) or for the system as a whole (gB) must be included in $pP^\star$.

The third step ensures that the process has, in its effective capability set, all capabilities needed by the newly executed binary to perform its task (fE) that it is allowed to have (logical AND with $pP^\star$).

The algorithm (Subsection 4.2.7) is triggered by two system events in *LinSec* capability model implementation:

- Current process executes a binary (both in *Linux* and *LinSec* capability model implementations), *and*
- Process changes ownership (uid) (in *LinSec* capability model implementation only).

In the former case, the algorithm is invoked to reflect capabilities of the new program that is being started while in the latter case, it is used to reflect capabilities of the new process owner.

4.2.10 Capability-Based System Boot Monitor

In many cases, after successfully penetrating target system, attackers install some sort of a back door to be able to return to the system at some later stage without the need to replay the intrusion. A considerable proportion of back doors are set up each time the system is booted or rely in some other sense on programs planted by attackers and executed during the system boot. In order to prevent this type of scenario LinSec introduces the notion of a monitored boot phase. A new, CAP_SYS_BOOTTIME, capability is introduced for the purpose. Every process spawned during the boot phase needs to have the capability in its effective capability set. If this requirement is not fulfilled the offending process is killed[5].

4.2.11 Capability-Based Process Protection

Capability-Based Process Protection is best explained through an example. Assume two processes: process A and process B. For process A to send a signal to some other process B, *Linux* requires either A to be a superuser owned process or A and B to be owned by the same user. As some signals can be potentially fatal for the receiving process this simple policy was not acceptable to *LinSec* primarily as root user should not be allowed to terminate system services that are running with uid of 0 (root) in *Linux*. Rather than change the whole process ownership philosophy of *Linux*, which would certainly break the compatibility between platforms and which would violate *POSIX 1003.1* set of standards, the *LinSec* solution is to introduce a set of capabilities to be used in controlling how signals are sent and received. The solution does not replace the current mechanisms, it builds up on them. The introduced capabilities and their meanings are:

- CAP_PROC_PROTECTED: process that has this capability in its ef-

[5]Terminated by the kernel.

fective capability set will not receive any signals unless the sending process has CAP_PROC_GOD capability in its effective capability set.

- CAP_PROC_UNKILLABLE: process with this capability in its effective capability set will not receive fatal signals 2, 3, 9 and 15 unless the sending process has CAP_PROC_GOD in its effective capability set.

- CAP_PROC_GOD: process that has this capability in its effective capability set can send signals to processes with CAP_PROC_PROTECTED and CAP_PROC_UNKILLABLE capabilities in their effective capability set.

Furthermore, in some cases it might prove valuable to hide certain process related information, or even a whole processes, from the eyes of users. Examples of desired invisible processes are various system monitoring programs or intrusion detection systems. For this purpose two more capabilities are introduced by *LinSec*:

- CAP_PROC_HIDDEN[6]: processes with this capability in their effective capability sets are not listed in */proc* and are therefore invisible to system utilities like ps, top etc.

- CAP_NET_HIDDEN: data about INET network connections using TCP, UDP or raw IP of processes having this capability in their effective capability sets are omitted from */proc*. Therefore, utilities like netstat etc. do not list the network info for the processes.

[6] */proc* represents standard mount point for *Linux proc* file system. *Proc* is a virtual file system that has a role of kernel — userspace interface. Its main role is to provide system status information to userspace programs.

4.2.12 INET Socket Capability-Based Protection

As *Linux* has very little protection for IPC mechanisms and none in particular for INET sockets, the desire was felt to extend the capability model to cover that aspect of the system as well.

LinSec allows a set of capabilities to be associated with an bound socket of INET family by the owning process denoting capabilities required for *local* processes to communicate to the socket. This mechanism enables fine grained control of who connects and sends messages to a certain socket locally on per socket basis. In conjunction with traditional firewall solutions a complete protection for sockets can, thus, be established both regarding requests coming from network (handled by a firewall) or the ones coming from the local machine (handled by *LinSec*). Furthermore, the mechanism enables administrators to run services for strictly defined groups of users.

4.2.13 New Capabilities Introduced

Several other capabilities had to be introduced in addition to the existing *POSIX* and *Linux* specific capabilities to enable correct functioning of *LinSec*:

- CAP_PROC_PROTECTED: refer to Subsection 4.2.11.
- CAP_PROC_UNKILLABLE: refer to Subsection 4.2.11.
- CAP_PROC_GOD: refer to Subsection 4.2.11.
- CAP_PROC_HIDDEN: refer to Subsection 4.2.11.
- CAP_NET_HIDDEN: refer to Subsection 4.2.11.
- CAP_SYS_BOOTTIME: refer to Subsection 4.2.10.
- CAP_MOD_CAP: a process that has this capability in its effective set is allowed to modify its own permitted and effective capability sets.

This feature is necessary for the correct operation of userspace *LinSec* administrative tools[7].

- CAP_ACD_OVERRIDE: a process with this capability in its effective capability set bypasses *LinSec* file system access domain control mechanisms (Section 4.3). This feature is necessary for the correct operation of userspace *LinSec* administrative tools.

- CAP_LINSEC_ADMIN: a process with this capability in its effective capability set can configure *LinSec* mandatory security policy (Section 4.5).

4.2.14 LD_PRELOAD Attack

One particular attack that kept recurring for years in different forms is so called "LD_PRELOAD" attack. In *Linux*, "LD_PRELOAD" is an environment variable that specifies which shared libraries are to be loaded in programs at runtime. The "LD_PRELOAD" attack affects *LinSec* as it is possible for an attacker to gain capabilities of other programs by executing custom code contained in the "LD_PRELOAD" variable. To circumvent this type of attack, *LinSec* removes all capabilities from a process executing a binary if "LD_PRELOAD" environment variable is specified at the time of the execution.

The action might be considered drastic but in cases, such as this, when, from the system's point of view, there is only one, rather coarse, way of recognizing potential problems, dropping the offending process' privileges is the least that can be done to prevent a potential security breach. Further-

[7]To perform *LinSec* administrative tasks, a process needs to have CAP_LINSEC_ADMIN capability in its effective capability set. As static allocation of the capability to any program is regarded risky (due to eg. buffer overflow attacks on the capability model itself), the process is allowed to modify its capability sets, after the user that invoked it has provided correct administrative password.

more, as there are other mechanisms, apart from the use of *LD_PRELOAD* environment variable, for preloading library code, the mechanism does not affect functionality of the system as a whole.

4.3 LinSec Filesystem Access Domains

4.3.1 Background and Definition

A *File System Access Domain* represents a portion of a file system visible and accessible by a process. It effectively creates a file system cage or a sandbox to which the running process is confined. *File System Access Domains* do not replace the traditional *Linux* file system access controls, they operate at a higher level, as illustrated in Figure 4.3.1.

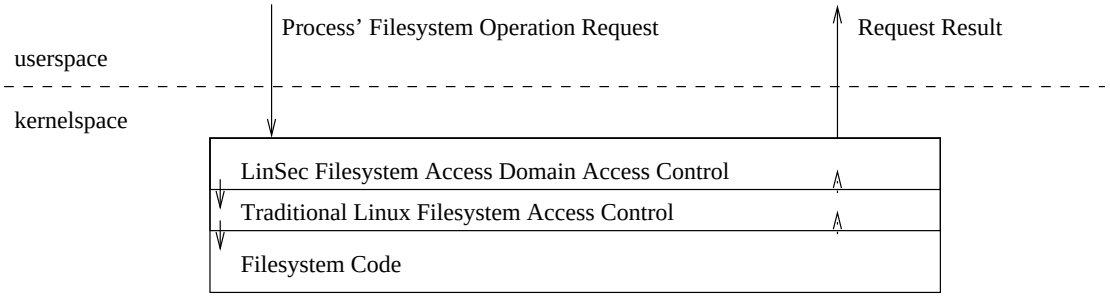

Figure 4.1: How LinSec FS Access Domain model fits Linux FS subsystem

The notion of *File System Access Domains* is best illustrated by an example: Figure 4.3.1 depicts a possible File System Access Domain of an process. By including */etc* and */usr* the process is restricted to the portion of the file system represented by the subtrees below the nodes respectively (denoted by the outer ovals). However, not the whole subtrees are accessible by the process as the file */etc/shadow* and all files below the directory */usr/local* are excluded from the FS Access Domain (denoted by dashed inner ovals in the Figure).

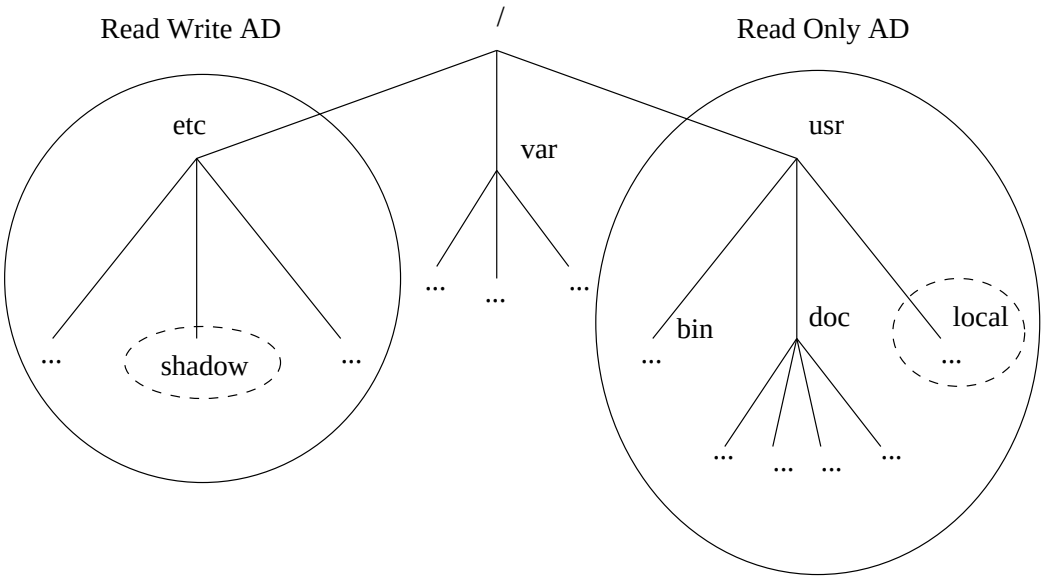

Figure 4.2: LinSec Access Domain Example

LinSec File System Access Domain can be further subdivided into:

- *Read Only* Access Domain
- *Read-Write* Access Domain

Read Only Access Domain denotes a portion of a file system that can be used for read access only and *Read Write* Access Domain denotes a portion of a file system that can be used both for read and write access. The latter Access Domain is *Read-Write* as opposed to *Write Only* as this avoids overlap in cases where files can be both read and written (in which case they would have to be duplicated in both of the Access Domains).

LinSec has no feature to prevent overlaps between the two access domains resulting from the ways in which they are configured in each particular case. Inclusion of such a feature is not regarded necessary as:

- Possible overlaps can not result in a *LinSec* system failure.

- It would make the design considerably more complicated.

In cases of overlap between the access domains, the order in which process' access domains are checked, on file system access request (Subsection 4.3.9), becomes important.

In the context of the example illustrated by the Figure 4.3.1 the process in question can access all files below */etc*, with exception of */etc/shadow*, but only for reading. Writing to this subtree will not be permitted.

File System Access Domains are built upon the traditional *UNIX* idea of changing root directory (popularly called *chroot*) for a process in order to confine it to a subtree of the file system. The *chroot* approach proved to be very inflexible as it is only capable of confining a process to a single whole subtree of the main file system tree.

LinSec File System Access Domain is abbreviated *AD* in the rest of the report.

4.3.2 Overview

All Linux processes, except for the process 0, are running image of an executable file and are owned by a user. Therefore Process ADs (Subsection 4.3.7) have to reflect both User AD privileges (Subsection 4.3.6) and Executable File AD privileges (Subsection 4.3.5). Furthermore, Process ADs have to be inherited through execution chains and through the changes of process ownership in a manner that obeys the principle of *Least Privilege* (Subsection 4.3.8).

4.3.3 Access Domain Elements

LinSec ADs are described in terms of directories and files that are referred to as AD *Elements*. Each AD Element consists of description of the directory or file it represents and a flag. The flag denotes whether an AD Element

can be inherited (Subsection 4.3.8) and whether it represents exclusion from AD (the example, Subsection 4.3.1).

4.3.4 Access Domain Groups

AD *Groups* are used for grouping individual AD Elements together to aid the ease of system configuration. The idea of AD Groups, analogous to the idea of Capability Groups (Subsection 4.2.6), exploits the fact that typical system configuration will require only a small number of very similar, and often the same, ADs. The number of the AD Groups that can be created in a system ensures that the possible granularity of mandatory security policy is not limited severely.

AD Group with id 0 represents the default AD Group and is treated in a special way (Subsection 4.3.5).

4.3.5 Executable File ADs

With every executable file two sets of AD Groups can be associated:

- *Read Only* AD Groups, *and*
- *Read Write* AD Groups.

Elements of which form *Read Only* and *Read Write* ADs of a process respectively, as specified in Subsection 4.3.1. Executable File ADs represent minimal portions of a file system that the running image of the executable needs to be able to access for its correct operation.

The AD Group 0 (Subsection 4.3.4) is forced into the executable files *Read Write* AD, as the default AD group.

4.3.6 User Access Domains

In analogy to the executable file ADs (Subsection 4.3.5), with every user in the system two sets of AD Groups can be associated:

- *Read Only* AD Groups, *and*
- *Read Write* AD Groups

Forming *Read Only* and *Read Write* ADs for the user respectively. In addition, another AD can be associated with every user:

- *User Default* Read Write AD

which contains user specific AD Elements such as the user's home dir, mail spool dir etc.

4.3.7 Process Access Domains

Since processes are the only active entities in an operating systems, AD associated with each of them needs to reflect permissions (ADs in this context) of:

- executable file (Subsection 4.3.5) whose image a process is running, *and*
- user on whose behalf it is running.

Therefore, each process' AD is split into:

- User AD (as specified in Subsection 4.3.6), *and*
- Executable File related AD.

The latter consisting of:

- *Non-Inheritable* Read Only AD,
- *Non-Inheritable* Read Write AD,
- *Inheritable* Read Only AD, *and*

- *Inheritable* Read Write AD

obtained when executable file ADs (Subsection 4.3.5) are split by the value of the non-inheritable flag of every AD Element contained. Elements of the *Non-Inheritable* AD do not take part in the *AD Inheritance Algorithm* (Subsection 4.3.8).

The reason for keeping *User ADs* and *Executable File ADs* separately within a process is put forward in Subsection 4.3.8.

The structure of a *Process AD* is illustrated in Figure 4.3.7.

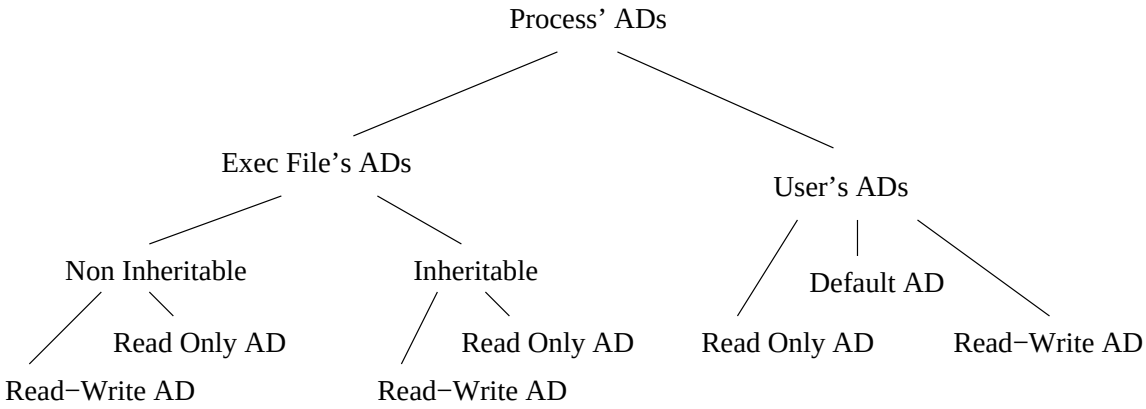

Figure 4.3: Process' AD structure

4.3.8 Access Domain Inheritance

There are two events in the lifetime of a process when its AD needs to be altered:

- When a process executes another binary, *or*
- When ownership of a process changes.

When an executable is invoked by a process, the process' AD needs to be modified to account for the AD of the newly executed file. The algorithm adopted for process AD recomputation, on the former event, is:

1. $inh_rw_acd^{\star} = inh_rw_acd + executable \rightarrow$inh_rw_acd
2. $ninh_rw_acd^{\star} = executable \rightarrow$ninh_rw_acd
3. $inh_ro_acd^{\star} = inh_ro_acd + executable \rightarrow$inh_ro_acd
4. $ninh_ro_acd^{\star} = executable \rightarrow$ninh_ro_acd

Where:

- $inh\_rw\_acd^{\star}, ninh\_rw\_acd^{\star}, inh\_ro\_acd^{\star}, ninh\_ro\_acd^{\star}$ are inheritable and non-inheritable read-write and read only ADs obtained after the execution of the algorithm respectively.
- executable→inh_rw_acd, executable→ninh_rw_acd, executable→inh_ro_acd, executable→inh_rw_acd are read-write and read only ADs associated with the executable file split into inheritable and non-inheritable portions by examining value of the inheritance flag of the constituent AD Elements respectively.
- *inh_ro_acd*, *inh_rw_acd* are Inheritable Read Only and Read-Write ADs of the current process before the execution of the algorithm.

As can be seen from the algorithm, Non-Inheritable ADs of the process involved are not taken into account when computing new ADs — they are just overwritten. This allows security policy administrator to decide on AD Elements associated with an executable that will be passed over to any other executable in the exec call chain (these AD Elements will not have their inheritance flag set to the Non-Inheritable value).

When ownership of a process changes the only thing that needs to be altered is User ADs associated with the process. This is simply accomplished by swapping User ADs of a user owning the process with User ADs of the new owner.

With reference to Subsection 4.3.7, if User ADs and Executable File ADs were not kept separately, in the context of a process, it would be extremely complicated to properly alter Process ADs on the change of ownership as it would be impossible to differentiate between AD Elements that originate from owner's ADs and the ones that were inherited from a, possibly very long, exec call chain. The chosen solution is a tradeoff between the complexity of the AD Inheritance algorithm and some overhead on performing access control checks. As is shown in Subsection 4.3.9, it is much more important to keep the inheritance algorithm simple.

4.3.9 File System Access Domain Access Control

LinSec groups all access types into two broad categories for the access control purposes:

- read access: read files, search directories, execute files, follow links etc., *and*
- write access: create, write, delete, move etc. files and directories.

In general, the AD access control algorithm works as follows: ADs of a process requesting access are checked for existence of the absolute path elements of the target file or directory in reverse order starting with the target file or directory. The algorithm has two possible outcomes:

- *Refuse access*: when none of the elements of the absolute path of the target file or directory exist in the process' ADs, including the file system root, or when an path element is found in the process' ADs but is marked as excluded from the AD.
- *Grant access*: if an absolute path element of the target file or directory was found in the process' ADs and the matching AD Element is not marked as excluded from the AD.

Granting access in context of *LinSec File System Access Domains* means that request is passed on to the lower level file system kernel code, as illustrated in the Figure 4.3.1.

In context of the example from Subsection 4.3.1: access to the file */etc/shadow* would be refused for the process as there exists a match denoting exclusion for the requested path (*/etc/shadow*) in the process' ADs. Both read and write access to eg. */usr/doc/faq/Linux/intro.html* would be granted as there exist a match, that is not exclusion, for */usr* element of the target path. Finally, write access to eg. */etc/inetd.conf* would be refused, even though there is a match for */etc* that is not an exclusion, as the AD where the match is found is denoted as *Read-Only*.

If an access of category *write* is requested the requesting process' ADs are checked as explained above in the following order:

1. User's Read Write AD
2. Executable Files' Non-Inheritable Read Write AD
3. Executable Files' Inheritable Read Write AD

If an access of category *read* is requested the check proceeds as if the access request were of the *write* category. If no hits are encountered in the process, the algorithm proceeds to examine:

1. User's Read Only AD
2. Executable Files' Non-Inheritable Read Only AD
3. Executable Files' Inheritable Read Only AD

N.B. User's Default AD is considered as a part of the User's Read-Write AD.

The order in which different ADs are checked provides scope for optimization depending on the nature of overall system use. The above scenario

assumes that most of the activity on the system is initiated by users and that is why User's ADs have precedence in the ordering. Simply, the probability of a hit in User's AD is high in the context. However, if, for example, a web server is considered, checking Executable Files' ADs first would yield better performance. Thus, the ordering in which the check proceeds should be configurable to ensure best performance.

The AD access control algorithm implements the *first match* policy. It is important to emphasize this point for the complete understanding of the resulting behavior. The behavior was illustrated in the previous example by the fact that on successful match of /etc/shadow against one of the process' AD entries the algorithm stopped further execution and access was refused immediately[8].

4.4 LinSec IP Labeling

4.4.1 Background

All remote host based attacks that take place in the Internet environment use existing network tools, clients and purposefully written software to penetrate remote hosts.

To aid the description the following example is used: Two hosts involved, A and B. Host A is a mail server running a buggy version of Sendmail MTA (Mail Transport Agent) which has a buffer overflow vulnerability. An attacker, operating from host B, has a script that connects to the Sendmail port and exploits the buffer overflow thus gaining the attacker root shell access to the remote machine, host A.

Obeying the principle of *least privilege*, mandatory security policy of an system should be able to restrict network connections to only the processes

[8]The behavior of AD access control algorithm is equivalent to the behavior implemented by network firewall rule matching algorithms.

that legitimately need them. In the above example, if such a policy was in place, the attacker's script would be denied the permission to establish connection with Sendmail on host A as the only software that needs to be able to communicate to remote MTAs are local MDA (Mail Delivery Agent) i.e. mail clients and local MTA used for relaying mail, if it exists. Furthermore, local MTA and MDA should not be allowed to establish network connections if destination port is different than 25 (mail exchange/delivery port) as they do not need the functionality for the correct operation. It is impossible to enable this sort of behavior by using the traditional firewall approach as the firewall software available can not be used to specify fine-grained enough policy which would make distinction between individual processes as needed. This is exactly where the *IP Labeling* model fits in, depicted in Figure 4.4.1.

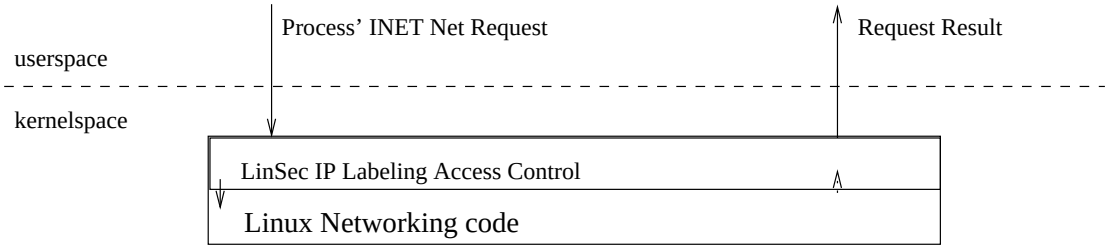

Figure 4.4: How LinSec IPL fits into Linux networking subsystem

Acronym *IPL* is used to mean *IP Labeling* in the rest of the text.

4.4.2 Overview

To obey the *Least Privilege* principle, in context of the network connections established by a process, each executable file in the system may be assigned a list of rules describing its allowed outgoing connections (Subsection 4.4.5). These rules are transformed into a Process IPL List (Subsection 4.4.6) once an executable is executed. In case of the Process IPL Lists there is no need

for inheritance across the execution chains (Subsection 4.4.7).

Current *LinSec* IPL design covers the *TCP/IP* set of protocols only. However, it should be possible for the principles to be applied to other transport level[9] protocols supported by *Linux*.

4.4.3 IPL Elements

LinSec IPL Lists are specified in terms of IPL Elements. Each IPL Element consists of an IP address, corresponding IP netmask, range of ports and a protocol identification. IPL Elements represent destination address and port for the specified TCP/IP protocol.

4.4.4 IPL Groups

Analogous to the idea of *Capability Groups* (Subsection 4.2.6), *Access Domain Groups* (Subsection 4.3.4) and *IP Chains* (standard *Linux* firewall software) IPL Groups consist of a number of IPL Elements. IPL Groups aid the ease of *LinSec* IPL configuration and maintenance.

IPL Group 0 is denoted the *default* IPL Group and has a special meaning as specified in Subsection 4.4.6.

4.4.5 Executable File IPL

With each executable file in the system a set of IPL Groups can be associated. This set forms, so called, *IPL List* describing in full network connections that a process running the executable can establish.

4.4.6 Process IPL

IPL List associated with any process in the system is simply the IPL List of the executable file the process is running the image of. If there is no IPL

[9] As in the OSI seven layer model.

List specified for the executable or the IPL List is empty the process' IPL List consists only of IPL Elements belonging to IPL Group 0.

4.4.7 IPL Inheritance

Effectively, there is no inheritance performed on IPL Lists. When a process executes another executable the existing IPL Lists are simply overwritten. The reason for this design decision was that no examples requiring the feature were found and, more importantly, that the IPL inheritance would open the possibility of attacks on the IPL model. An example of possible attack scenario would be causing a mail client, which obviously needs to have permission to connect to local and/or destination MTA, to execute a script via buffer overflow or similar. This would cause the script to inherit the mail client's IPL Lists which is undesirable.

4.4.8 IPL Access Control

IPL Access Control takes place when requested operation and specified INET protocol are one of the following:

- connection establishment, TCP
- connection establishment[10] or message sending, UDP
- message sending, RAW

Since TCP is connection oriented protocol no messages can be exchanged if the connection has not been established first. This is why it is enough

[10]Standard *Linux* network API provides the notion of a *connected* UDP socket. UDP socket connection is effectively only a kernel association between a connection id and a destination address. Programmers in user space use the network id, obtained when the connection is established, for sending UDP datagrams and the kernel ensures that the network id is properly matched to desired destination address.

to check IPL Lists only on connection establishment request for TCP. The same does not apply to UDP or RAW INET protocols.

In all of the above specified operation/protocol combinations network request specifies destination entity in terms of a IP address and, if the protocol is not RAW, destination port. IPL Access Control algorithm grants the request i.e. passes the request to the lower OS layer as specified in Figure 4.4.1 if an IPL Element is found in requesting process' IPL List that matches the following criteria:

1. ipl_element→protocol == request→protocol
2. ipl_element→ip_addr & ipl_element→netmask == request→destination_ip & ipl_element→netmask
3. If protocol used is not RAW,
   - (request→destination_port ≥ ipl_element→start_port) AND (request→destination_port ≤ ipl_elementi→end_port), if both ipl_element→start_port and ipl_element→end_port are defined, *or*
   - request→destination_port == ipl_element→start_port, if only ipl_element→start_port is defined.

Where:

- anything prefixed with *ipl_element* represents contents of the matching IPL Element, *and*
- anything prefixed with *request* represents network request's destination parameters.

Port range checking is not applicable for RAW protocol and that is why the last step of the access control algorithm is skipped if RAW protocol is used.

The algorithm, as such, implies default DENY_ALL policy as if no matching IPL Element is found the requested network operation is refused. This can be overridden, although it is not advisable, by putting IPL Element with IP addr 0.0.0.0 and netmask 0 in the, default, IPL Group 0.

4.5 Summary: LinSec Mandatory Security Policy

4.5.1 Overview

LinSec Capability model (Section 4.2), *File System Access Domains* (Section 4.3) and *IP Labeling* (Section 4.4) mechanisms form a very powerful mandatory security model when combined. This model, however, would be of little practical value unless substantial amount of flexibility for specifying mandatory security policy on top of it was provided.

In the following three subsections elements of *LinSec* mandatory security model that can be configured to yield overall system mandatory security policy are listed. The individual elements listed are grouped according to the sections they were defined in. For the explanation of *LinSec* specific terminology please refer to the appropriate sections of this chapter.

4.5.2 Capability Model

- User Capability Groups
- User Capabilities (uP, uB, User Capability Groups membership)
- Executable File Capabilities (fA, fF,fE and INET Socket Capabilities)
- Global Bounding Capability Set (gB)

4.5.3 File System Access Domains

- Access Domain Groups (in terms of Access Domain Elements)

- Executable File Access Domains (Read Only and Read-Write)
- User Access Domains (Read Only, Read-Write and Default User Access Domain)

4.5.4 IP Labeling

- IP Labeling Groups (in terms of IP Labeling Elements)
- Executable File IP Labeling List (in terms of IP Labeling Groups)

4.5.5 LinSec Mandatory Security Policy Specification

The flexibility provided by *LinSec* for specifying the mandatory security policy, in terms of the above listed elements, is immense. Policies can also be of arbitrary granularity ranging from the very coarse-grained ones that effectively mimic the traditional *Linux* behavior to extremely fine-grained ones that define different roles for each of the users and different privileges for every executable file. The functionality to implement the principle of *Least Privilege* exists, it is up to the system administrator to implement it.

LinSec mandatory security policy can be specified by any process that has CAP_LINSEC_ADMIN capability in its effective set. It should be emphasized that notion of the security policy administrator was not used in the previous sentence as the capability model dictates that access control is not based on user identities but on capability possession. One or more users in the system might be assigned CAP_LINSEC_ADMIN and act as security policy administrators.

Chapter 5

Implementation

5.1 Introduction

5.1.1 Chapter Contents

This Chapter covers the implementation of *LinSec*, as specified in Chapter 4, in the *Linux* kernel. Due to the space constraints on this report the implementation is presented on a rather high level with the exception of particularly interesting parts and problems encountered. Effort was made to explain implementation of the core *LinSec* mechanisms which is important for proving that the concepts specified in Chapter 4 are practically viable for employment in an mainstream operating system. For full details on the implementation a reference to the, well documented, source code is recommended.

5.1.2 Prerequisites

In order to be able to follow all the details in this Chapter, the reader is expected to be confident with C programming and operating system design. Some Linux and UNIX expertise is needed as well.

5.1.3 Implementation Overview

General Information

LinSec design, as specified in Chapter 4, was implemented fully in *Linux* kernel in approximately five thousand lines of C code. The implementation begun on *Linux* kernel version 2.4.15 and was ported onwards to the newer stable versions as they were published. *Extended Ext2 FS Attributes* (EA) kernel extension is used by *LinSec* implementation to enable association of special attributes with files as specified in the Subsection 5.3. At the time of writing this report, the current stable versions of *Linux* kernel and EA patch are 2.4.18 and 0.8.20 respectively.

Implementation approach

LinSec was developed in a modular fashion. Three main modules were identified:

- LinSec Capability Model
- LinSec File System Access Domains
- LinSec IP Labeling

Once implemented, each of the modules was thoroughly tested through real time use and behavior monitoring. Furthermore, each of the modules was built as a sequence of standalone increments. Effort was made to make increments as fine grained as possible to aid tracing of possible bugs through kernel. Acceptance criteria for the increments was that they could be configured in a way which would result in the modified *Linux* kernel produce the traditional behavior. Only when the entire modules were built was it possible to test the desired *LinSec* behavior.

Linux Kernel Subsystems Affected

LinSec implementation affected the following subsystems of the *Linux* kernel:

- File System (linux/fs), including *Ext2* (linux/fs/ext2) and *Ext3* (linux/fs/ext3) file systems.
- Networking System (linux/net), IP v4 in particular (linux/net/ipv4).
- Kernel Initialization System (linux/init).
- Kernel Core System (linux/kernel).

Endeavor was made to spread *LinSec* code across kernel as little as possible. Most of the *LinSec* code is contained within newly created files prefixed with linsec_ and stored in linux/kernel directory. Original kernel code was altered only to add calls to *LinSec* specific functions where needed. Furthermore, all of the *LinSec* code can be left out of the compiled kernel if *CONFIG_LINSEC* option is not chosen when selecting the compile options.

5.1.4 Chapter Layout

This Chapter proceeds in a way the actual implementation went with slight alterations for the reasons of clarity. For successful implementation a good understanding of *Linux* kernel was needed (5.2) in the first place. To provide flexibility in defining *LinSec* mandatory security policies, configuration data (5.3) and the process of communicating that data to *LinSec* (5.4) had to be specified. The choice of *LinSec* data structures and *Linux* data structures that would play major role in *LinSec* (5.5) was as important as the implementation of *LinSec* algorithms that manipulated the data (5.8, 5.9, 5.10, 5.11). The *LinSec* algorithms are triggered by the relevant existing *Linux* mechanisms (5.12). Finally, to setup and administer a proper *LinSec*

mandatory security policy a set of userspace administrative tools is needed (5.13).

5.2 Linux Kernel Analysis

With reference to the *LinSec* design (Chapter 4), four kernel mechanisms that would suffer most alterations were identified as:

- Access Control
- Executing a binary
- Changing ownership of a process
- Sending packets via INET protocols

Although level of understanding of these mechanisms was high on the theoretical side [14, 25, 6], some practical insight into up-to-date kernel was needed before the actual implementation started. The approach used was to place calls to *LinSec* functions where it was thought they should go, determined by reading the kernel source, but with the actual *LinSec* functions being dummies printing a text message via `printk`. This proved invaluable as it revealed some undocumented intricacies of the *Linux* kernel. Having the actual references to *LinSec* code in correct places in *Linux* kernel formed a framework for further implementation.

5.3 LinSec Configuration Data

LinSec Configuration Data is the data representing *LinSec* mandatory security policy for a system. It can be roughly divided into two groups:

1. data closely related to a particular executable file, eg. File Capabilities, File Access Domains etc., *and*

2. other, more general, data eg. Capability Groups, IP Labeling Groups etc.

To avoid penalties of frequent access to configuration files or occupying large chunks of kernel memory, data of the group 1 is kept in the disk blocks used by a particular executable file. The configuration data is stored in a way that enables it to be read together with the other file's data, avoiding the penalties of additional reads. Functionality that provides this is not in the original implementation of *Linux* kernel but is part of *Extended Ext2 & Ext3 Attributes* [15] kernel patch (add-on) used for the purpose.

The group 2 consists of (Chapter 4):

- Capability Groups configuration data
- Access Domain Groups configuration data
- Per User configuration data
- IP Labeling Groups configuration data

This data is kept in configuration files that are read during the system boot and stored in, *LinSec* implemented, kernel buffers. The data structure used for the *LinSec* kernel buffers is a chained hash table as it provides an average O(1) access time for locating an element. For each of the buffers (storing one of the above configuration data types) a set of functions for manipulation (retrieve, update, create and delete) of the data is provided. An important detail to emphasize is that any function that retrieves an element from one of the buffers returns a copy of the actual element and not the reference to it. This prevents various race conditions from occurring due to data sharing.

5.4 LinSec Configuration Process

LinSec mandatory security policy can be fully configured at runtime, without the need for system to be rebooted. In addition to storing configuration data on stable storage, as outlined in Section 5.3, the data can also be fed to the running kernel through /proc file system interface implemented by *LinSec*. The changes in configuration can, thus, have an immediate effect. In case of manipulation of *LinSec* user configuration data, privilege revocation for all of the processes owned by the user in question takes place atomically (Subsections 5.8.6, 5.9.4). *LinSec* /proc runtime configuration interface caters for creation, modification and deletion of any of the mandatory security policy elements specified in Section 4.5.

5.5 LinSec Data Structures

LinSec implementation defines many data structures most of which are obvious from Chapter 4 (eg. Capability Group, AD Group etc.) and detailed consideration of which is therefore omitted.

Two data structures, however, are worth mentioning explicitly as they take direct part in the enforcement of *LinSec* mandatory security policy and will, therefore, be referred to frequently in this Chapter. These structures and their main roles are:

- `linsec_usr` (linux/linsec_dt.h): holds per user configuration data such as capability sets, access domain elements etc. (refer to Chapter 4 for full set of items).

- `linsec_task` (linux/linsec_dt.h): generated runtime for each of the processes in a system, containing privileges of the executable file each of them is running (refer to Chapter 4 for a list of mandatory security policy data associated with executable files).

Both of the structures are referenced from the *Linux* per process structure `struct task_struct` as illustrated in Figure 5.5 (cardinality is indicated on the arrows).

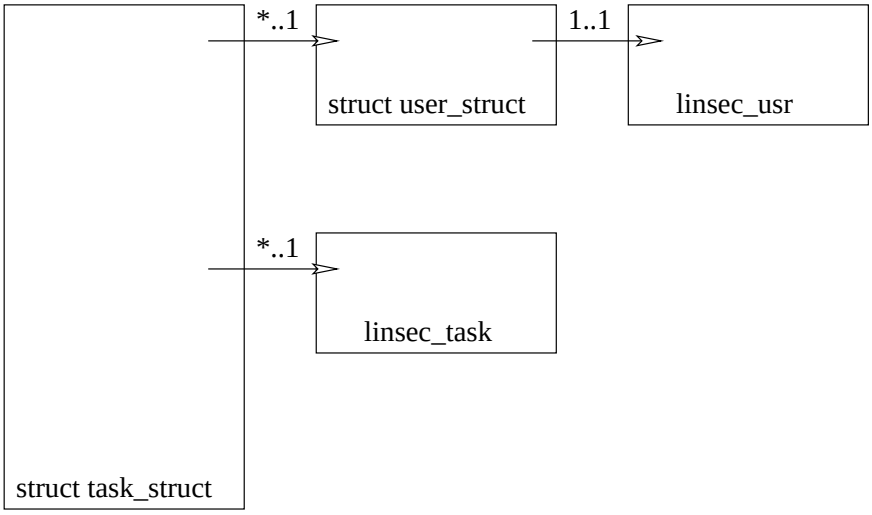

Figure 5.1: Linux - LinSec per process data struct relationships

Mapping of `struct task_struct` to `linsec_task` is *many to one* as the reference to `linsec_task` is copied (shared) on process *forking*[1]. To be able to free memory occupied by a `linsec_task` once it is no longer needed, the structure contains, among other elements, a reference count denoting the number of `struct task_struct` structures that are referencing it at any particular moment. The reference count is increased on a call to `do_fork` (kernel/fork.c) and it is decreased on a call to `release_task` (kernel/exit.c), when a process dies or on call to `do_execve` (fs/exec.c) i.e. when `linsec_task` for a process is replaced to reflect privileges of the newly executed binary, respectively. Once the reference count reaches 0 memory occupied by `linsec_task` is released by `linsec_cleanup_task` (kernel/linsec_misc.c).

`linsec_usr` structure is, as depicted in Figure 5.5, referenced from

[1] Creating new processes by calling the `fork` syscall.

`struct task_struct` indirectly via the reference to `struct user_struct`. `struct user_struct` is used in *Linux* for user accounting and is shared[2] among `struct task_struct` structures describing processes owned by a same user. Consequently, `linsec_usr` is shared in the same manner among `struct task_struct` structures. Unlike `linsec_task`, `linsec_usr` does not contain any reference counts as it is safe to release memory it occupies once the referencing `struct user_struct` is dismissed.

When a process changes its ownership, eg. by calling the `sys_setuid` function, the `struct user_struct` referenced by it is replaced with the `struct user_struct` corresponding to the new owner (uid) which in turn references `linsec_usr` for the new owner (uid).

5.6 SMP Issues

As of version 2.0, *Linux* kernel supports *Symmetric Multi Processing* (SMP) allowing processes to run in kernel mode in parallel on different processors. To avoid race conditions on shared kernel data, *Linux* kernel provides a set of SMP specific lock data types and corresponding locking primitives. All *LinSec* shared data structures are implemented in a SMP safe manner by making use of the *Linux* SMP features. In particular, to enable capability revocation (Subsection 5.8.6) and access domain revocation (Subsection

[2]The first implementation of `linsec_usr` actually contained a reference count field which was managed in the same sort of way as for `linsec_task` structure. This, however, did not reflect completely the shared nature of `struct user_struct` as the reference count represented only the number of processes sharing `linsec_usr` that belong to the same process creation subtree. It is possible for independently created processes to be owned by the same user and therefore share `struct user_struct` and corresponding `linsec_usr`. This omission was noticed when the code for releasing of the memory occupied by `linsec_usr` was implemented and tested. The kernel OOPS messages caused by *LinSec* code referencing *linsec_usr* through invalid pointers were extremely difficult to follow and it was very hard to locate the problem.

5.9.4), `linsec_usr` structure contains a set of, so called, *spinlocks*, that are used to protect relevant portions of the structure as they are being updated. There is no need for the similar mechanism to be implemented for `linsec_task` as its contents are only used for read operations once they are created.

5.7 LinSec Lifetime

LinSec lifetime, from system boot until system shutdown, can be divided into two phases:

1. Initialization: used to set up *LinSec* (read configuration files, initialize kernel buffers, etc.), *and*
2. Operation: *LinSec* mandatory security policy enforcement.

The *Initialization* phase is performed after main kernel subsystems have been set up and just before *init* executable is loaded (function `linsec_do_setup`, kernel/linsec_setup.c, called from function `init`, init/main.c). Thus, *LinSec* mandatory security policy enforcement starts from the very beginning of the userspace system boot phase.

5.8 LinSec Capability Model

5.8.1 Linux Legacy

Overview

Linux implements POSIX 1003.6 (Subsection 4.2.2) capability model to minimal possible extent in order to be able to claim compliance. No flexibility in terms of specifying any form of capability policy is provided. In fact, all of the relevant configuration details have been hardcoded in a manner which

ensures traditional *Linux* behavior[3]. Some of the elements of the capability model implementation are reused by *LinSec*, with substantial alterations, and some had to be excluded, as outlined in the rest of the section.

This section outlines how *LinSec* fits into the existing *Linux* capability model.

Capability Representation

Linux implements capabilities as values ranging from 0 to 31. The particular range is chosen for the easier representation in a 32-bit bitmaps where each capability occupies one bit. As already existing capabilities occupy values up to 28, *LinSec* implementation had to enlarge the possible maximum capability value to 63. This implied a change of bitmap size for capability set representation from 32 to 64 bits. The alterations were carried out in linux/capability.h header file.

Supporting Functions

Linux capability implementation also provides a basic set of functions and macros for capability and respective bitmap manipulation. These functions are used in *LinSec* but are adapted to support 64 bit wide data types.

The Task Structure

Linux standard definition of `struct task_struct` contains elements representing process' effective, permitted and inheritable capability sets as defined by POSIX 1003.6. These are of the type `kernel_cap_t` (linux/capability.h). *LinSec* did not alter these as changes made in linux/capability.h propagated automatically through the definition of `kernel_cap_t`.

[3]Mainly traditional DAC behavior.

User Capabilities

No support for configuring user capabilities exists in *Linux*. The values of capability sets used in the `struct task_struct` are hardcoded as:

- full capability set for *root* (uid 0), *and*
- empty capability set for "the rest of the world".

Inheritance Algorithm

Linux capability inheritance algorithm is presented in Subsection 4.2.9 and is implemented in the function `compute_creds` contained in the source file fs/exec.c. Effectively, the only thing it does, in the current *Linux* capability model implementation, is to ensure that root owned processes have all privileges and that processes owned by anyone else have none. *LinSec* implementation carried out substantial changes of the algorithm, as specified in the Subsection 4.2.9.

Hardcoded Mechanisms

To support the "patchwork" implementation of capabilities certain related mechanisms had to be hardcoded. Two, in particular, had to be disabled as they did not comply to *LinSec* mandatory security model:

- When a process dies all its children are reparented to *init* (function `reparent_to_init`, kernel/sched.c). In doing so, capabilities of the orphaned processes are raised to full. Thus, processes having no privileges initially can obtain them all if orphaned. Clearly an undesirable effect in a mandatory security environment.
- When a non-*root* user executes a suid binary capability sets are adopted to allow the process to perform actual change of ownership (function `prepare_binprm`, kernel/sys.c).

5.8.2 Executable File Capabilities

Executable File Capabilities are implemented as extended attributes (Section 5.3) of executable files. System administrator can manipulate the attributes as part of the *LinSec* mandatory security policy configuration process (Section 5.4). Executable File Capabilities are retrieved from extended attributes of a particular executable file, when they are needed, by the Capability Inheritance Algorithm (Subsection 5.8.4).

5.8.3 User Capabilities

User Capabilities are specified, by the system administrator, as part of general *LinSec* per user configuration and are stored in the `linsec_usr` structures.

5.8.4 Capability Inheritance Algorithm

Capability Inheritance Algorithm was implemented in full as specified in the Chapter 4 (Subsection 4.2.9). The inheritance algorithm (function `linsec_compute_creds`, linsec_exec.c) is invoked in two occasions:

- when a new executable is loaded by a process, to reflect the executable file's privileges (eg. when `exec` is called by an userspace program to run some other program), *and*
- when a process changes its ownership, to reflect privileges of the new owner (eg. when `setuid` function is called by a suid binary).

Therefore, a call to `linsec_compute_creds` is placed in `compute_creds` (fs/exec.c) function invoked from `do_execve` (fs/exec.c) and also in `linsec_do_suid` (kernel/linsec_suid.c) invoked from the family of `sys_setXuid` (kernel/sys.c) functions. `linsec_do_suid` also handles manipulation of `linsec_usr` (Section 5.5).

5.8.5 Linux Process Ownership Model Problem

In a traditional *Linux* approach all processes executed during the boot phase are owned by root (a user with uid 0).

This behavior is hardcoded in Linux kernel when the *init* binary is executed (definition of `INIT_TASK`, linux/sched.h). With the introduction of the mandatory security policy definition (Chapter 4) the notion of the *root* (uid 0) user, as all powerful, was lost and it became equal to any other user in a system. This creates a complication on boot time if the privileges of the *root*, as defined in *LinSec* security policy for the system, are too restrictive.

To overcome the problem of association of boot processes with root user (uid 0), *LinSec* hardcodes (`setup_init_task`, kernel/linsec_setup.c) a special instance of `linsec_usr` structure (Section 5.5) to which a reference is placed in `struct task_struct` (via `struct user_struct`) describing *init* task. Values in the `linsec_usr` structure are chosen so that after the Capability Inheritance Algorithm (Subsection 5.8.4) is executed only privilege settings associated with the executable file a particular process is running are reflected in the actual process' capability settings. This special `linsec_usr` settings are inherited by all children of *init* until the first call to *setuid* (Section 5.5) for each of them.

In this way, all programs executed during the boot phase of a system are effectively disassociated from users, as desired.

5.8.6 User Capability Revocation

When user capability sets are modified (through `linsec_usr`) and the modifications are fed to running kernel (through */proc* interface) two actions are performed by *LinSec*:

- appropriate `linsec_usr` structure in *LinSec* kernel buffer is updated, *and*

- `struct user_struct` for the affected user is located (`linsec_user_struct_find`, kernel/user.c) and, if it exists, the referenced `linsec_usr` is updated accordingly.

It might be the case that no process is currently running under ownership of the user whose settings have been changed so that corresponding `struct user_struct` does not exist in kernel buffers. In this case the second action is skipped. Update of `linsec_usr` capability related fields is done, in the latter event, while holding appropriate *spinlocks* (Section 5.6). The effects of the changes done to `linsec_usr` will be observable on the next execution of the Capability Inheritance Algorithm (Subsection 5.8.4) for the user.

5.8.7 Process Information Hiding

LinSec configurable mandatory security policy provides two ways of hiding information related to processes (Chapter 4, Subsection 4.2.11):

- hide all info about a process, *and*
- hide info about network connections of a process.

Functionality, for both of the options, is implemented by forcing kernel functions that provide the information, obtainable through */proc* file system, to omit the relevant data belonging to the hidden processes.

In the first case, it is enough to test whether the process to be listed has *CAP_PROC_HIDDEN* in its *effective* capability set. The `struct task_struct` for any process is easily located given the *pid* (process identifier). The functionality is implemented in function `get_pid_list` in fs/proc/base.c.

The second case, however, proved to be much more tricky to implement *Linux* kernel does not provide a mechanism to determine `struct task_struct` of a process that created a socket given `struct sock`

(net/sock.h) structure that describes the socket. It is possible to find process group owning a socket (receiving IO signals on the socket) but that is not good enough. Therefore, *LinSec* had to extend `struct sock` by adding a flag element that shows whether a socket is hidden or not, determined by the existence of the CAP_NET_HIDDEN capability in the effective capability set of the process that created the socket at the time of the creation (`sys_socket` function in net/socket.h had to be modified). After such a solution has been implemented, it was enough to extend appropriate functions in net/ipv4/raw.c, net/ipv4/udp.c and net/ipv4/tcp_ipv4 to filter network information they provide to the /proc interface based on the flag value.

5.9 LinSec Filesystem Access Domains

5.9.1 Executable File Access Domains

Executable File Access Domains are implemented as extended attributes (Section 5.3) of executable files. Each of the executable file ADs (Chapter 4), Subsection 4.3.5) is defined in terms of:

- AD Groups (Subsection 4.3.4) executable file is a member of, *and*
- AD Elements (Subsection 4.3.3) belonging to the AD Groups executable is a member of but with differing flag value.

Rationale for implementing the latter feature has to do with optimization issues. *LinSec* implementation allows a maximum of 64 AD Groups to be defined for the ease of implementation and for the performance gains of representing AD Groups and AD Group sets as 64 bit bitmaps. However, if new AD Groups had to be created whenever an executable requires an AD which differs from what can be obtained from one of the existing AD Groups just in value of the flag of one or more of constituent AD Elements, the maximum number of AD Groups would soon become a bottleneck. Therefore,

LinSec allows executable files to override the flag setting of a number of AD Elements belonging to AD Groups they are a member of by explicitly specifying AD Elements in question and storing them directly in extended attributes along with the AD Group membership information. For example, supposing that AD Group 1 contains an AD Element denoting that `/etc` directory can be accessed and that an executable file `foo` needs exactly the AD as described by AD Group 1 but with `/etc` excluded from it. Instead of defining a new AD Group for the purpose, system administrator can give `foo` a membership of AD Group 1 and, in addition, explicitly assign `foo` an AD Element representing `/etc` but with flag value denoting the exclusion.

5.9.2 Access Domain Representation

User Access Domains (Chapter 4, Subsection 4.3.6) and Process Access Domains (Subsection 4.3.7) are kept in `linsec_usr` and `linsec_task` structures (Section 5.5) respectively.

Process Access Domains are represented in `linsec_task` in terms of AD Elements rather than AD Groups for two main reasons:

- chained hashing, used for storing AD Elements, provides much better search performance than would traversing AD Group definitions, *and*
- AD inheritance would be much more complicated in terms of AD Groups with respect to possible variability in the value of flag field of AD Elements.

User AD settings are kept in `linsec_usr` both in terms of AD Groups and in terms of AD Elements. While `linsec_usr` is stored on stable storage (in config files) and while it is kept in *LinSec* kernel configuration buffers there is no need to represent User ADs in terms of AD Elements (except User Default R-W AD which is configured in terms of AD Elements) as no AD operations are performed on the structure as such. Once a copy of `linsec_usr` is

requested (`linsec_get_user` function, kernel/linsec_buffs.c), to be referenced from a `struct user_struct` (Section 5.5), AD Group representation of User ADs is expanded into corresponding AD Element representation which is stored in hash tables[4] of the `linsec_usr` copy returned.

5.9.3 Access Domain Inheritance

Access Domain Inheritance algorithm was implemented in full (`do_ad_inh` function, kernel/linsec_exec.c) according to the specification in Chapter 4 (Subsection 4.3.8). The inheritance algorithm is triggered by a new binary being loaded for execution (from `do_execve` function, fs/exec.c) to reflect privileges of the executable file in `linsec_task` structure. When ownership of a process changes (`sys_setXuid` family of functions, kernel/sys.c), there is no need to perform the inheritance algorithm explicitly as the replacement of `linsec_usr` itself (Section 5.5) achieves the desired shift in privileges for the process.

During testing it arose that no AD inheritance should take place if process ownership change occurred before a call to `do_execve` (fs/exec.c). For example, *shell* program should not inherit AD settings from *login* program that executed it. Therefore on call to one of `sys_setXuid` functions (kernel/sys.c), `linsec_do_suid`, among other things, sets a flag (implemented by *LinSec*) in `struct task_struct` to denote ownership change for the process. When Access Domain Inheritance algorithm executes it checks the flag for current process and if it is set no inheritance takes place and the flag is cleared.

5.9.4 User Access Domain Revocation

User Access Domain Revocation is completely analogous to the User Capability Revocation (Section 5.8.6) in its implementation with the following

[4] chained hashing based on inode number

slight differences:

- Changes to a `linsec_usr` structure referenced from a `struct user_struct` structure are being carried out while holding AD related spinlocks of the `linsec_usr`.
- Effects of the changes carried out on AD related fields in `linsec_usr` are observable immediately after the update process has finished. This is because AD access control checks take place directly on `linsec_usr` and, unlike for the capability related fields, no separate algorithm has to be executed to reflect the changes in process' privileges.

5.9.5 Access Domain Access Control

AD Access Control mechanism was implemented fully as specified in Chapter 4 (Subsection 4.3.9) in function `linsec_check_ad_perms` (kernel/linsec_access_control.c).

After the inspection of all file system related system calls provided by *Linux* kernel it was noted that traditional DAC for all of them is performed by the `permission` function (fs/namei.c). The logical choice would be to place a call to `linsec_check_ad_perms` in this function. The parameter to the `permission` function that denotes a file, for which access control check is to be performed, is of type `struct inode` (linux/fs.h). Unfortunately, `struct inode` does not provide sufficient information for the algorithm to be able to traverse the file's parent directories up to the file system root as required by the specification. Therefore, a call to `linsec_check_ad_perms` had to be placed in every single file system related system call just before a call to the `permission` function to obtain the desired behavior.

5.10 LinSec Socket Access Control

5.10.1 Socket Access Control Information Storage

Because of its "per executable file" nature, Socket Access Control (SAC) (Chapter 4, Subsection 4.2.12) information is stored in extended file system attributes of executable files. SAC is retrieved by `linsec_read_file_sac` function (linsec_attr.c) when an binary is being read into memory for execution (`do_execve`, fs/exec.c). Once in memory, SAC information is stored in `linsec_task` associated with the process that executed the binary (Section 5.5).

Due to the difficulty of matching `struct sock` (net/sock.h) with `struct task_struct` of the process that created the socket, as explained in Subsection 5.8.7 and affecting access control algorithm (Subsection 5.10.2), *LinSec* implementation was forced to extend `struct sock` by another element. The new field represents capabilities required to communicate to a socket and is initialized when the socket is being bound (`sys_bind` function, net/socket.c) according to the info contained in `linsec_task` of the process creating the socket. If no SAC entry is defined for a socket being bound, the required capability set is initialized to be empty.

5.10.2 Socket Access Control Algorithm

SAC algorithm is triggered by four events:

- a connection request using TCP (`tcp_v4_connect`, net/ipv4/tcp_ipv4.c),
- a connection request using UDP[5] (`udp_connect`, net/ipv4/udp.c),
- a packet send request using UDP (`udp_sendmsg`), *and*
- a packet send request using RAW IP (`raw_sendmsg`, net/ipv4/raw.c).

[5]As explained in the Section 4.2.

The algorithm proceeds in the following steps:

1. Determine whether the destination IP address is local (`linsec_is_addr_local_v4`, kernel/linsec_ipc.c). This is accomplished by traversing the list of registered network devices and comparing IP addresses bound to each one with the requested destination IP.

2. Match the destination port number to a `struct sock` examining various protocol dependant hash tables. The matching is based on the destination port number, network device id and the protocol used.

3. Check whether the process that requested the service contains capabilities required by the socket in its effective capability set.

Only if the last step of the above algorithm fails, i.e. if the process requesting the service is not allowed to communicate to the desired socket, is the service forced to fail. Otherwise the algorithm exits and allows standard *Linux* networking code to continue servicing a request.

5.11 LinSec IP Labeling

5.11.1 IP Labeling Information Storage

As IP Labeling (IPL) (Chapter 4, Section 4.4) information is, by its definition, closely related to executable files it is, therefore, kept in extended file system attributes of executable files. IPL lists of an executable file are described in terms of IPL Groups file is a member of. In implementation terms, extended attribute describing IPL list of an executable file just holds a 64 bit bitmap in which each IPL Group is represented by a single bit. IPL information is retrieved from extended attributes by `linsec_read_file_ipl` (kernel/linsec_attr.c) when the executable is loaded in memory for execution

(do_execve function, fs/exec.c). Once read, the bitmap is transformed into a linked list of IPL Elements that is stored in linsec_task structure of the process that called do_execve.

5.11.2 IP Labeling Access Control Algorithm

IPL Access Control Algorithm is implemented in linsec_check_ipl function (kernel/linsec_access_control.c) as specified in Chapter 4 (Subsection 4.4.8). The algorithm is triggered by the same set of events that cause SAC (Subsection 5.10.2) algorithm to execute. A call to linsec_check_ipl is, however, placed before a call to SAC algorithm, in the affected network subsystem functions, to avoid the costly invocation of the SAC algorithm if much faster IPL algorithm fails the service request.

5.12 Exec and Setuid

LinSec implementation, as outlined in this Chapter, heavily depends on two kernel mechanisms to trigger *LinSec* specific process' privilege recomputation mechanisms. As the mechanisms play the central role in *LinSec* and they were referred to throughout the Chapter it is worth summarizing the actions they trigger for clarity. The mechanisms, their role within *LinSec* and the *LinSec* actions they trigger are:

- Executing a binary (do_execve function, fs/exec.c) - a trigger for process' privilege recomputation to reflect the privileges of newly executed binary (linsec_do_exec, kernel/linsec_exec.c). *LinSec* actions performed on the event:

  1. recompute process' capabilities, linsec_compute_creds function (kernel/linsec_exec.c),
  2. allocate new linsec_task structure for the process,

3. read SAC settings from file's extended attributes into the `linsec_task`, `linsec_read_file_sac` (kernel/linsec_attr.c),
4. read IPL settings from file's extended attributes into the `linsec_task`, `linsec_read_file_ipl` (kernel/linsec_attr.c),
5. if *suid* flag is not set in old `linsec_task` perform AD inheritance, `do_ad_inh` (kernel/linsec_exec.c),
6. read AD settings from file's extended attributes into the `linsec_task`, `linsec_read_file_adgrps` (kernel/linsec_attr.c) and `expand_ad_gid` (kernel/linsec_exec.c),
7. clean up old `linsec_task`, `linsec_cleanup_task` (kernel/linsec_misc.c),
8. replace old `linsec_task` with the new one.

- Changing ownership of a process (`sys_setXuid` family of functions, kernel/sys.c) - a trigger for process' privilege recomputation to reflect privileges of the new owner (`linsec_do_suid`, kernel/linsec_suid.c). *LinSec* actions performed:

  1. set *suid* flag in current's `linsec_task` to denote that process ownership switch took place,
  2. recompute process' capabilities according to the new user's capability settings and the capability settings of the binary whose image the process is running.

  Reference from `linsec_task` structure, describing the process changing the owner, to the `linsec_usr` structure, containing *LinSec* specific information about the new owner, is updated prior to execution of the above algorithm. As `linsec_usr` is referenced indirectly via `struct user_struct` (Section 5.5), the update of `struct user_struct` (function `set_user`, kernel/sys.c) is at the same time the update of the

`linsec_usr`. However, if it happens that `struct user_struct` for the particular user does not exist in kernel buffers at the time (i.e. no process in the system are currently owned by the user), a new one is created by the `set_user` function and appropriate `linsec_usr` is requested from *LinSec* via `linsec_get_user` function. If no `linsec_usr` for the user exists in *LinSec* kernel buffers a new one is created, initialized to default and stored in the buffers for future reference.

5.13 Userspace Administrative Tools

For *LinSec* to represent a fully functional package a complete set of userspace tools to manage the mandatory security policy has to be provided. Due to the time constraints, *LinSec* userspace tools have been regarded as an external element from the project onset. However, *LinSec* provides interface to userspace via config files and */proc* file system (Section 5.3). The interface implemented is complete and provides functionality for manipulating any of the aspects of *LinSec* mandatory security policy specified in Chapter 4, Section 4.5. The exclusion of the userspace tools from the project does not hamper its academic value but it severely impacts *LinSec* acceptance in the real world. As one of the aims of *LinSec* was that it should represent a practical system the decision was made to allow external development of the tools according to the strict interface specification. Userspace tools were, therefore, implemented by Mr Bosko Radivojevic and Mr Veselin Mijuskovic at the Computer Centre, Faculty of Electrical Engineering, University of Belgrade.

5.14 Summary

LinSec implementation represents a complete realization of the design as specified in Chapter 4.

In as much as the design itself is clear and the concepts are well understood, the implementation in the *Linux* kernel proved very challenging. The difficulties encountered were largely due to the sheer size of the *Linux* kernel source code (in excess of 2M lines of code) contributed to by hundreds of developers worldwide. From the very readable, easy to follow, largely MINIX source code at the very beginning, *Linux* kernel has evolved to a very intricate set of optimizations, code interdependencies and side effects which are anything but obvious or easy to follow. Not only did the *LinSec* implementation have to adapt to the *Linux* kernel but some of the *Linux* kernel mechanisms had to be altered to avoid clashes of methodologies (mainly DAC vs. MAC issues). Combining *LinSec* design with an mature operating system has shown to be equally challenging to devising the design itself.

Chapter 6

Testing

6.1 Introduction

Due to the fact that *LinSec* was implemented as a *Linux* kernel add-on (*patch*) and is deeply embedded in it there is no way of running *LinSec* code separately for testing purposes. Therefore, application of the standard testing procedures found in software engineering books was simply not possible for this project. *LinSec* was tested in a way that fits its nature, as described in the rest of the chapter.

Figure 6.1 is a pictorial representation of the testing process and criteria and the way it fits the overall structure of *LinSec* implementation. It is further discussed in the following sections.

6.2 Test Criteria

Two main testing criteria were used for accepting or rejecting any tested code:

1. Ability of the tested code, being embedded into the *Linux* kernel and running within, to reproduce the traditional *Linux* behavior. This criteria was used to ensure the correctness of implementation of the

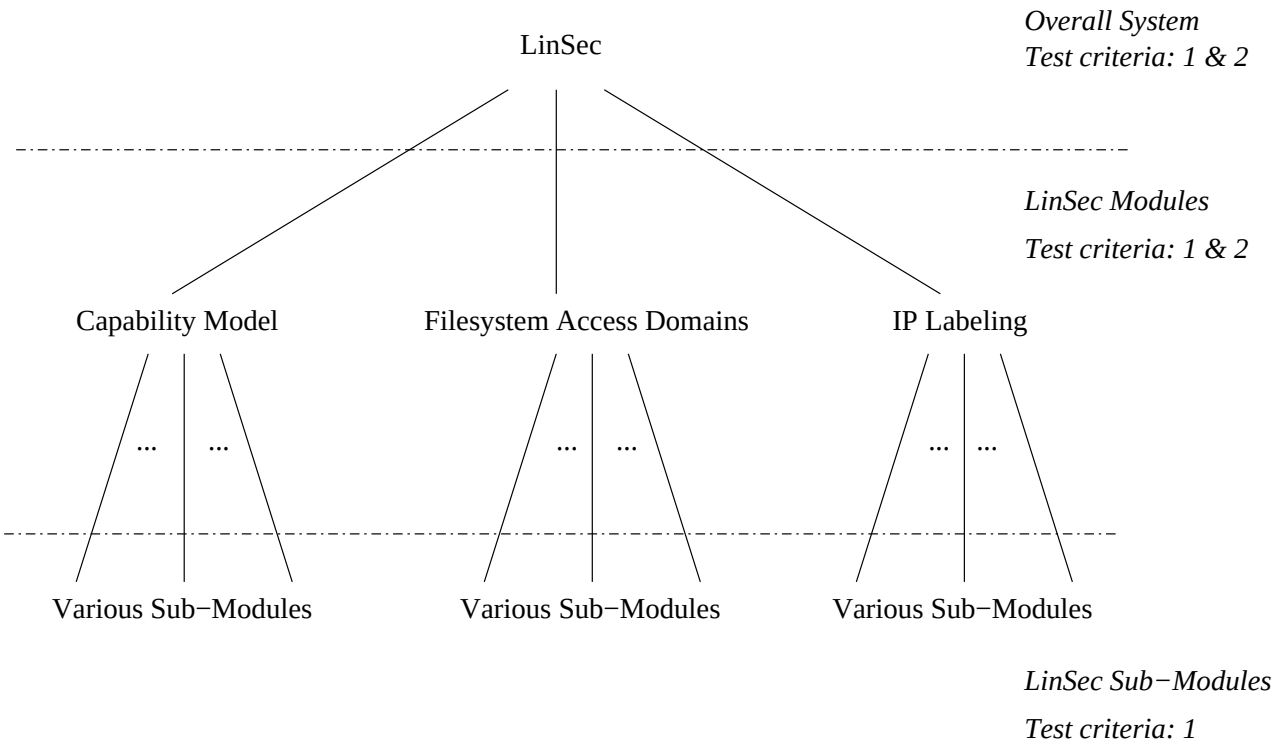

Figure 6.1: Approach to Testing

particular bit of *LinSec* code, both in terms of error free coding and in terms of the correct interaction with existing *Linux* mechanisms.

2. Ability of the tested code, running within the *Linux* kernel, to produce the required *LinSec* specific behavior as specified in the design (Chapter 4). This criteria was used to ensure the correct *LinSec* specific operation of the tested bit of code.

6.3 Test Process

LinSec was developed in a modular fashion as depicted by the tree in Figure 6.1. Firstly, it was subdivided into a set of modules that could exhibit autonomous operation (middle layer in Figure 6.1) and then these mod-

ules were subdivided into a set of comprising submodules that implemented various parts of the overall module functionality (lower layer in Figure 6.1).

As they were implemented, each of the submodules was tested according to the first test criteria (Section 6.2). The reason the second criteria was not used in this case is that the submodules themselves were not able of producing *LinSec* specific behavior on their own.

Once all of the submodules of a larger *LinSec* module were implemented and tested on their own the module as a whole was tested. In this case, both of the test criteria were used to ensure the correct interaction of submodules among themselves and with the rest of the *Linux* kernel and also to test the required *LinSec* specific behavior of the module.

Per module *LinSec* behavior included, among other:

- Capability module: configuring user (including capability groups) and executable files' capabilities and ensuring the correctness of the capability inheritance algorithm by observing the privileges of the running processes. Distributing roles across users and disempowering root. Testing capability revocation.

- Filesystem Access Domains module: configuring user and executable files' access domains (through the usage of the access domain groups) and observing contents of access domains belonging to the processes in the system to ensure the correctness of the AD inheritance algorithm and other related mechanisms. Testing effects of various user and executable files' access domain configurations. Testing access domain revocation.

- IP Labeling module: configuring IP labeling groups, associating them to executable files and testing network connectivity of the corresponding processes. Testing IP Labeling revocation.

Finally, when all of the modules were developed and tested, *LinSec* as a

whole was tested. Again, both of the criteria were used to ensure the correct intra module interaction, stable operation of the modified *Linux* kernel and the desired *LinSec* specific behavior of the overall system.

6.4 Test Environment

LinSec testing was carried out in two distinct environments:

- On a dedicated test machine with no other functionality which provided a strictly controlled environment, *and*
- On several *Linux* systems in Computer Centre, Faculty of EE, University of Belgrade, which have the role of secondary DNS, FTP and WWW servers in the centre. This represented a real world, hectic environment.

Testing of submodules and the initial *LinSec* testing was carried out solely in the former environment. Modules as a whole, providing some observable *LinSec* functionality, were tested in both environments. The dedicated testing machine was also used for reproducing and tracing of any bugs discovered during the testing.

6.5 Summary

The approach used for testing proved to be fit for the particular nature of the project. The testing itself yielded valuable feedback during the implementation. Careful analysis of the problem and *Linux* kernel source code at the beginning and thorough design resulted in no conceptual errors and major bugs or system deficiencies being introduced during the implementation phase. Several issues, described in Chapter 5, arose due to the intricacies in the *Linux* kernel code, some of which took days to identify. Most of the "ordinary" programming omissions and bugs were discovered in real world

use, at the Faculty of EE, University of Belgrade. All of these were easily rectified.

Chapter 7

LinSec Benchmarking

7.1 Introduction

LinSec, although functional, would be of little practical value if its performance had a substantial impact on the overall performance of a host system.

This Chapter describes benchmarks carried out to assess the performance related aspect of *LinSec* as well as the obtained results.

7.2 Benchmark Target

From *LinSec* design (Chapter 4) and implementation (Chapter 5) it can be observed that *Linux* kernel mechanisms most affected by the alterations are:

- executing a binary (`do_execve` function),
- changing ownership of a process (`setXuid` family of functions), *and*
- file system access control.

It can, thus, be expected that these mechanisms bare most of the performance degradation.

Other *Linux* kernel mechanisms, like eg. INET networking code, are also affected but to a much lesser extent and as such are not considered in the performed series of benchmarks.

Furthermore, process ownership change mechanism are not bechmarked as well since:

- As can be seen from Section 5.12, *LinSec* mechanisms triggered by process ownership change are largely subset of the mechanisms triggered by `do_execve`.
- Calls to `setXuid` family of functions are very rare compared to the number of invocations of the other two mechanisms. Moreover, by introduction of mandatory security policy it is reasonable to expect that `setXuid` functionality will become obsolete.

Therefore, benchmarking performed concentrates on assessing performance degradation of `do_execve` and the fs access control mechanisms.

7.3 Benchmark Structure

Most of the COTS[1] benchmark suites available for *UNIX* platforms and *Linux* in particular target issues such as:

- memory latency and bandwidth,
- I/O device latency and bandwidth,
- context switching,
- system call overhead,
- processor performance, etc.

[1] Common Off The Shelf.

These tests do not target the desired kernel mechanisms and are unsuitable for benchmarking *LinSec*.

It transpired that kernel compilation task is fit for the purpose. Each time *Linux* kernel is compiled `gcc` is executed on several thousand source files (actual number depends on the kernel configuration in place). Every new invocation of `gcc` means creating a new process and executing the binary via `do_execve`. In addition, every instance of `gcc` reads one or more source files and writes an object file thus triggering the fs access control mechanism several times per run.

The approach to benchmarking using the kernel compilation has three steps:

1. perform multiple kernel compilations on both platforms running *LinSec* and on platforms running *clean Linux* kernel,

2. use standard *UNIX* `time` utility to measure time spent in kernel space consumed by each of the compilations, *and*

3. produce benchmark result by averaging and comparing the observed kernel time consumed by kernel compilations on both platforms.

One aspect of *LinSec* setup used for the benchmarks having an explicit effect on the results is:

- Only one AD Group exists, the default one, containing only file system root as an element. This causes AD Access Control algorithm to exhibit worst case performance as it needs to iterate a number of times that is equivalent to the depth in the file system tree branch of the target file to find a match (file system root in this case). Existence of multiple AD Groups would increase the time needed for process creation linearly with the number of elements each AD Group, a process is member of, contains. This is considered negligible as it is

performed only when first process owned by a user is created in the system (`linsec_usr` being retrieved from *LinSec* buffers) and when an executable is loaded to be executed.

Different possible capability related configurations do not influence benchmark results as, irrespective of the configuration, the same set of operations is always performed on them. An exception to this is reflecting capability group settings in a permitted capability set of a process whose owner is a member of the capability groups. However, analogous to the above AD case, the time taken for this operation is negligible.

7.4 Benchmark Environment

The benchmarking was performed on two different hardware platforms:

1. Host A:
   - Dual Pentium III 1GHz
   - L1 I & D cache: 16KB
   - L2 cache: 256KB
   - 1GB RAM
   - 2x Adaptec AIC7899 Ultra 160 with 4x IBM 18GB HDD
2. Host B:
   - Pentium Pro 200 MHz
   - L1 I & D cache: 8KB
   - L2 cache: 256KB
   - 96MB RAM
   - IDE, HDD 2Gb ST32140A

7.5 Benchmark Results

7.5.1 Host A

The results obtained on Host A are the following (all values are in seconds):

	LinSec	*Vanilla*
1.	13.060	13.110
2.	12.300	12.030
3.	12.760	12.510
4.	12.300	12.720
5.	12.270	12.240
6.	12.970	12.710
7.	12.820	12.230
8.	12.370	12.190
9.	12.200	12.110
10.	12.660	11.990
AVG	*12.517*	*12.303*

N.B. The AVG (average) value excludes the first measurement as it is almost certainly caused by cache related penalties.

Therefore, on Host A, kernel running *LinSec* is approximately 1.7% slower than the clean kernel.

7.5.2 Host B

The results obtained on Host B are the following (all values are in seconds):

N.B. The AVG (average) value excludes the first measurement as it is almost certainly related to cache related penalties.

In this case no *LinSec* related performance penalty can be observed from the data. This represents a very interesting result that suggests that on Host B, being a much weaker hardware platform, performance bottlenecks exist

	LinSec	*Vanilla*
1.	42.160	41.950
2.	41.020	42.030
3.	41.530	42.000
4.	41.830	41.430
5.	41.120	41.470
6.	41.350	41.330
7.	41.620	41.400
8.	41.700	41.380
9.	40.710	40.360
10.	41.370	42.090
AVG	*41.361*	*41.61*

that mask out the impact of *LinSec* code on overall system performance.

7.6 Conclusion

The scope of the performance assessment carried out was limited both in terms of the benchmark structure, due to the unavailability of appropriate benchmark suites, and in terms of the hardware platforms and functional environments covered.

The approach taken has both pros and cons:

- Pro: *Linux* kernel compilation represents a mix of activities that can be found in a heavily loaded system. Furthermore, kernel compilation targets two *LinSec* mechanisms expected to cause most performance degradation, executing a binary and file system access control. Therefore, the results obtained should be representative of *LinSec* performance in real world applications.

- Con: The benchmark performed is too coarse grained to precisely show

the actual performance degradation of the affected kernel mechanisms solely. A Custom made benchmark suite would certainly provide valuable detailed information that might pinpoint *LinSec* mechanisms that need optimization.

Overall, the benchmark demonstrated that *LinSec* implementation did not introduce substantial performance degradation into *Linux* kernel. The results obtained are favorable and show that a mandatory security mechanisms can be implemented in *Linux* kernel without affecting the system usability. It is, however, expected that further testing, both using a purposefully developed benchmark suite and gathered as feedback from the Open Source community, will prompt optimization and redevelopment of parts of the *LinSec* code.

Chapter 8

Conclusion

8.1 Project Summary

LinSec was envisaged as a system that would provide a mandatory security policy based access control in *Linux* as opposed to the traditionally implemented discretionary access control mechanisms. The inspiration being continuing failures of *DAC* in providing reliable security.

The concepts and ideas used for *LinSec* design are not new to the field but they were never combined together in a way done by *LinSec*. Mandatory security policy introduced by *LinSec* is based on *Capabilities* and *File System Access Domains*.

LinSec was fully implemented in approximately 3 months and in excess of 5,000 lines of kernel code. The implementation of the design proved tricky in an existing, highly developed, mainstream operating system like *Linux* is. A lot of care had to be taken of the side effects of any changes made to *Linux* kernel code as well as of the clash of methodologies. Several *Linux* kernel mechanisms had to be adapted to support the notion of mandatory security. *LinSec* was implemented in a way that provides as transparent as possible transition to it and can be employed safely in all existing *Linux* systems.

Performance benchmarks carried out on *LinSec* have shown minimal

overhead introduced. Minimal being less than 2% in worst benchmarked case.

Testing of *LinSec* was carried out by the author at UCL and also at the Computer Centre, Faculty of EE, University of Belgrade. No conceptual mistakes were encountered throughout the implementation and testing phase, owing to the careful and detailed design. Several "ordinary" programming bugs were discovered and were easily rectified.

Overall, *LinSec* project has fulfilled its initial requirements. It represents a fully functional *Linux* kernel patch which, when applied, enables a significant increase in system security through the highly flexible and configurable mandatory security policy provided. Furthermore, *LinSec* is released to the Open Source community via the web site *www.linsec.org* (still under construction at the time of writing this report) and under the terms and conditions of GNU/GPL license.

8.2 LinSec Future

LinSec, although functional, because of its evolving nature, is not finished and will probably never be. Further development shall be based on the feedback obtained from the Open Source community and future expansion of the areas *LinSec* is built on. Major directions in which *LinSec* will evolve are roughly:

- Extension of existing *LinSec* features or introduction of new ones in order for *LinSec* to be able to:

    1. Respond to new security challenges: In particular, *LinSec* currently lacks sophisticated network security mechanisms provided by most network intrusion detection systems. Also, the existing *LinSec* mechanisms have to evolve as existing exploits evolve.

2. Fit better in the environments it is used in: For example, one of the interesting ideas is to expand the existing capability model to fit distributed systems eg. to provide capability based means of defining privileges in a distributed computing environments. Embedded *Linux* also represents an interesting challenge for *LinSec*.

- Natural evolution with the *Linux* kernel.
- Various optimizations of *LinSec* mechanisms, as outlined in chapters 6 and 7.

However, it is difficult to predict all possible requirements that will arise as a consequence of the way *LinSec* will hopefully be used in. Open Source world is full of surprises and the future of *LinSec* depends on its ability to overcome thus imposed challenges.

8.3 Project Scope

Scope of the *LinSec* project far exceeds the undergraduate curricula at the Department of Computer Science, UCL, in almost all aspects that were required for its successful completion, namely:

- C programming,
- Operating Systems theory and practice,
- Linux OS *and*
- Computer Security.

Although the author had strong background in these areas, large proportion of the knowledge required has been acquired during the project realization from the literature, various online sources and practice. Furthermore, the fact that very few projects exist that address the same problem

in *Linux* added a rather strong research component to the *LinSec* project. This is regarded as invaluable experience.

The *Linux* related background knowledge came largely from, in addition to the sources listed in the bibliography, various *Linux* kernel development forums and mailing lists such as: the *linux-kernel* mailing list, the *Linux Security Module* (LSM) mailing list, the *kernel-newbies* mailing list, etc.

Appendix A

Systems Manual

A.1 Introduction

The Systems Manual provides instructions on how to include *LinSec* code, provided on the floppy disk accompanying this report, into the *Linux* kernel source code and how to compile the extended kernel.

The manual does not cover installation and setup of *Linux* nor any of the needed administrative actions. Anyone wishing to use *LinSec* should be familiar with the *Linux* system administration.

All of the steps in the described procedure need to be performed as *root* user.

A.2 Software Requirements

- Any *Linux* desktop/server distribution, installed and running.
- *Linux* kernel source code, version 2.4.17.
- GNU, or compatible, C compiler.
- standard *Linux/UNIX patch* tool.

All of the above software is included on the standard *Linux* distributions.

A.3 Step I - Patching the Kernel

1. Copy the *LinSec* patch (linsec.patch), from the floppy disk, into the top kernel source code directory, eg. `mcopy a:linsec.patch /usr/src/linux`.

2. Change the current working directory to the top kernel source directory, eg. `cd /usr/src/linux`.

3. Include *LinSec* source into the kernel source code tree using the *patch* tool: `patch -p1 < linsec.patch`.

LinSec code is now included into the *Linux* kernel source code tree. Most of the code can be found in linsec_*.c files in the kernel kernel subdirectory.

A.4 Step II - Configuring and Compiling the Kernel

1. Start the kernel configuration process in a preferred way (either by executing `make config` or by executing `make menuconfig` in the top kernel source directory).

2. In the *Code Maturity Level Options* choose the following options:

   - Prompt for development and/or incomplete code/drivers.

3. In the *File Systems* section choose the following options:

   - Extended filesystem attributes.
   - Extended user attributes.
   - Extended attributes for ext2 (if using ext2).
   - Extended attribute block sharing for ext2 (if using ext2).
   - Extended attributes for ext3 (if using ext3).

4. In the *LINSEC* section choose the following options:

   - LINSEC support.

5. Compile the kernel in the preferred way (eg. by executing `make dep; make clean; make bzImage`).

A.5 Step III - Installing and Running the Kernel

To install the precompiled kernel with *LinSec* support use the standard preferred procedure.

In order to run the kernel some initial *LinSec* specific configuration has to be performed. This is a laborious manual process and tools for automating it are being developed at the time of writing this report. Therefore, no details regarding running the modified *Linux* kernel are included in this version of the Systems Manual.

N.B. The configuration tools were not regarded as a part of the final year project. They are, however, part of the overall *LinSec* project.

Appendix B

Users Manual

The Users Manual should include instructions on how to use *LinSec* userspace tools to configure system's mandatory security policy. However, as userspace administrative tools were not regarded as part of the final year project and as they have been developed externally, there is no scope for details on them in this report. Overall *LinSec* documentation, as released to the Open Source community at the site www.linsec.org (still under construction at the time of writing this report), will include full users manual, systems manual and the source code.

Appendix C

November Project Plan

1.1 Student Name:

Boris Dragovic (bdragovic@cs.ucl.ac.uk)

1.2 Project Title:

LinSec - Linux security protection and intrusion detection system

1.3 Supervisor:

Graham Knight

1.4 External Supervisor:

Prof Jon Crowcroft, Cambridge University Computer Laboratory

2.1 Project Aim:

The main project aims are introduction of Mandatory Access Controls

(MAC) to Linux operating system as opposed to the currently implemented Discretionary Access Controls (DAC), provision of mechanisms for enabling fine grain user role division in the system and eliminating the notion of all powerful root user.

2.2 Project Objectives:

There are two main mechainsms that need to be built into Linux to enable the fullfillment of the above specified aims: capabilities and filesystem access domains.

2.2.1 Capabilities

As of the version 2.2.0 Linux supports POSIX capability model but only partially. A portion of access control using capabilities has been implemented as well as capability bounding set but unfortunately no fs support is available yet which results in still having allpowerfull root user and the rest of the world. The current implementation is only an incomplete framework and needs to be heavily extended.

2.2.2 Process Capabilities

The implementation of POSIX capabilities defines three bitmapped sets of capabilities (in the task struct) for each process:

- Inheritable set (pI): set of capabilities that will be passed over to children processes spawned using exec() call.

- Permitted set (pP): set of capabilities that a process can

acquire during its lifetime.

- Effective set (pE): set of capabilities that are currently
used for access controll.

2.2.3 File Capabilities

Every executable file can have three sets of capabilities:

- Allowed set (fA): set of capabilities that an executable can
inherit from the parent process when turned into a new process
by exec().

- Forced set (fF): set of capabilities that the new process must
contain in its permitted/effective set after exec().

- Effective set (fE): set of capabilities that will be copied
from the permitted set of the new process to its effective set.

If any of the sets haven't been defined for the file the default is
used.

2.2.4.1 User Capabilities

Each user in the system has two capability sets associated:

- User permitted (uP): this capability set is added to the
pP of a new process (created by the user, under his uid)
thus enabling each user to have some extra rights on the system.

- User bounding (uB): capability set representing the maximum
set of capabilities any process of specific user can ever reach.

If the uP and uB haven't been specified for a certain uid the default is used.

2.2.4.2 User capability groups

Capability groups represent an idea analogue to user groups in standard Unix in terms of capabilities. Each capability group has a set of capabilities associated with it. A user can be in one or more capability groups at a time (at least in the default one). Capabilities possesed by the capability group a user is member of become part of user's permitted capability set according to the following algorithm:

```
uP' = uP | gxP | gyP | ... | gzP;
```

where uP' is resulting permitted capability set, uP is users individual permitted set and g[n]P are capability sets of capability groups user is member of. uP' is computed when capability inheritance algorithm is executed.

2.2.5 Global Capabilities

Each system has a global capability bounding set (gB) that represents the maximum capability set any of the process on the system can ever acquire.

2.2.6 New capabilities

In order to reach maximal flexibility and usability of the POSIX capability model, LinSec will implement several new capabilities:

- CAP_PROC_PROTECTED: process that has this capability will not receive any signals unless the sending process has CAP_PROC_GOD capability and the usual rights to send signals.

- CAP_PROC_UNKILLABLE: same as above but applies only for signals 2, 3, 9 and 15.

- CAP_PROC_GOD: process that has this capability can send signals to any process that has CAP_PROC_PROTECTED or CAP_PROC_UNKILLABLE if it has usual rights (same uid or CAP_KILL capability).

- CAP_PROC_HIDDEN: process having this capability is not to be listed in /proc.

- CAP_NET_HIDDEN: network connection info of the process with this capability are not listed in /proc.

- CAP_SYS_BOOTTIME: process with this capability may execute during system booting.

- CAP_MOD_CAP: process with this capability may modify its own permitted capability set.

- CAP_ACD_OVERRIDE: process with this capability may overide its access domain and access files outside it.

- CAP_MOD_ROUTE: process with this capability may modify kernel routing table.

- CAP_MOD_FW: process with this capability may modify kernel firewall rules.

This list is expected to enlarge as critical regions that must be protected in this way are identified.

2.2.7 Algorithm for computing capabilities of a new process

Capability sets of the newly created process are computed on the execution of exec() sys call according to the following algorithm:

1) pI' = pI
2) pP' = (fF | (fA & (pI' | uP))) & uB & gB
3) pE' = pP' & fE

where:
pP - permitted set of the parent process
pI - inheritable set of the parent process
pE - effective set of the parent process
' appended to the end of the above names denotes that the respective set belongs to the new process eg. pP' is the permitted set of the new process
fA, fF, fE - file allowed/forced/effective capability sets

The order of the above computations is important and should not be changed.

N.B. uP is actually original uid uP combined with capabilities possesed by capability groups uid is member of

2.2.8 Capability controled system booting

The need for being able to control which processes execute during the system boot has been identified. From the point of view of the Linux kernel the system has finished booting process when init is forked. On the other hand, from users point of view, the booting has finished when all the startup scripts invoked by init have finished execution. CAP_PROC_BOOTTIME capability is introduced to denote programs that are allowed to execute during the init execution phase of system boot. All other programs invoked by exec() during the boot phase (need to alocate kernel variable to indicate this) are stoped.

2.2.9 Capability based process protection

Processes having CAP_PROC_PROTECTED and CAP_PROC_UNKILLABLE capabilities set are protected from receiving any/fatal signals. They can only receive signals from a process that has CAP_PROC_GOD capability set and has the same uid or CAP_KILL capability.

Processes having CAP_PROC_HIDDEN are completely invisible and so are net connections of processes that posses CAP_NET_HIDDEN capability (from the users point of view). These can be used for various mission critical

applications or intrusion detection software.

2.2.10 Inet socket IPC protection

With each socket in the system a set of capabilities required to communicate to the socket can be associated as an attribute of the executable fs image of the process that created the socket. When a connection is initiated to a socket in case of TCP or when packet is sent to a socket in case of UDP and RAW IP the process initiating the connectiong (or sending a message) has to have in its effective capability set all of the capabilities required by the destination socket in order for the operation to succeed.

2.2.11 Unix domain sockets

Required set of capabilities to connect to any process' unix domain socket may be defined on per socket basis. If a source process does not posses a required set of capabilities to connect to destination socket of the destination process its connection request will be refused.

2.2.12 Routing and Firewall tables

Routing and firewall kernel tables can only be modified by processes that posses CAP_MOD_ROUTE and CAP_MOD_FW capabilities.

2.2.13 Dev Security

Certain system resources, such as raw I/O to certain devices needs to be protected. Therefore, separate capabilities will be required for accessing /dev/mem, /dev/[raw disks], /dev/[io ports], /dev/[net] etc. as the project develops.

2.3.1 Acess Domains

The term "Access Domain" denotes a reachable file system environment of a process. Each of the processes on the system can have specific portion of the filesystem reachable to them. Furthermore, each proces can have to access domains:

i) read only access domain
ii) read write accesss odmain

References to the access domains of each of the processes on the system are kept in their task structure. Access domain structure holds dev/inode pairs of each of the files and directories that can be accessed by the process. It is sufficient that one of the parent directory's dev/inode pair is in process' access domain to access any of the files below it in fs hiararchy.
Access domains are originaly stored as extended fs attributes of executables.

2.3.2 Access Domain Groups

In the analogy to the capability groups and traditional notion of user groups there are also access domain groups. Each user can be a member of one or more of the access domain groups either in

a read only or read write manner. Each user is a member of at least default access domain groups.

2.3.3 Inheritance of access domains

When a new process is spawned to execute an executable fs image access domains are inherited by obeying the following algorithm:

```
child->rw_acd = parent->rw_acd + executable->rw_acd;
child->ro_acd = parent->ro_acd + executable->ro_acd;
```

where child/parent->ro_acd and rw_acd are access domains related to executables that were called to create parent, grand parent etc. processes. So effectively, access domains are inherited through exec call chain.
If suid is called to change real uid of process a special flag in current's task_struct is turned on to tell next exec to break the access domain inheritance chain. In such a case the algorithm is:

```
child->rw_acd = executable->rw_acd;
child->ro_acd = executable->ro_acd;
```

User specific access domains come into play when access control check is performed. Then both current->rx_ad and uid->rx_ad are checked, where uid->rx_ad represents rw_ad and ro_ad which is uid specific and depends on access domain groups uid is member of.

2.4.0 Network security

Network security doesen't depend neither on capability support or access domains but it is an important part of the overall security system.

2.4.1 IP Labeling

Task struct of each of the processes in the system might have an IP Labeling list associated with it that is checked whenever the process tries to establish a net connection (INET family only). Entries in the IP Labeling list contain touples [protocol, source ip, source port, destination ip, destination port], they may contain '*' representing a joker sign. If the connection is not found to be allowed in the labeling list it is refused to the process.

2.5.0 Admin tool

LinSec will comprise a set of security admin tools that will enable on-the-fly reconfiguration of the features of the security system. The use of the tools will be granted only after a suitable password has been obtained from the invoking user.

2.6.0 Error messages

Each of the events where access to an object hasn't been granted by LinSec system will be logged.

3.0 Expected outcomes/deliverables

I expect to produce a very detailed design of the overall system meeting all of the objectives mentioned above. However, it might not be

possible to implement all of the objectives fully and therefore system is going to be implemented in an incremental fashion where each of the increments would be functional and operational on its own. The objectives with lower priority are: IPC Security, Dev Security, IP Labeling, Routing and Firewall tablesa. None the less, I shall do my best to implement all of the objectives fully.
The system is expected to be completely transparent to all existing software (depending on the security policy implemented by the system administrator) so that it can be seamlessly used on existing linux servers. The form in which linsec is distributed will be a series of kernel patches.

4.0 Work plan

* Project start - end of reading week (5 weeks):
Further background research, reading and documenting parts of the Linux kernel relevant to the project to understand existing kernel mechanisms, outline design

* End of reading week - end of term (5 weeks):
Detailed design phase, some test code, placing skeleton into linux kernel, testing thus obtained flow of control

* End of term - reading week (8 weeks):
Full implementation

* Start of reading week - end of term (6 weeks):
Work on final report

Appendix D

Interim Report

1. Student's name:

Boris Dragovic

2. Project title:

LinSec - Linux Security Protection System

The title that was specified in the November Project Plan was "LinSec - Linux Security Protection and Intrusion Detection System". However, as the intrusion detection aspect of the project is addressed only by the fact that warning messages are produced and logged on an attempt to violate the system security policy I decided to leave out that part of the title. Any full featured IDS can base its detection mechanisms on the messages produced by LinSec. The fact that the intrusion detection part of the title existed in the first place is due to the misunderstanding of the term.

3. Supervisor:

Graham Knight

4. External supervisor:

Prof Jon Crowcroft, Computer Lab, University of Cambridge

5. Progress made to date:

5.1. LinSec design

The design of the overall LinSec project has been fully finished during the first term in accordance with the project schedule. The design contains definitions of all algorithms, mechanisms, scenarios that are neccessary for corect operation of LinSec as well as documentation on parts of the Linux kernel that needs to be modified and the ways the modifications should be carried out.

5.2. LinSec implementation

Up until this moment full support for capability based MAC operation has been implemented in the Linux kernel (current ver. 2.4.17) according to the specification in the November project plan. The capability model comprises both per user and per executable file capability association. User space admin tool for capability manipulation has been developed as well. The capability model is approximately 45% of the overall expected size of LinSec and was implemented in about 1,700 lines of kernel only code (excluding user space admin tool). The modifications to Linux kernel were across files system code, process management code and bootup code. The system is standalone and fully operational, it is being tested in real world envirnment at the Faculty of EE, University of

Belgrade.

6. Further work to be done

Before the Final Report deadline, at the current pace, I expect to fully implement file system access domain access control mechanism. IP Labeling and TCP/IP INET Socket access control as well as other minor features will be implemented time permitting. The capability model + access domain mechanisms represent 95% of the overall project and I hope that I will have time to implement all of the features.

Appendix E

LinSec Source Code

E.1 Introduction

This Appendix contains a portion of *LinSec* C source code. In particular, code from four different source files has been included:

1. linsec_setup.c: not all source code from this file has been included, due to the space constraints. The source code presented contains the functions responsible for:
   - Setting up the *init* task (pid 1) *LinSec* specific configuration.
   - Configuring capability groups (through the */proc* interface).
   - Configuring file system access domains (through the */proc* interface).
   - Configuring *LinSec* specific user parameters (through the *proc* interface).
   - Managing executable files' capability group and file system access domain settings (through the */proc* interface).
   - Reading user, file system access domain groups and capability groups configuration files.

2. linsec_suid.c: code responsible for *LinSec* actions triggered by a process ownership change (invoked from the `sys_setXuid` family functions).

3. linsec_exec.c: code responsible for *LinSec* actions triggered by executing a binary (invoked from the `do_execve` function).

4. linsec_access_control.c: code responsible for performing *LinSec* specific access control checks on:

   - File system access domains.
   - Signaling capabilities.
   - Process hiding capabilities.
   - IP Labeling lists.

The code included was chosen on the grounds of relevance to the contents of the report.

The full source code can be found, released under the terms and conditions of the GNU/GPL license, on the disk included with the report. The source code is in the form of a standard kernel patch due to the nature of the project. To access the code in a more readable format please apply the patch to the *Linux* kernel source code, as explained in the Systems Manual.

E.2 The Source Code

Bibliography

[1] Kernel newbies. *www.kernelnewbies.org.*

[2] Linux security. *www.linuxsecurity.com.*

[3] Openwall project. *www.openwall.com.*

[4] Security focus. *www.securityfocus.org.*

[5] A.S.Tanenbaum, S.J.Mulender, and R. van Renesse. Using sparse capabilities in a distributed operating system. *Proceedings of the 9th International Symposium on Distributed Computing Systems*, 1986.

[6] Maurice J. Bach. *Design of the Unix Operating System.* Prentice Hall, 1987.

[7] Daniel Pierre Bovet and Marco Cesati.

[8] CERT. Cert/cc statistics 1998-2001. *www.cert.org.*

[9] Crispin Cowan, Steve Beattie, Greg Kroah-Hartman, Calton Pu, Perry Wagle, and Virgil Gligor. Subdomains: Parsimonious server security. *Proceedings of the USENIX 14th Systems Administration Conference (LISA)*, December 2000.

[10] D.Baker. Fortress built upon sand. *Proceedings of the New Security Pradigms Workshop*, 1996.

[11] D.Ferraiolo and R.Kuhn. Role-based access control. *Proceedings of the 15th National Computer Security Conference*, October 1992.

[12] Badger et al. Practical domain and type enforcement for unix. *Proceedings of the 1995 IEEE Conference on Security and Privacy*, 1995.

[13] Linus Torvalds et al. Linux operating system. *www.linux.org.*

[14] Michael Beck et al. *Linux Kernel Internals.* Addison Wesley, 2001.

[15] Andreas Grunbacher. Extended ext2 and ext3 file system attributes. *acl.bestbits.at.*

[16] Andreas Grunbacher. Extended attributes and access control lists for linux. *acl.bestbits.at*, December 2001.

[17] SANS Institute. The twenty most critical internet vulnerabilities. *www.sans.org*, November 2001.

[18] J.S.Shapiro, J.M.Smith, and D.J.Farber. Eros: a fast capability system. *Proceedings of the 17th ACM Symposium on Operating Systems Priciples*, December 1999.

[19] J.S.Shapiro and N.Hardy. Eros: a principle-driven operating system from the ground up. *IEEE Software*, February 2002.

[20] Bell Labs. The creation of the unix operating system. *Bell Labs, Lucent Technologies WWW Site.*

[21] Butler W. Lampson. Protection. *Proceedings of the 5th Princeton Conference on Information Sciences and Systems*, 1971.

[22] P.Loscocco and S.Smalley. Integrating flexible support for security policies into the linux operating system. *Proceedings of the FREENIX Track: 2001 USENIX Annual Technical Conference*, June 2001.

[23] P.Loscocco, S.Smalley, P.Muckelbauer, R.Taylor, S.Turner, and J.Farrell. The inevitability of failure: The flawed assumption of security in modern computing environments. *Proceedings of the 21st National Information Systems Security Conference*, pages 303–314, October 1998.

[24] R.Spencer, S.Smalley, P.Loscocco, M.Hibler, D.Andersen, and J.Lepreau. The flask architecture: System support for diverse security policies. *Proceedings of the 8th USENIX Security Symposium*, pages 123–139, August 1999.

[25] Alessandro Rubini and Johnathan Corbet. *Linux Device Drivers, 2nd Edition.* O'Reilly, 2001.

[26] William Stallings.

[27] W. Richard Stevens. *Advanced Programming in the UNIX Environment.* Addison Wesley, 1992.

[28] Medusa DS9 Team. Medusa ds9 security system. *medusa.fornax.sk.*

[29] T.H.Cormen, C.E.Leiserson, R.L.Rivest, and C.Stein. *Introduction to Algorithms, Second Edition.* MIT Press, 2001.

[30] T.Jaeger, D.Sufford, and D.Franke. Security requirements for the deployment of linux kernel in enterprise systems. *IBM WWW Site*, 2002.

[31] Peter van der Linden. *Expert C Programming.* Prentice Hall, 1994.

[32] W.A.Wulf, R.Levin, and S.P.Harbison. *HYDRA/Cmmp: An Experimental Computer System.* McGraw Hill, 1981.

[33] Dave Wreski. Linux capability faq v0.2. *ftp.guardian.no/pub/free/linux/capabilities/capfaq.txt.*

[34] Huagang Xie and Philippe Biondi. Lids: Linux intrusion detection system. *www.lids.org*.

[35] Steve Zdancewic, Lantian Zheng, Nathaniel Nustrom, and Andrew C. Myers. Untrusted hosts and confidentiality: Secure program partitioning. *Proceedings of the 18th ACM Symposium on Operating Systems Principles*, October 2001.