Slovak University of Technology in Bratislava
Faculty of Electrical Engineering and Information Technology

Reg. No.: FEI-104372-72983

Operating Systems Security

Dissertation Thesis

Ing. Roderik Ploszek

Bratislava, 2023

Slovak University of Technology in Bratislava
Faculty of Electrical Engineering and Information Technology

Reg. No.: FEI-104372-72983

Operating Systems Security

Dissertation Thesis

Ing. Roderik Ploszek

Study Programme:	Applied Informatics
Study Field:	Computer Science
Training Workplace:	Institute of Computer Science and Mathematics
Supervisor:	doc. Ing. Milan Vojvoda, PhD.
Consultant:	Mgr. Ing. Matúš Jókay, PhD.

Bratislava, 2023

Slovak University of Technology in Bratislava
Institute of Computer Science and Mathematics

Faculty of Electrical Engineering and Information Technology
Academic year: 2022/2023
Reg. No.: FEI-104372-72983

DISSERTATION THESIS TOPIC

Student:	**Ing. Roderik Ploszek**
Student's ID:	72983
Study programme:	Applied Informatics
Study field:	Computer Science
Thesis supervisor:	doc. Ing. Milan Vojvoda, PhD.
Head of department:	doc. Ing. Milan Vojvoda, PhD.
Consultant:	Mgr. Ing. Matúš Jókay, PhD.

Topic: **Operating Systems Security**

Language of thesis: English

Specification of Assignment:

On the brink of the third millennium on the Faculty of Electrical Engineering and Information Technology of STU in Bratislava, a new project, Medusa DS9 was created. Its objective was to strengthen the security of the Linux kernel. Similar projects have started to appear around the world. That is why the community of Linux developers defined and implemented LSM (Linux Security Modules) interface that can be used to connect any arbitrary solution providing additional checks and access controls for the Linux kernel.

After almost twenty years, the original design of the Medusa DS9 project has been transferred to the current version of the Linux kernel. The Medusa project differs specifically from other solutions in that only a small fraction of the decision logic is located in the kernel. Most of it is transferred to a process called the authorization server. It applies decision rules based on which it will decide if the event in the kernel will be allowed or denied. Since the implementation of authorization server is separated from an OS kernel, implementation of security model does not require special knowledge of programming and development of the Linux kernel. For this reason, the Medusa project does not enforce any methodology of security policy (MAC, DAC, etc.). Due to this architecture, it is possible to implement almost any access control model.

Access control in Medusa is still on a very low level, ordinary user cannot create consistent rules to increase operating system security. The aim of the thesis is to create a system that monitors the behavior of an application and is capable of automatic learning of rules and creating a policy that can be deployed on a production system.

Goals:
1. Analyze access control models and their connection to operating system security.
2. Analyze the protection mechanisms present in the Linux operating system.
3. Propose a methodology that can be used to obtain information about events in the operating system to create a security policy.
4. Design and implement automatic security rule creation for a selected authorization server.
5. Propose a methodology for comparison and compare existing security policy rules with automatically generated policy.

Deadline for submission of Dissertation thesis:	31. 05. 2023
Approval of assignment of Dissertation thesis:	29. 05. 2023
Assignment of Dissertation thesis approved by:	prof. Dr. Ing. Miloš Oravec – Chairperson of Field of Study Board

SÚHRN

SLOVENSKÁ TECHNICKÁ UNIVERZITA V BRATISLAVE
FAKULTA ELEKTROTECHNIKY A INFORMATIKY

Študijný program:	Aplikovaná informatika
Autor:	Ing. Roderik Ploszek
Dizertačná práca:	Bezpečnosť operačných systémov
Vedúci záverečnej práce:	doc. Ing. Milan Vojvoda, PhD.
Konzultant:	Mgr. Ing. Matúš Jókay, PhD.
Miesto a rok predloženia práce:	Bratislava 2023

Hlavným výskumným problémom riešeným v tejto dizertačnej práci je automatické vytváranie bezpečnostných politík. Bežný používateľ operačného systému Linux zvyčajne používa bezpečnostné politiky navrhnuté vývojármi distribúcie, ale vytvorenie vlastnej bezpečnostnej politiky zostáva zložitou úlohou, ktorá si vyžaduje dôkladnú znalosť systému, jeho služieb a aplikácií. V dizertačnej práci preto navrhujeme nové algoritmy a metodiky na automatické generovanie bezpečnostných politík analýzou správania aplikácií v systéme Linux. Primárne sa zameriavame na operácie so súbormi prostredníctvom dynamickej analýzy. Výskum prispieva sadou algoritmov implementovaných v jazyku Python na generovanie bezpečnostných politík prispôsobených pre bezpečnostný modul Medusa.

Kľúčové slová: Ťažba bezpečnostnej politiky, Linux, Medusa, Linux Security Modules, Povinné riadenie prístupu

ABSTRACT

SLOVAK UNIVERSITY OF TECHNOLOGY IN BRATISLAVA
FACULTY OF ELECTRICAL ENGINEERING AND INFORMATION TECHNOLOGY

Study Programme:	Applied Informatics
Author:	Ing. Roderik Ploszek
Dissertation:	Operating Systems Security
Supervisor:	doc. Ing. Milan Vojvoda, PhD.
Consultant:	Mgr. Ing. Matúš Jókay, PhD.
Place and year of submission:	Bratislava 2023

The core research problem addressed in this dissertation is the automation of security policy creation. The average Linux user typically adopts a security policy designed by the distribution developers, but creating a personalized security policy remains a complex task requiring intimate knowledge of system services and applications. The dissertation thus proposes novel algorithms and methodologies to automatically determine security policies by analyzing application behaviors on the Linux system. Primarily focusing on file operations through dynamic analysis, the research contributes a suite of algorithms for security policy mining tailored for the Medusa security module, implemented in a Python application.

Keywords: Policy mining, Linux, Medusa, Linux Security Modules, Mandatory Access Control

Acknowledgments

I would like to thank my supervisor for his pedagogical care during my PhD studies.

To my advisor, thank you for your professional guidance and expert advice on my thesis.

I thank the staff of the Institute of Computer Science and Mathematics for the excellent working environment.

My great thanks go to the Computer crimes course team. Without their help, this thesis would not have been finished yet.

Contents

Introduction **1**

1 Access control **3**
1.1 Preliminaries 3
1.1.1 Discretionary Access Control 5
1.1.2 Mandatory Access Control 5
1.2 Classic Models 5
1.2.1 Access Control Matrix 5
1.2.2 Access Control List 6
1.2.3 Capabilities 7
1.3 Military Models 7
1.3.1 Multi-level security 7
1.3.2 Multi categories security 9
1.4 Modern models 9
1.4.1 Role-based Access Control 9
1.4.2 Attribute-based Access Control 11
1.4.3 Relationship-based Access Control 12
1.5 Reference Monitor 12

2 Security in Linux **15**
2.1 Discretionary Access Control 15
2.1.1 UGO model 15
2.1.2 Access Control Lists 18
2.2 Linux capabilities 20
2.2.1 Modification of capability sets 23
2.2.2 Computation of capabilities during `execve()` 24
2.2.3 Requirements and examples of capabilities 25
2.2.4 Backward compatibility of `setsuid()` operations 26
2.2.5 Towards capability-only system 27
2.3 Mandatory Access Control 28
2.3.1 Linux Security Modules framework 28
2.3.2 SELinux 29

2.3.3 TOMOYO 32
2.3.4 AppArmor 35
2.3.5 Smack 35
2.3.6 Minor modules 37
2.4 Automatic policy creation 39
2.4.1 SELinux 39
2.4.2 AppArmor 40
2.4.3 TOMOYO 41
2.4.4 Smack 42

3 Introduction to Medusa 43
3.1 Overview of the Medusa system 43
3.2 Medusa Security Model 44
3.3 Medusa Communication Protocol 45
3.3.1 Data types 45
3.3.2 Operations 48
3.4 Kernel module 48
3.5 Authorization server 50
3.5.1 Unified namespace 50
3.5.2 Insertion into the tree 51

4 Related Work 53
4.1 Policy mining from logs 53
4.2 Automatic policy generation 55
4.3 System call interposition 56
4.4 Containerization and Sandboxing 57

5 Policy Mining 59
5.1 Problem Definition 59
5.2 Research questions 59
5.3 Solution Proposal 60
5.3.1 Basic Definitions 60
5.3.2 Decision function in Medusa 62
5.3.3 Getting Logs 62
5.3.4 Generalization 65
5.4 Evaluation 69
5.4.1 Methodology 72
5.4.2 Individual mining 74
5.4.3 Cumulative mining 77

Conclusion 79

6 Rezumé **81**
6.1 Riadenie prístupu 81
6.1.1 Matica riadenia prístupu 81
6.1.2 Zoznam riadenia prístupu 82
6.1.3 Schopnosti 82
6.2 Bezpečnosť v systéme Linux 82
6.2.1 Model UGO 82
6.2.2 Zoznamy riadenia prístupu 83
6.2.3 Schopnosti v Linuxe 83
6.2.4 Povinné riadenie prístupu 84
6.2.5 Linux security modules 84
6.3 Bezpečnostný modul Medusa 85
6.4 Ťažba bezpečnostnej politiky 86
6.4.1 Výskumné otázky 86
6.4.2 Návrh riešenia 86
6.4.3 Získavanie záznamov 87
6.4.4 Generalizácia 87
6.4.5 Zhrnutie výsledkov 88

Bibliography **91**

A Audited operations **101**

B Source code **103**

List of Figures and Tables

Figure 2.1 Data stored within an inode to represent security context of a file. 17

Figure 3.1 Tree structure of unified namespace in Constable. . . 50

Table 5.1 Legend of generalization names used in evaluation tables 74
Table 5.2 Results of policy mining for PostgreSQL 75
Table 5.3 Results of policy mining for OpenSSH 75
Table 5.4 Results of policy mining for Postfix 76
Table 5.5 Results of policy mining for Apache HTTP Server . . 77
Table 5.6 Summary of policy mining for individual services . . . 77
Table 5.7 Results of cumulative mining 78

List of Abbreviations

ABAC	Attribute-based access control
ACL	Access Control List
API	Application Programming Interface
ASCII	American Standard Code for Information Interchange
AVC	Access Vector Cache
BMU	Best Matching Unit
BPF	Berkeley Packet Filter
CIL	Common Intermediate Language
CIPSO	Commercial Internet Protocol Security Option
CPU	Central Processing Unit
DAC	Discretionary Access Control
dentry	directory entry
DS9	Deep Space Nine
DSM	DiskStation Manager
DTE	Domain Type Enforcement
eUID	Effective User Identifier
EVM	Extended Verification Module
FHS	Filesystem Hierarchy Standard
Flask	Flux Advanced Security Kernel
FN	False Negative
FP	False Positive
GID	Group Identifier
HMAC	Hash-based Message Authentication Code
HTTP	Hypertext Transfer Protocol
ILP	Inductive Logic Programming
IMA	Integrity Measurement Architecture
inode	index node
IPC	Inter-Process Communication
LSM	Linux Security Modules
MAC	Mandatory Access Control
MCP	Medusa Communication Protocol

MCS	Multi-Category Security
MLS	Multi-Level Security
NSA	National Security Agency
OS	Operating System
PID	Process Identifier
PPV	Positive Predictive Value
RBAC	Role-based access control
RCU	Read-Copy-Update
ReBAC	Relationship-based access control
RHEL	Red Hat Enterprise Linux
RSBAC	Role-Based Access Control
rUID	Real User Identifier
SELinux	Security Enhanced Linux
SEN	Sensitivity
SID	Security Identifier
SLES	SUSE Linux Enterprise Server
Smack	Simplified Mandatory Access Control Kernel
SOM	Self-Organizing Maps
SSH	Secure Shell
sUID	Saved User Identifier
TCP	Transmission Control Protocol
TF-IDF	Term Frequency - Inverse Document Frequency
TN	True Negative
TP	True Positive
TPM	Trusted Platform Module
TUI	Text User Interface
UBAC	User Based Access Control
UGO	User Group Others
UID	User Identifier
VS	Virtual Space

List of Algorithms

1 Computation of thread capability sets during `execve()` 24
2 Typical processing of an access on layer L2 49
3 Computation of access 63
4 Creation of rules from audit log 65
5 Generalization based on non-existing files 67
6 Generalization based on multiple runs 70
7 regexpFromDiff 71
8 prefixPostfixRegexp 71

Listings

3.1 Definition of file_kobject k-class 45
3.2 Definition of file_kobject structure 46
3.3 Definition of attributes for file k-object 47
3.4 Definition of a getfile event 47
3.5 Snippet of Constable configuration for syslog domain 51
3.6 Example of an automatic hierarchy handler in Constable . . . 52
5.1 Example of an event from audit log 64

Introduction

This dissertation deals with the security of operating systems. Specifically, it concerns mandatory access control (MAC) in the Linux operating system, which is implemented using the Linux security modules (LSM) interface. This interface allows modules to manage security in the system, in such a way that it monitors all security-related operations and, depending on the subject and the object of the operation, decides whether the operation is allowed according to the current security policy.

There are several LSM security modules, the most popular being AppArmor and SELinux. At FEI STU, the Medusa security module was developed, which differs from existing modules in several ways. The security policy is determined by an external authorization server running in the user space, which allows, among other things, remote control of machines. The authorization server communicates with the module in the kernel using a standard protocol, and the information about the permissions of the system entities itself is abstracted into sets called virtual spaces. This makes it possible to represent most security models, as long as the user implements the transformation of the model into a model of virtual spaces in the authorization server.

The research problem we are dealing with in our work is the automatic creation of security policy. A user of a Linux operating system will mostly receive this policy from the developers of the distribution they are using. This is how it works, for example, for Ubuntu, where a standard and extended security policy is available for AppArmor. The SELinux module provides a reference policy for all systems from which distribution maintainers fork, as is the case with Fedora. It is not standard for the average user to create their own security policy.

It is the same in Medusa. For proper functioning system, the user must create his own configuration. In order to do this, the user must have a perfect overview of which services and applications are running on his system and which authorizations are required for their correct operation. Since this is not an easy task, there is an incentive to design a way to automate this activity as much as possible, with minimal user intervention.

Therefore, the research goal of this work is to design processes and algorithms by which it is possible to determine the security policy, or the set of objects to which the application is authorized to access. To do this, it is necessary to analyze the behavior of applications on the Linux system and find out the properties from which information about the security policy can be derived. There are several sources of this information: we are mainly interested in every operation that the program performs and the files on the disk that are associated with the given program.

The scope of our work is limited only to a dynamic analysis of programs, where we focus on file operations. The resulting policy is applicable to the Constable authorization server.

The thesis begins with an explanation of the theoretical background of access control. In the first chapter, theory of access control is introduced and basic access control models are described. The second chapter explains how access control is implemented in the Linux operating system. The chapter briefly describes all the security controls, including UGO, ACL, and MAC. Linux security modules are presented at the end of the chapter. The third chapter is devoted to the Medusa security module. It provides basic definitions and information that are illustrated with code snippets and examples. The fourth chapter presents existing research in related fields. It should be noted that research dealing with the automatic creation of security policies for LSM modules is very small. In the chapter, we summarize the research of security policy mining from logs, then the automatic creation of security policies, the research that deals with the interposition of system calls and finally the research that examines the containerization and sandboxing of processes. In the fifth chapter, we present our security policy mining solution for the Medusa security module.

The main contribution of the work is a set of algorithms for security policy mining for the Medusa security module and the implementation of these algorithms in a Python application.

Chapter 1

Access control

Access control is a crucial component of data security and privacy. In this thesis, we focus on computer access control, specifically applied to an operating system.

An operating system, as one of the fundamental components of a computer, acts as a mediator between user programs and hardware. It must be appropriately designed and implemented to accurately provide services and ensure optimal performance. One of the basic services it provides is **isolation**. Thanks to isolation, running processes (user code running on a CPU which may be possibly harmful) is isolated from other processes and also from the operating system itself, meaning it doesn't have the access to read or modify memory of other processes or the operating system itself.

Isolation provides very safe computing environment — program errors are contained within the same process and don't propagate through the system. However, such strict isolation also creates very restrictive environment for the system application programmers — restricting communication between processes means prohibiting modular design. It is no suprise that operating systems normally allow some sort of communication between processes and the operating system breaks the isolation barriers in a **controlled** way. This is where the access control comes in. It allows to carefully restrict communication flows and explixitly state which operations are enabled for which process. Detailed description of how access control can be implemented is described in this chapter. Implementation of access control in the Linux operating system is discussed in chapter 2.

1.1 Preliminaries

Throughout the thesis, we will be using terms that are well known in the area of access control, but each term might have slightly different meaning

depending on the author or the specific publication. Purpose of this section is to define each term as intended for this thesis.

Subject Subject is an entity in the operating system that is capable of executing system calls. In most of today's operating systems this unit is called a **process**. If not mentioned otherwise, we denote subjects as s_i individually and S collectively.

Object Object is an entity in the operating system that is capable of being an argument of a system call or being pointed to from an argument of a system call. In most of today's operating systems, this entity may be a process, file, socket, device, IPC object and so on. If not mentioned otherwise, we denote objects as o_j individually and O collectively.

Operation Operation is a procedure in which a subject executes some action on one or more objects. In an operating system this procedure is usually invoked through a **system call**.

Permission Permission is a categorical way of describing the type of an operation, for example: read, write, append, execute, etc. Each operation can have one or more permissions. Permissions may also depend on arguments of the operation.

Domain Domain is a property of a subject that represents execution context. This context can depend on various properties — the parent domain (which domain executed current domain), under which effective user is the subject running, under which group is the subject running, etc.

There is 1:N relation between domains and subjects. A subject can only run under one domain, but there may be multiple subjects running under the same domain. Permissions are usually assigned based on a domain, not just a specific subject. For example, mail server program image started by an admin user should get different privileges than the same image started by an ordinary user.

Access Access is a tuple (d, o, P), where d is the domain of currently running subject executing some operation on object o with a set P of requested permissions.

Decision function Decision function $df_{\Pi}(a) : A \to \{1, 0\}$ is a function that for a given access a returns constant 1 if a is authorized according to policy Π, 0 otherwise. A is a set of accesses.

1.1.1 Discretionary Access Control

Access control can be categorized in multiple ways. One of the basic categorizations is to split access control models into two categories: Discretionary Access Control (DAC) and Mandatory Access Control (MAC). Definition of DAC can be found in *Trusted Computer System Evaluation Criteria* [1]:

> A means of restricting access to objects based on the identity of subjects and/or groups to which they belong. The controls are discretionary in the sense that a subject with a certain access permission is capable of passing that permission (perhaps indirectly) on to any other subject (unless restrained by mandatory access control).

Current operating systems define *owners* of individual objects. These owners then have full control over permissions on these objects. This is a typical implementation of DAC, where users can manipulate permissions of objects at their own *discretion*. Examples in the Linux operating system are UGO (see section 2.1.1) and ACL permissions (see section 2.1.2).

1.1.2 Mandatory Access Control

MAC allows only a specific principal in the system (the administrator) to specify permissions in the system. These permissions are not overridable. In other words, users can't delegate their permissions to other users — permissions are set system-wide.

It took a longer time for this type of access control to appear in operating systems. Initially, it was designed for military use. Bell-La Padula, Biba, Multi-layer security and Multi-category security are typical examples of MAC. When operating systems such as UNIX were being developed, they operated in mostly harmless environment, where the only danger was operator error. The DAC present in this system was mostly for protection of the data. MAC takes this notion up a level — it brings security to the access control.

1.2 Classic Models

This section presents the basic access control model, the access control matrix, together with two models thar are derived from it: access control lists and capability lists.

1.2.1 Access Control Matrix

This section is based on [2].

Access control matrix was described by Lampson [3]. It is a simple, but fundamental model of access control. Access Control Matrix consists of rows

representing subjects in the system and columns representing objects in the system.

State of the system is defined by a triple (S, O, M), where S is the set of subjects, O is the set of objects and M is the access control matrix. One element of the matrix is denoted as $M[s_i, o_j]$, where s_i is subject in i-th row and o_j object on j-th column. The matrix element contains a set of **permissions** that s_i is able to act on o_j. Since on real systems the size of this matrix tends to be big and most of the elements are empty, it is not used in this form. Rather, decomposition of the matrix into either rows or columns is used. In the former case, the permissions are stored along with subjects, creating capability lists of permissions for each object (see 1.2.3). In the latter case, the permissions are stored with each object in the system, creating access control lists (see 1.2.2).

Access Control Matrix models a very simple representation of the policy: it shows which subjects can execute some operations on which objects. However, there are some rules which cannot be easily represented in the matrix without losing the original meaning. For example, rule such as *everyone can write* o_1*, except* s_1 can be represented in the matrix, but once written, the original semantic meaning of the rule is hard to infer just from looking at the resulting matrix.

Either way, the access control matrix serves as a theoretical basis for other security models, since every security model can be transformed into access control matrix, losing the original semantic meaning of the policy model. Since access is represented as a triple (s, o, a), we can query the decision function of any model for each $s \in S$, $o \in O$ and $a \in A$. Triples for which the decision functions returns 1 can be used to construct an equivalent access control matrix.

1.2.2 Access Control List

By decomposing the access control matrix by columns, we get access control lists (ACLs). Access control list for object o is represented as a list of tuples $(s_i, \{a_1, \ldots, a_n\})$, where s_i is a subject and $\{a_1, \ldots, a_n\}$ is a set of permissions s_i is allowed to act upon o. Since the access control list is stored together with the object, this makes it easier for the system administrator to see a complete list of subjects that are able to act upon a specific object. The reverse is more complicated — to be able to see all operations that a specific subject can do, we would have to scan all objects in the operating system for their access control lists.

1.2.3 Capabilities

By decomposing the access control matrix by rows, we get capability lists (C-lists). Capability list for a subject s is represented as a list of tuples $(o_i, \{a_1, \ldots, a_n\})$, where o_i is an object and $\{a_1, \ldots, a_n\}$ is a set of permissions s can act upon o_i.

Now the advantages and disadvantages of C-lists are swapped when compared to access control lists. Getting all permissions for a specific subject is trivial, since they are stored together with the subject. Getting a list of subjects that are allowed to make some operations on a specific object is not possible without iterating through all subjects in the system.[1]

We can think of capabilities as credentials — owning them allows the owner to access objects listed in the C-list and execute permitted operations. This offers possibilities that are not available with ACLs, for example delegation. Systems that support capability delegation have to solve the problem of capability revocation.

1.3 Military Models

This section presents access control models that were originaly developed for military purposes and later repurposed for usage in non-military systems.

1.3.1 Multi-level security

These models were created and used by the U.S. military. Most well-known model is the model created by Bell and LaPadula, known as the Bell-LaPadula model [4].[2]

For the explanation of the Bell-LaPadula model, we are using adaptation from [5]. In this model, a partially ordered hierarchy of security levels is defined as a security structure $(\mathbf{L}, \leq)$, where $\mathbf{L}$ is a set of security levels and $\leq$ is a partial order defined on $\mathbf{L}$. $L_i \leq L_j$ means that security level L_i is less than or equal to L_j. In terms of the security model, this means that L_j dominates L_i. L_j is called the **dominating** level and L_i is the **dominated** level. When $L_1 \nleq L_2$ and $L_2 \nleq L_1$, the security levels L_1 and L_2 are **incomparable**. The standard set $\mathbf{L}$ used by the military is defined as $\{\mathit{UNCLASSIFIED}, \mathit{CONFIDENTIAL}, \mathit{SECRET}, \mathit{TOPSECRET}\}$, in ascending order of security level.

Each entity in the system is assigned a security level. Security level of an object is called a **classification** and security level of a subject is called

[1] But the real time of this operation compared to searching ACLs for a specific subject might be smaller as in most systems there is significantly more objects than subjects.

[2] Note that Bell-LaPadula module presented here intentionally doesn't mention the category part of the model. This is mentioned in the following section 1.3.2.

clearance.

Note that while in classic models, the permissions are given explicitly, in the multi-level security, the permissions are inferred based on rules using the notions of information flow [6]. Bell-LaPadula defines following properties which determine which operations are allowed. These definitions use function L that assigns security levels to the respective subject or object.

Simple Security Property A subject s can *read* object o only if the security level of object o dominates the security level of s, that is, $L(o) \leq L(s)$. This property is also known as the *no read-up rule*, meaning that a subject shouldn't have read access to objects that are above its clearance level.

***-Property** A subject s is allowed to *append*[3] to an object o only if the security level of object o dominates security level of subject s, that is $L(s) \leq L(o)$. This property is also known as *no write-down rule*, meaning that a subject can't write to a lower sensitivity objects, therefore preventing unauthorized disclosure of high-sensitivity information to an object that can be read by subjects with lower-sensitivity clearance.

Discretionary Security Property enables additional DAC. Individual users may modify access to an object to other users provided the previous two propertiers are enforced (i.e., DAC can't override denial of access by previous properties).

As a consequence, if a subject s wants to have both read and write access (ability to observe and modify the object) to object o, it has to have the same clearence as the sensitivity of the object, that is, $L(o) = L(s)$.

As stated above, Bell-LaPadula ensures only confidentiality of data. Other important aspect of information assurance from the well-known CIA triad is *integrity.* Thus, shortly after the original model by Bell-LaPadula, Biba [7] presented an integrity model as a complementary model to the original Bell-La Padula.

This model introduces *integrity levels* that may have the same values as security levels, but they have different semantic meaning. Each subject and object in the system are assigned an integrity level. Function I returns integrity level for a given subject or object. The system complies with the strict integrity policy if the following properties are kept [2]:

[3] In the original technical report by Bell and LaPadula, append access is defined as having the right to write to an object without observing the object

The Simple Integrity Property A subject s can *observe* object o only if the integrity level of s dominates the integrity level of o, that is, $I(s) \leq I(o)$. This property is also known as the *no read-down rule*, meaning that a subject shouldn't have read access to objects with lower integrity that may corrupt the integrity of the subject.

***-Integrity Property** A subject s is allowed to *write* to an object o only if the integrity level of s dominates the integrity level of o, that is $I(o) \leq I(s)$. This property is also known as *no write-up rule*, meaning that a subject of lower integrity level can't write to a higher integrity object, therefore preventing the corruption of a high-integrity object.

Invocation Property Subject s can invoke other subject s' only if the integrity of s dominates integrity of s', that is, $I(s') \leq I(s)$. This prevents subject s from affecting high-integrity objects indirectly through another high-integrity subject.

1.3.2 Multi categories security

This section is based on [2].

Until now, we have presented only one dimension of Bell-LaPadula model that lies in the security level. Next formal security designation is a formal *category*. Set of categories C is usually determined according to the entities of the organization, or the topic of the given object. An example set for an organization might be: management, public relations, accounting, audit, etc.

Motivation for adding categories to the model is a *need-to-know* principle. According to this principle, subjects should have access only to objects that are necessary for their function.

Definition of a security level is changed to a tuple (l, c), where l is a confidentiality level and c is a set of categories. The definition of partial order $\leq$ on security levels is also changed. Security level (l, c) dominates (l', c') when $l \leq l'$ and $c' \subseteq c$.

1.4 Modern models

Following models are "modern" in the sense that they were developed after the classic and military models and they are focused on improving the management of policies and bring the model closer to the actual structure of the protected system.

1.4.1 Role-based Access Control

Role-based access control (RBAC) was introduced by Ferrailo and Kuhn [8] and later formalized into four reference models by Sandhu et al. [9]. RBAC is

a form of mandatary access control, but a discretionary version has also been developed.

As RBAC was developed for organizational security requirements, subjects are represented by **users** in the organization. These users can perform transactions which are higher-level collections of operations on some data contrary to read/write permissions of simpler access control models. However, later papers use the term *permissions* instead of transactions. From now on, we will use the term permissions. RBAC adds a layer of indirection between users and permissions represented by **roles**. A role defines permissions that the user who is assigned to this role can perform. Users are then assigned to roles according to their position in the organization.

Dynamic state of the system is represented by **sessions**. These are controlled by the individual users. When user starts a session, he can choose which roles to activate. During the session, he can disable or enable roles that are assigned to the user. This encompasses the base reference model that is referred to as $RBAC_0$. The main advantage of RBAC comes from the management abilities of this model. When a new person is introduced to the organization, security officer will assign roles to him according to his position without the need of consulting and assigning individual permissions. A similar procedure applies when revocating privileges for people switching departments or leaving the organization. Security officer simply removes or adds necessary roles. When a new system is added to the organization, security officer can assign new permissions to existing roles without manually assigning permissions to users. This minimizes mistakes and allows principle of least privilege to be enforced (each role contains permissions necessary to perform that role, nothing more).

In adition to $RBAC_0$, two independent models were introduced. The first one, $RBAC_1$ introduces role hierarchies. These hierarchies naturally reflect organization's lines of authority and responsibility [9]. More powerful senior roles inherit permissions from less powerful junior roles. As an example, take a hiearchy of these roles: *teaching assistant*, *lecturer* and *course supervisor*. Course supervisor is the senior role, so it has it's own permissions along with inherited permissions from the teaching assistant and lecturer roles. Role of a lecturer is in the middle of the hierarchy, so it inherits permissions of teaching assistant but it doesn't include permissions assigned to course supervisor. The hierarchy of roles inherently creates a partially ordered set.

The second additional model, $RBAC_2$, introduces concept of constraints. Multiple types of constraints were described in the literature. The first one is the ability to forbid *mutually exclusive* roles. This means that the system won't allow security officer to assign mutually exclusive roles to one user. This allows separation of duties, a well known concept from physical

security that predates computer access control. Second type of constraints are *cardinality constraints*. These can be used to limit number of users for one role. For example, the role of a *dean* of a faculty can be assigned to one user at a time. Last type of constraints mentioned in this section are *prerequisite constraints*. These are based on notions of competence and appropriateness [9]. Prerequisite constraints can relate to roles themselves — for example a user can only be assgined role B if she already has role A assigned. They can also relate to permissions — permission p can be assigned to a role only if that role already possesses permission q [9].

Last reference model, $RBAC_3$ is a consolidated model that combines hierarchy model $RBAC_1$ and constraints from $RBAC_2$.

RBAC can be considered high-level model compared to ACLs or C-lists, since it doesn't define system-level operations such as read or write but rather complete transactions that don't just include data, but specific operations in correct order needed to succesfully execute the transaction. Thus we say that RBAC allows data abstraction [9]. As a result, it's not used as much in operating system security as it is used in application-level security. RBAC is currently considered a standard model for almost all software. It is used in numerous applications, including Discord, Azure Active Directory [10], Kubernetes [11] and many others.

1.4.2 Attribute-based Access Control

Description of Attribute-based access control (ABAC) is based on [12, 13].

ABAC is a continuation of RBAC intended for complex and dynamic systems, where role administration may be too cumbersome. Instead of working with identities of user and object, it bases the access decision on attributes of the subject (id, clearance, role, company position, ...), the object (type, sensitivity, location, ...), the operation (read, write, execute, ...) and the environment (current time, location or other attributes that dynamically change according to context).

There are two types of attributes [13]: atomic-valued and set-valued. Atomic-valued are represented by a single value. For example $sensitivity(o) = S$ specifies atomic-valued object attribute *sensitivity* that for object o has a single value S (secret). Similarly, set-valued attributes return a set of values. For example, $roles(u) = \{instructor, examiner\}$ returns set-valued subject attribute *roles*.

In ABAC models, the policy rules can primarily be articulated through two distinct methods: based on logic formulas and enumerated policies. The traditional method involves the creation of policies using logical formulas that incorporate the values of attributes. These formulas consist of one or more predicates joined by various logical operators. One predicate compares

a subject/object attribute with another subject/object attribute or a constant value. The alternative method for formulating policies relies on enumeration. Policy expressed in Policy Machine is defined as (sa_i, OP, oa_i), where sa is value of subject attribute, oa is the value of object attribute and OP is a set of allowed operations for subjects and objects with the listed attributes.

Similar to RBAC, ABAC is implemented at the application level in web-based applications and information systems. We have not found any notable implementation of ABAC for an operating system.

1.4.3 Relationship-based Access Control

Relationship-based access control (ReBAC) makes authorization decisions based on relationship between subjects and objects. Typical usage scenarios are for social networks. For example, a post can be edited by user who created it and read only by friends of that user.

Term ReBAC was used for the first time by Carrie E. Gates [14]. One notable implementation of a ReBAC system is Zanzibar [15] by Google. Altough it never specifically mentions ReBAC, it has the characteristics of ReBAC. Rules are represented by triples (u, r, o) which mean user u has relation r to object o. Language of Zanzibar includes set-algebraic operators that allows to create more complex rules, such as set of users S has relation r to object o. S itself may be defined as another object-relation pair.

1.5 Reference Monitor

Reference monitor is an abstract concept that describes requirements of the enforcement mechanism of a perfectly secure system. It can be used to implement some access control model into a real system. The security policy can be correctly enforced if these requirements are followed [2]:

Complete mediation requirement The enforcement mechanism should *mediate* all security-sensitive operations. Correct authorization of domain operations can be performed only if the security monitor is always invoked during security-sensitive operations.

Tamper-proof requirement This requirement prohibits anyone from changing the reference validation mechanism except the system administrator (meaning an ordinary user shouldn't be able to override the permission decision of the policy). An ordinary user should also be not able to modify the security policy itself.

Verifiability requirement This requirement states that practical verification of correctness of the reference monitor should be possible. One

should be able to verify if the reference monitor produces correct access tuple, if this tuple is correctly processed against the security policy and if the resulting decision is correct according to the security policy. Individual reference monitors can also state their own verifiable goals that should be followed by the reference monitor mechanism.

Fulfilling all requirements of the security monitor might not be enough to have *a secure system*. Correct function of the reference monitor mechanism also depends on a number of supporting functions, such as authentication system, correct hardware operation and physical security [16].

Chapter 2

Security in Linux

2.1 Discretionary Access Control

Linux, as a UNIX-derived operating system inherited basic protection scheme from the UNIX operating system. In this section, we will introduce two systems of discretionary access control available on most Linux distributions.

As a reminder, discretionary access control allows owners of objects to change the permissions at their own discretion. This allows simple sharing of objects, but at the cost of information disclosure — any user can set permissions too low (either by mistake or intentionally), allowing data to be read by unauthorized users. This problem is solved by using mandatory access control. Implementation of mandatory access control in Linux is explained in section 2.3.

2.1.1 UGO model

This section is based on [16].

UGO architecture is a simplification of fully-fledged access control lists that was proposed and implemented due to memory and processing requirements of full ACLs.

Note that the main purpose of UGO architecture is to compartmentalize and secure accesses to files to each individual user on the system and provide methods to share these files in a controlled way. In this case, it can be controlled by owners of these files — thus, UGO is a DAC model.

Each file is assigned an *owner* and a *protection group*. These are identified by numerical identifiers UID and GID, respectively. When a new file is created, these values are usually set using the effective UID and GID of the thread that creates it.

Permissions in the UGO model can be assigned to three "sets" of users:

owning user These permissions apply to the user that *owns* this file. This set always contains just one user.

group Permissions that apply to users that belong to the owning file group.

other users Permissions that apply to users not covered by the previous sets.

Each of these sets may contain three permissions: *read*, *write* and *execute*, denoted by *r*, *w* and *x*, respectively. Current state of permissions in a given set is represented by three bits.

Lastly, there are three special protection bits that are not linked to any UGO set, but apply to the whole file: *setuid*, *setgid* and *restricted deletion* bit, also known as sticky or t-bit. Closer explanation of these special bits follows:

setuid When a process executes a program file with *setuid*, it's effective UID will be set to the owner of the program file.[1] When *setuid* is set on a directory, this causes newly created files in that directory to inherit the directory owner UID. *setuid* is also recursively applied to new subdirectories.

setgid Has the same effect as *setuid*, but affects the group UID.[1]

restricted deletion When set on a directory, files inside the directory can only be removed or renamed by their owner, owner of the directory or a privileged process. This is useful for writeable public directories such as `/tmp`, as it protects files from being deleted by other malicious processes. Setting it on a file doesn't have an effect in Linux.[2]

Note that permissions are checked against the first set that matches. This means that owning user can disable permissions to herself. This can be useful when she wants to protect the file from accidental modification or for disabling a configuration file by removing read acces to it [17].

[1] There are three exceptions to this. Effective UID or GID is left unchanged if the `no_new_privs` sttribute is set for the calling thread, if the underlying filesystem is mounted with `nosuid` flag or if the calling process is being *ptraced*.

[2] In other Unix-like operating systems it was used to "stick" the program file to swap space, so it loads faster in subsequent executions. Advances in memory caching techniques and hardware made it obsolete and it was never implemented in Linux.

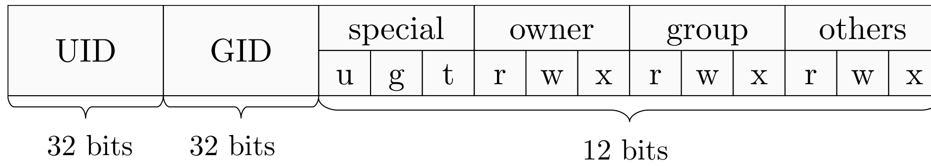

Figure 2.1: Data stored within an inode to represent security context of a file.

Credentials

Linux stores all process identifiers in a special structure whose contents is called **credentials**. There are few identifiers, that concern users and groups:

Real user and group IDs These IDs determine who owns the process.

Effective user and group IDs These IDs are used by the kernel to check permissions when the process accesses shared resources such as message queues, shared memory and semaphores.

Saved set-user-ID and set-group-ID These IDs are used for storage of the corresponding previous effective IDs when process executes a set-user-ID or set-group-ID programs. Such program can switch the value of the effective ID between real and saved ID as needed.

Filesystem user ID and group ID These IDs are used to match file owner and group IDs to determine permissions for accessing files in the filesystem. When effective UID or GID is set, the filesystem UID/GID is set to the same value as well. These IDs were introduced to protect user space servers (such as NFS) from receiving signals from processes with the same effective UID. By splitting effective and filesystem UIDs, the server could use less privileged UID for filesystem access while staying protected from unwanted signals with higher privileges.

However, this was made obsolete and these IDs are kept just for backwards compatibility.

Supplementary group IDs This set contains other group IDs to which the process belongs. Together with the filesystem group ID, they are used for permissions checks as long as one of these groups matches the file group.

Permission checks

Permission bits are checked during the path resolution. First, the process needs to have search permission to all components up to the final component of the path. The final component represents the accessed object. The

permissions to access it are determined by the access required (read, write, execute). To allow the operation, the credentials of the thread must contain all required permissions.

2.1.2 Access Control Lists

This section is based on the withdrawn POSIX.1e draft [18] that defined requirements for Access Control List (ACL) implementations and acl(5) Linux manual page [19] that documents the Linux implementation of ACL.

Access Control Lists as defined by POSIX.1e set out to solve the problem of expressiveness of the original UGO implementation with the goal of keeping the implementation as simple as possible and compatible with UGO model. The result was an extension of the UGO model. ACL uses the same `rwx` permissions from the UGO model, but allows more fine-grained access control of additional users and groups.

There are two types of ACLs: access ACLs and default ACLs. Access ACLs are used when requesting access to an object, while default ACLs are used when a new object is created within a folder.

ACLs consist of individual ACL entries. An ACL entry contains type of the entry, qualifier specifying identity of a user or a group and permissions pertaining to the entry. Valid ACL has to contain at least three ACL entries of these types: `ACL_USER_OBJ`, `ACL_GROUP_OBJ` and `ACL_OTHER`. These directly reflect UGO permissions in that order (however, there is an exception for `ACL_GROUP_OBJ` and file group permissions, see below).

Granularity that wasn't possible with standard UGO model is achieved using ACL entries of two additional types: `ACL_USER` and `ACL_GROUP`. ACL can contain arbitrary number of entries of these types. These types allow to specify permissions for individual users other than the file owner (using the `ACL_USER` type). Also, compare this with the situation from the UGO model — if the object owner wanted to share an object with multiple users, he would have to group those users in a group. Restricting permissions is also possible — `ACL_USER` entry may prohibit access of some user even though the user may be present in a group that has permissions to access the object. This can be achieved because the order of permission check is the same as in UGO model. It starts matching against the most specific entries (`ACL_USER_OBJ` and `ACL_USER` entries), moving to group entries and finally using permissions listed in the `ACL_OTHER` entry if no other entry matched. For the full description of ACL decision algorithm, see the end of this section.

The `ACL_GROUP` type gives file owner an ability to set additional permissions to groups that are distinct from the file group. This allows administrators to create more specific groups with finer granularity of users that improves the application of the principle of least privilege.

The implementation explained up till now has some drawbacks. Consider following scenarios [18]:

1. Legacy program that doesn't support ACLs calls `chmod(object, 0)`. In the UGO model, this would prevent access to the object to everyone. However, if the object has access ACL with entries of types `ACL_USER` or `ACL_GROUP`, these would still allow users or groups specified in those entries to access the object. Thus, the original semantics of `chmod()` operation would be broken.

2. Similarly, the command `chmod go-rwx` restricts access only to the file owner (if such permission already exists for the file owner) according to the UGO model. However, if ACL entries of types `ACL_USER` or `ACL_GROUP` exist for the file, there may be users or groups that still have access to the file.

To solve these compatibility issues, a new type of entry was added, `ACL_MASK`. Permissions included in this mask represent the maximum permissions that can be granted by ACL entries of type `ACL_USER`, `ACL_GROUP_OBJ` and `ACL_GROUP`. There can be only one entry of this type in ACL and it's *required* to be present if there is any entry of type `ACL_USER` or `ACL_GROUP` (note that the compatibility problems with UGO were present if any of those entry types was present in ACL).

Addition of `ACL_MASK` entry also changes the interconnection between UGO and ACL models. When `ACL_MASK` is present in ACL, changes to it are propagated into file group permissions and vice versa. This is to preserve backwards compatibility with UGO model, as explained in the examples above.

Another type of ACLs, default ACLs can be assigned to directories. Entries contained within default ACL are automatically assigned to new objects created in that directory as their access ACL. Operations that create new objects (`creat`, `open`, `mkdir`, `mkfifo` and `mknod`) honor the `mode` parameter and set permissions of ACL entries that correspond to file permission bits so that if the permission is not present in `mode`, it is removed from the related ACL entry.

Whether access is granted to a object is determined according to this algorithm:

1. If the current effective user is *owner* of the object:

    (a) if `ACL_USER_OBJ` entry contains permissions, access is **granted**.

    (b) otherwise, the access is **denied**.

2. If the current effective user matches any `ACL_USER` entry:
   (a) if `ACL_MASK` entry and `ACL_USER` entry contain the requested permission, operation is **granted**.
   (b) otherwise, operation is **denied**.
3. If the current effective group or supplementary group matches entry of type `ACL_GROUP_OBJ` or any entry of type `ACL_GROUP`:
   (a) if ACL contains entry of type `ACL_MASK`:
      i. if `ACL_MASK` entry and any matching entry belonging to the types `ACL_GROUP_OBJ` or `ACL_GROUP` contains requested permission, operation is **granted**.
      ii. otherwise, operation is **denied**.
   (b) otherwise, if `ACL_GROUP_OBJ` contains requested permission, operation is **granted**.
   (c) otherwise, operation is **denied**.
4. Otherwise, if `ACL_OTHER` entry contains the requested permission, opertaion is **granted**.
5. otherwise, operation is **denied**.

2.2 Linux capabilities

This section is based on Linux kernel manual page on capabilities [20] and withdrawn POSIX-1003.1 draft [18].

In UNIX-like operating systems (without capability support), there are two basic privilege levels. When process is running under effective UID equal to zero, it is *privileged* and can perform any task on the system. Otherwise, it is *unprivileged* and can't execute privileged actions (such as opening low port or rebooting the system).

Capabilities in Linux is a set of permissions that divides privileges of the superuser into smaller units, making it possible to explicitly specify which permissions are needed for the running process, thus adhering to the principle of least privilege.

The main motivation for the introduction of capabilities were privileged applications (e.g., network daemons) that did not need all the privileges of the superuser. Since such programs were often available remotely via open ports to the Internet, they became targets of attacks. If an attacker was able to execute arbitrary code through such a program, he automatically obtained all superuser privileges and could perform arbitrary actions on the system.

There are two types of capabilities in Linux: ones that are assigned to a running thread (*thread capabilities*) and those that are assigned to an executable file (*file capabilities*). File capabilities are activated once the executable is executed using `execve()` system call. This allows unprivileged threads to acquire new privileges in a controlled way and also to specify capabilities at the level of individual program images. For the full explanation of capability activation during `execve()`, see section 2.2.2.

Before we introduce specifics of capabilities in Linux, we have to establish requirements placed on capability system during its design:

- It should be possible to set capabilities for each program image individually.
- A thread should be able to obtain capabilities from a program image by executing (`execve()`) the program.
- Running thread should be able to temporarily disable or enable capabilities according to current needs and the principle of least privilege.
- Running thread should be able to revoke obtained capabilities so it can't use them anymore unless they execute another program that grants capabilities.
- Running thread should be able to selectively pass capabilities to executed programs (via "inheritence").
- Program images should be able to restrict which capabilities can be inherited by the executing thread.
- Capability system should be backwards compatible with programs that don't support capabilities or rely on standard POSIX privilege-gaining mechanisms via set-user-ID-root.

To implement these requirements, each thread needs to have multiple sets of capabilities. They are described in the list below. Note that bounding and ambient sets were added in later versions. See the explanation and rationale below the list.

effective This is the set of capabilities that is used for permission checking. Having a capability in this set means that the capability is active and the thread can utilize it.

permitted This is the maximum set of capabilities that a thread can use. It is also the maximum set of capabilities that a thread can add into the inheritable and effective set.

inheritable Capabilities in this set are marked to be preserved during `execve()` if the program image capabilities agree with this.

bounding This set restricts capabilities that the thread can gain during `execve()`. Bounding set was originally a system set and per-thread support was added later in 2008 [21]. It normally contains all capabilities (capabilities not present in the set are not inherited).

ambient Capabilities in this set are preserved during `execve()` when executing non-privileged files. Note that this is useful for unpriviliged threads as privileged threads can use the *inheritable* set for the same purpose. A capability can be inserted into this set only if it is already present in *permitted* and *inheritable* sets.

When capabilities were originally introduced, only first three sets from the POSIX draft were available. It soon became evident that another sets were needed to better control transfer of capabilities during `execve()`.

Bounding set was originally introduced in version 2.2.11 as a system-wide set [22]. Capabilities not present in this set were automatically removed from all threads, essentially removing them from the system.

Ambient capabilities were added as a compatibility “inheritence” solution for unprivileged processes that want to execute unprivileged image files. Typical example presented in the proposal [23] is when an unprivileged (non-root) thread with capabilities executes a helper program image without file capabilities, this thread can’t keep the inheritable capabilities according to exec transition rules (see 2.2.2). Thus, ambient capabilities are essentially inheritable capabilities for unprivileged threads.

As mentioned above, the only way to gain new permitted capabilities is to execute a program image that offers some new capabilities. This is achieved by interlinking *file capabilities* with executable files. When the program image is executed using `execve()`, the capabilities of the image file together with capabilities of the calling thread are used to determine final capabilities. Similar to thread capabilities, there are multiple sets of capabilities that can be used to precisely specify the final thread capabilities. There are two file capability sets and one file capability flag:

permitted Capabilities in this set are automatically permitted to the calling thread during `execve()` (except for capabilities not present in the bounding set). This set should contain **required** capabilities without which the program can’t function properly.

inheritable This set is compared with the *inheritable* set of the calling thread and only capabilities present in **both** sets are transferred into

the *permitted* set. *Inheritable* set should contain capabilities that the program can potentially use, but are not required for normal operation. It is the responsibility of the calling thread to set requested capabilities into it's *inheritable* set.

effective This is not a set, but merely just a bit flag. If it is set, then capabilities present in the *permitted* set are automatically activated in the *effective* set. This is useful for programs that are not capability-aware and can't change *effective* capability set on their own (they are also referred to as *capability-dumb* in the manual).

2.2.1 Modification of capability sets

Capability sets of a thread can be changed dynamically during the duration of thread execution using `capset()` system call, or preferably, using the *libcap* library. Thanks to this, a thread can achieve better granularity of capabilities according to current need by selectively enabling, disabling or revoking its capabilities.

For thread capabilities, it is always valid to remove capability from effective, inheritable and ambient set. See the list below for the rest of the rules:

permitted Thread can only remove capabilities from this set. A capability can't be removed if it's also present in the effective set. The same capability is also removed from the ambient set, if present. New capabilities can be added into this set only by executing program images that have file capabilities.

effective A thread can "activate" a capability by adding it to it's effective set. This can be done only if the capability is present in the permitted set.

inheritable Capablity can be added to this set only if it is present in both permitted and bounding set. When removing a capability, it is also removed from the ambient set, if present.

bounding Capablities can be removed from the set if the thread posesses `CAP_SETPCAP` capability. Capabilities can not be added to the set.

ambient Capabilities can be added to this set only if they are present in both permitted and inheritable set.

File capabilities can be arbitrarily modified by a thread that holds the `CAP_SETFCAP` capability.

2.2.2 Computation of capabilities during `execve()`

Algorithm 1 shows the transformation of thread capability sets when executing a program image. P denotes state of thread capability sets right before `execve()`. Low indices of P: e, p, i, b and a signify effective, permitted, inheritable, bounding and ambient sets, respectively. P' denotes state of thread capability sets after `execve()`, i.e. after the new program image is executed. F denotes state of file capabilities of the program image that is about to be executed. Meaning of low indices of F: e, p, i is the same as for the thread capabilities. F is privileged if the file has file capabilities or set-user-ID or set-group-ID bit set. $\mathcal{C}$ is a set of all capabilities. See the explanation of the algorithm below.

Algorithm 1 Computation of thread capability sets during `execve()`

Input: P, F
Output: P'
1: **if** $P_{ruid} = 0$ **or** ($P_{euid} = 0$ **and** F is not privileged) **then**
2: $F_p \leftarrow \mathcal{C}$
3: $F_i \leftarrow \mathcal{C}$
4: **end if**
5: **if** $P_{euid} = 0$ **then**
6: $F_e \leftarrow 1$
7: **end if**
8: **if** F is privileged **then**
9: $P'_a \leftarrow \{\}$
10: **else**
11: $P'_a \leftarrow P_a$
12: **end if**
13: $P'_p \leftarrow (P_i \cap F_i) \cup (P_b \cap F_p) \cup P'_a$
14: **if** $F_e = 1$ **then**
15: $P'_e \leftarrow P'_p$
16: **else**
17: $P'_e \leftarrow P'_a$
18: **end if**
19: $P'_i \leftarrow P_i$
20: $P'_b \leftarrow P_b$

Lines 1–7 are provided for backwards compatibility with standard UNIX semantics. When thread is running under supervisor, the file capabilities are ignored and file capability sets are considered to contain all possible capabilities with file capability effective bit considered to be set. Evaluation

continues by determining the new ambient set (lines 8–12). As mentioned above, ambient capabilities were designed to be used with unprivileged files. Ambient set of thread that executes a privileged file (such as a file with associated file capabilities) is cleared, so that file capabilities are observed.

New permitted set is computed as a combination of three sets: ambient capabilities, permitted file capabilities intersected with thread's bounding set and file inheritable set intersected with thread inheritable set (line 13).

Next, effective set is evaluated based on the file capability effective flag. If it is set, the permitted set is copied into the effective set (every possible capability that might be activated for this thread is activated). Otherwise, ambient set is copied into the thread effective set (lines 14–18).

Inheritable and bounding sets are left unchanged.

2.2.3 Requirements and examples of capabilities

This section shortly summarizes recommendations for designing the actual capability set. These principles were originally developed in the POSIX draft [18] and later followed by the Linux kernel developers when introducing capabilities into the Linux kernel.

- A capability should permit the system to exempt a process from a specific security requirement. This means that the capability should provide only the minimum rights to perform a specific task. This essentially captures the principle of the least privilege.
- There should be a minimal overlap between the effects of capabilities. Capabilities should be unique and specialized. No capability or combination of capabilities should provide the privileges supplied by another capability.
- Considering previous two principles, fewer capabilities are better than more. There are two reasons that support this principle: fewer capabilites are more manageable for the system administrator, and the memory requirements are lower.

  Minimum number of capabilities can be achieved by comparing security requirements of proposed capabilities. If there is an overlap between two capabilities, it's better to combine them into one capability.

As of Linux 6.1, there are 40 capabilities defined [20]. Here is a random small selection of them. The last two capabilities in the list are presented as examples of bad capability proposals that got into the kernel.

`CAP_CHOWN` Allows making arbitrary changes to file UIDs and GIDs.

`CAP_DAC_OVERRIDE` Bypasses UGO file permission checks.

`CAP_KILL` Bypasses permission checks for sending signals.

`CAP_MAC_OVERRIDE` Overrides Mandatory Access Control (see 2.3). This is currently recognized only by the Smack security module.

`CAP_NET_BIND_SERVICE` Allows to bind a socket to Internet domain privileged ports (port numbers less than 1024).

`CAP_SYS_BOOT` Allows thread to use `reboot()` and `kexec_load()`.

`SYS_TIME` Allows to set system and real-time clocks.

`CAP_SYS_PTRACE` Among other abilities, allows to trace arbitrary processes and inspect them using `kcmp()`.

`CAP_SYS_MODULE` Allows loading and unloading of kernel modules.

`CAP_SYS_ADMIN` Perform various system administration operations. The flawed design of this capability became apparent only later, when it contained large amount of security privileges. It is not advised to for kernel developers to use this capability for new privileged features of the kernel.

`CAP_SYS_PACCT` Allows to disable/enable process accounting. This is an example of a capability that is too specific and it should have been merged into other capability.

2.2.4 Backward compatibility of `setsuid()` operations

As mentioned earlier, one of the requirements for capability system was to provide backwards-compatible behavior that would preserve the original semantics of standard UNIX user–superuser. We already mentioned one of those behaviors in section 2.2.2, when threads running under root automatically consider file capabilities to be all present in permitted and effective set. In this section, we present remaining backwards-compatible behaviors. For additional information, see the capability(7) manual page [20].

1. If any of the real, effective, or saved set user IDs was zero and after the set-id operation all of these IDs have non-zero value, then effective, permitted and ambient capability sets are cleared. This represents dropping of privileges when switching to an unprivileged user.

2. If the effective user ID is changed from zero to a non-zero value, the effective capability set is cleared.
3. If the effective user ID is changed from a non-zero ID to zero, the permitted set is copied into the effective set.
4. If the filesystem user ID is changed from zero to non-zero, then selected capabilities[3] concerning the DAC are cleared from the effective set. Capabilities are copied back into the effective set from the permitted set when the filesystem user ID changes to a zero value.

2.2.5 Towards capability-only system

The purpose of capabilities is to make set-user-id-root flag obsolete. To achieve this, file capabilities have to be included with all executable files in the system and the backwards-compatibility behavior of capabilities has to be disabled. To help make this change gradual, Linux includes following per-thread *securebits* flags to disable certain backwards-compatible capability behavior:

`SECBIT_KEEP_CAPS` Setting this flag will stop the kernel from clearing capability sets when a thread switches all of its UIDs to non-zero values (assuming at least one of them was zero before). Note that this just concerns behavior 1 from the previous section. This means that effective set will be still cleared when a thread switches its effective UID from zero to a non-zero value.

`SECBIT_NO_SETUID_FIXUP` This is a superset of the previous flag and prohibits the kernel from making changes to thread's capability sets when UID changes.

`SECBIT_NOROOT` Setting this flag will cause lines 1–7 of algorithm 1 not to be applied, meaning that executing a new program under root or with set-user-ID-root will not automatically set filesystem capability sets so that the thread gets all existing capabilities.

`SECBIT_NO_CAP_AMBIENT_RAISE` Setting this flag disables the ability to raise ambient capabilities.

[3] These capabilitie are: `CAP_CHOWN`, `CAP_DAC_OVERRIDE`, `CAP_DAC_READ_SEARCH`, `CAP_FOWNER`, `CAP_FSETID`, `CAP_LINUX_IMMUTABLE`, `CAP_MAC_OVERRIDE` and `CAP_MKNOD`.

2.3 Mandatory Access Control

Near the end of the 90s, it became evident that access control mechanisms of Linux are inadequate to provide strong security. First projects that enhanced access control mechanisms in Linux relied on system call interposition or required a patch that inserted decision functions to appropriately selected locations, following the complete mediation principle. Medusa, SELinux, grsecurity and other security solutions were originally implemented as such patches.

2.3.1 Linux Security Modules framework

First security module that was considered to be included in the kernel was SELinux. Linus Torvalds[4] requested that instead of implementing a one security module, the kernel should be modified to allow user to choose which security module should be activated. So he asked kernel developers to create a generic framework that would allow any module complying with the API to be loaded and used as a security enhancement. This lead to the creation of the Linux security modules framework [24].

This framework allows security module to intervene operations in critical kernel sections, usually during system calls. This is thanks to a careful placement of hook function right before kernel provides access to some resource. Security modules can place callback functions to these hooks. Every callback in a hook has to be called — this allows each security module to evaluate the access and decide whether it follows enforced security policy. For the operation to be permitted, all security modules have to allow it. Denial from one security module suffices to deny the whole operation. As of Linux kernel v6.2, LSM provides 247 security hooks.

There are two types of hooks in LSM:

security hooks These are actual security hooks used to decide access requests to resources. They are located in places of code where user space subjects are about to access a kernel object. Security module has to decide if the access is granted.

control hooks These hooks serve purpose of notifying security modules about important events in the system that may concern them. Most notable examples are *alloc* and *free* hooks. These hooks can be used by the security module to allocate and initialize (or free in the case of a *free* hook) a security blob (see below). These hooks are usually called when a new kernel object is created or freed, respectively.

[4]the creator and current maintainer of the Linux project

Another important feature of LSM are the **opaque security fields**, also called security blobs. These are pointers to data structures that are used by the security module for storing information about the system object. It is responsibility of the security module to allocate and free memory for these data structures. As explained above, control hooks are used for this purpose.

Even though LSM was designed to be generic, there are still some disadvantages of the framework. For example, if the operation is denied by the DAC privileges, there is no way for LSM to override this decision (for the full evaluation path, see section 5.4). In other words, LSM is a restrictive framework, not permissive[5]. Another disadvantage is that critical sections, where the security modules can decide are fixed by the LSM framework and programmer can't choose arbitrary location of security module intervention.

There are still some security modules distributed as a kernel patch because of these problems. Grsecurity[6] and RSBAC[7] are notable examples.

2.3.2 SELinux

SELinux was the first security module included in the mainline kernel in December 2003. Originally developed by NSA, it is now maintained by Red Hat. It is enabled by default in Fedora, RHEL, CentOS, Android and available in other distributions as well.

SELinux provides most advanced coverage of access control out of security modules available in the Linux kernel. This makes it very powerful to be able to implement many types of security models, but it's also harder to manage than other security modules.

Architecture

SELinux implements Flask architecture (Flux Advanced Security Kernel) that was developed by the Utah university and the US Department of Defense [25]. SELinux is composed of following components:

Object manager manages available operations of objects and enforces security policy decisions. SELinux implements default object manager of kernel objects. It is also possible to implement a user space object manager that can manage objects not known by SELinux and then apply security policy decisions on these objects.

Access vector cache stores decisions made by the security server to make repeated security decisions faster.

[5]Synonym for "permissive" that is commonly used is "authoritative".

[6]`https://grsecurity.net/`

[7]`https://www.rsbac.org/`

Security server makes decisions based on the current security policy.

Security model

SELinux uses labeling approach to system security. Every kernel entity has assigned a label and policy decisions are made based on contents of these labels.

SELinux provides multiple types of mandatory access control that may be used simultaneously. In this section, we explain available access control types.

Security context SELinux labels subjects and objects in the system with security contexts. Security decisions are based on the contents of this label, class of the object, type of operation and context of the subject. Label is represented by a variable length string of form: `user:role:type[:mls/mcs]`, where `mls/mcs` is an optional field. Explanation of fields follows:

SELinux user Linux users are *mapped* to SELinux users. Based on a configuration file, once the user logs into the system, they get assigned a SELinux user. This user is immutable, user cannot change it once it has been assigned. One SELinux user might be used by many Linux users.

SELinux role Roles are used in the role-based access control model. Role contains a list of permissions and can be assigned to any SELinux user, if it is contained in a set of roles available for that user. Role is mutable, meaning the user can change roles in a session, but only one role can be active at a time.

SELinux type Type is the most important field of the security context, as it covers most of the access control decisions. That is to say, most of the rules specified in the security policy are concerned with the type of the subject, object and type of the access [26].

Type in a process security context determines which objects can be accessed by that process. SELinux types of processes are also called *domains*. Type in a object security context (file, socket, pipe, . . .) defines access permissions which a SELinux user has to that object.

MLS/MCS This optional field specifies either security level or security range. Security level is in the form of `sensitivity[:category]`, where `sensitivity` represents object classification and optional `category` represents compartment of the object. The range specification is specified as `sensitivity[:category]-sensitivity[:category]`. These fields are used in multi-level security model. For more information, see section 2.3.2.

Type enforcement Type enforcement is the primary MAC model of SELinux. Policy rules in this model consist of three parts:

- Subject of the operation,
- Object of the operation. It is defined by its *type* and *class* (see below).
- Type of the access.

This triplet is also called an *access vector*. Each time an access control is requested, a new triplet is created and searched in the active security policy. If it's found, the request is allowed. Access vectors not contained in the policy are denied.

Object classes Classes are used to differentiate policy rules on objects of different types (not in a sense of SELinux type but by the kernel type). For example, classes are used to distinguish rules applied to files with some label from sockets with the same label. In other words, files labeled `lib_t` have different privileges than sockets with the same label.

Classes are defined by the SELinux implementation and the Linux kernel, they can't be defined by system administrator.

Permissions Permissions are supported accesses that can be performed by subjects [27]. For each resource class (class of the object), SELinux defines a set of permissions that is supported by that class. Supported permissions can be displayed using the `selinuxfs` pseudo file system.

If some access is not mentioned, then it's not known to the Linux kernel or not yet supported by SELinux. Policy can be configured to either allow or deny unknown permissions.

Role-based access control For explanation of RBAC, see section 1.4.1. To correctly set up RBAC in SELinux, following steps must be followed:

1. Linux accounts must be mapped to SELinux users.
2. SELinux user is allowed one or more roles (this achieves segregation of duties).
3. Roles have accesses to certain domains (types and classes of subject).

Multi-level security For the explanation of the MLS model, see section 1.3.1.

Each process may define two sensitivities — one at which it is running and another one which is the maximum reachable sensitivity of objects.

MLS also provides fine-grained approach to security on top of sensitivity layers. Second part of MLS is the category system. A set of categories is defined for each subject that the subject is allowed to access. Unlike sensitivity layers, categories are not hierarchical. They are compartments used to divide objects into groups (for example based on departments).

Sets of categories are also defined for objects. Access to an object is allowed if the categories of subject and object match and sensitivity of the object is in the range of sensitivity of the subject.

Multi-level security is optional. In the case it is turned off, the fourth field of the security context is empty.

Configuration of the security policy

Security policy for SELinux is usually not written by administrator from scratch. Instead, each new policy is based on the *Reference Policy* [28]. It is a repository containing various policies that may be configured to suit administrator's needs. Distribution maintainers contribute to the reference policy project where their contributions are peer reviewed so that changes to the reference policy don't break security policy on other distributions.

Reference policy is a modular system. One module should contain access control policy for one application or a group of related applications. Module also contains private and shared resources, information about labels and definitions of interfaces used to communicate with other modules.

2.3.3 TOMOYO

TOMOYO was the third security module included in the mainline kernel in June 2009.

TOMOYO was originally developed as a series of kernel patches and thus didn't implement its interface on top of LSM. This version is now known as 1.x and it is supported by kernels v2.4 and v2.6.

Authors of TOMOYO wanted to include it into mainline kernel and created a new version that used hooks provided by the LSM framework. This version is known as 2.x and is a part of mainline Linux kernel. However, as it is restricted by the LSM framework, it doesn't include all of the functionality of the 1.x version.

There is also a third version of TOMOYO called AKARI. It is based on the 1.x version of TOMOYO, but uses the LSM framework to implement its security features. It is distributed as a kernel module, so end users have to compile just the module, not the full kernel with patches applied. AKARI has fewer features than 1.x, but more then 2.x.

In this overview, we will focus on the 2.x version, since it is part of the mainline kernel.

Security model

TOMOYO differs from other security modules in its security model. It doesn't use the standard subject-object entity relationship. Instead, it focuses on the behavior of processes. Process in TOMOYO is called a *domain*. Every process in the system belongs to some domain. Domains are stored in a tree structure based on the execution history of corresponding processes. Root of the domain tree is called `<kernel>`. First process in the system is usually the *init* process. This means that the first domain in the system for the *init* process is identified as `<kernel> /sbin/init`. Once the *init* process starts executing other processes, new domains will be created by appending the path of the executable to the domain of *init* process. Creation of a new domain is called a *domain transition*. This means that there might be two processes with the same executable that have different domains, because they were executed in a different context.

Domain transitions Since TOMOYO is focused on processes, it offers extensive management of domain assignment to new processes. As denoted in the previous section, creation of a new process is called *domain transition*. This section explains various ways of managing domain transitions. For clearer understanding, it also includes configuration snippets. For the full reference, refer to the TOMOYO manual [29].

Rule `initialize_domain` Normally, domain of the new process will be derived from its parent — parent domain will be a prefix of the child domain. However, this behavior might not be always desired. Some applications need to have the same restrictions no matter which process executed them. Administrators can use rule `initialize_domain` to set this behavior. For example, following directive in the *exception policy*:

```
initialize_domain /usr/sbin/sshd from any
```

will cause all newly executed instances of `sshd` program to switch to domain `<kernel> /usr/sbin/sshd` upon their execution.

This directive can also be used to choose any domain when executing some program from other domain. Instead of `from any`, administrator can define from which domain the domain transition to `<kernel> /usr/sbin/sshd` will occur. Rule

```
initialize_domain /usr/sbin/sshd from <kernel> /etc/rc.d/init.d/sshd
```

will cause `sshd` program to switch domains only when executed from the specified domain.

Domain specification doesn't have to be complete, it is possible to use just a part of the domain name from right side (at least one full path). For example,

instead of `<kernel> /usr/sbin/sshd`, one can use `/usr/sbin/sshd`. This allows creation of more general rules that apply to more domains disregarding their origin.

Rule `no_initialize_domain` This rule is useful when exceptions to rule from the previous section are desired, i.e. domains shouldn't switch when starting some program with `initialize_domain *** from any` rule. For example, following rule is in the exception policy:

```
initialize_domain /usr/sbin/sendmail.sendmail from any
```

Application `/bin/mail` executes `/usr/sbin/sendmail.sendmail` to send mails. If you want to do a normal domain transition when starting `sendmail` from `mail`, you can use following rule:

```
no_initialize_domain /usr/sbin/sendmail.sendmail from /bin/mail
```

Rule `keep_domain` In other security modules, when a new process is starting, it usually inherits security context of its parent (if the module is not configured to do otherwise). This behavior is also supported by TOMOYO.

Suppose that the same policy (and hence domain) is needed for all applications executed by the `sshd` daemon. This can be configured using the following rule: `keep_domain any from <kernel> /usr/sbin/sshd /bin/bash`. This rule can also be generalized to all `bash` instances like this: `keep_domain any from /bin/bash`.

The rule can also specify domains that will be kept. For example, rule `keep_domain /usr/bin/xargs from /bin/bash` will cause program `xargs` to run under the same domain as `bash`, but every other program executed under `bash` will undergo domain transition.

Directive `initialize_domain` has higher precedence than `keep_domain`.

Rule `no_keep_domain` This rule has the same meaning to `keep_domain` as `no_initialize_domain` has to `initialize_domain`. It defines exceptions to a previous `keep_domain` rule.

Domain policy Domain policy is a list of actions that may be performed by the domain. If the action is not on the list, it normally may not be performed, but that depends on the profile used.

TOMOYO supports access control of following actions:

file actions TOMOYO supports standard file operations that correspond to file LSM hooks. Complete list of supported file operations [30] follows: `execute`, `read`, `write`, `append`, `getattr`, `create`, `unlink`, `chown`, `chgrp`, `chmod`, `mkdir`, `rmdir`, `mkfifo`, `mksock`, `mkblock`, `mkchar`, `symlink`, `truncate`, `link`, `rename`, `ioctl`, `mount`, `unmount`, `chroot`, `pivot_root`.

network actions TOMOYO supports access control on both network sockets and UNIX domain sockets. Administrator can specify to which address (and port in the case of network sockets) the socket may perform operations. Supported operations are: `bind`, `listen` and `connect` for streaming sockets and UNIX sequential packet sockets, `bind` and `send` for datagram and raw sockets.

Exception policy Exception policy is a list of actions that may be performed by any domain, it is applied globally to all domains.

2.3.4 AppArmor

AppArmor was the fourth security module to be included in the mainline kernel in October 2010. So far it is also the last major security module included.

AppArmor is one of the most popular security modules, it is enabled by default in SLES, openSUSE [31], Ubuntu [32], Synology's DSM, Solus [33] and also Debian starting from version 10 [34].

Similar to TOMOYO, AppArmor is focused on task-based security. Processes in AppArmor are confined by objects they can access — files, network connections, capabilities and others. List of process privileges is called a **profile**. Main difference between TOMOYO and AppArmor is that while TOMOYO creates domain for every running process, AppArmor manages only processes with assigned profiles. Processes without profile are running *unconfined*, i.e., they are not controlled by AppArmor.

Partition into profiles allows easy deactivation of a misbehaving profile without disturbing rest of the mandatory access control. Similar approach can be found in SELinux's *Reference Policy*, where every application has its own module that can be activated or deactivated at run time.

2.3.5 Smack

Smack was the second security module included in the Linux kernel, in February 2008.

It's goal is simplicity. Contrary to SELinux, which provides multiple security models with highly customizable configuration, Smack provides simplified mandatory access control. Simpler configuration makes it a popular choice for embedded Linux distributions. For example, it is used by the *Tizen* mobile operating system [35].

Security model

For its basic security model, Smack uses a labeling system. Labels for subjects and objects are represented as regular ASCII strings. Their size is limited to 255 characters by definition, but developers of Smack recommend to use

twenty-three characters at most. One-character non-alphanumeric labels are reserved for the Smack development team. Currently, there are five one-character labels that serve a special purpose.

Security decisions are made according to these rules:

1. Any access requested by a subject labeled `*` (*star*) is denied.
2. A read or execute access requested on an object labeled `^` (*hat*) is permitted. This label is useful for backup software that should have read access to any file on the system.
3. A read or execute access requested on an object labeled `_` (*floor*) is permitted. This is useful for system libraries, root of the file system, or any other file that should be available for everyone to read or execute.
4. Any access requested on an object labeled `*` is permitted. This is more permissive version of the `_` label. It should be set on files which everyone can write to, in addition to reading and executing, for example `/dev/null`.
5. Any access requested by a subject on an object with the **same** label is permitted.
6. Any access requested that is **explicitly** defined in the loaded rule set is permitted.
7. Any other access is denied.

For file system objects, Smack closely follows traditional DAC permission types. To open a file for reading, a *read* access permission is required on the file. To search a directory, an *execute* access is required. Creating a file with write permission requires both read and write access on the containing directory. Sending a kill signal is a write access from the sender to the receiving process.

For sockets, sending a packet from one process to another requires a write access of sender to the receiver. Receiving process doesn't need to have a read access to the sender, since receiver is not the executer of this operation.

Explicit rules

Implementing a basic separation of privileges using basic labeling system is easy. However, there may be special cases, where limited access by subjects to objects with different labels is needed. An example of this is the Bell-LaPadula model. Hence, Smack allows explicit definition of access rules between two labels. Rule format definition is:

`subject-label object-label access`

Accesses are defined by single letters. These are written together, with no space between them in any order. Smack supports standard access types: `r` (read), `w` (write), `x` (execute), `a` (append) and two special types that are described below.

t rule requests transmutation. Normally, when a process creates a new file, this file gets the label of the process. However, if the parent directory of the file is marked as transmuting and an explicit rule for the label of the process (as subject) and the directory (as object) exists with the `t` access type set, then the new file will have the label of the *directory* instead of the process.

b rule should be reported for bring-up. This means that any rule marked with `b` will be logged on a successful access using that rule if the bring-up mode is turned on. Bring-up mode is a special mode of operation that allows rules to be logged when they are applied. It is useful during creation of a new configuration.

2.3.6 Minor modules

Integrity

Integrity is comprised of a number of different submodules, including the Integrity Measurement Architecture (IMA), Extended Verification Module (EVM), IMA-appraisal extension, digital signature verification extension and audit measurement log support. [36] It doesn't use LSM hooks, it defines its own hooks that are called from the main security modules.

Its main purpose is to verify integrity of loaded programs and running processes. EVM protects integrity of file's security extended attributes using HMAC [37]. IMA maintains a list of hash values of executables and other sensitive files. It compares hash values when the files are executed or read. It can also use facilities provided by a specialized TPM chip, if present in the system [38].

Loadpin

Loading files into the kernel may present security risks. That's why Linux kernel provides options to cryptographically check loaded kernel modules and accept only those whose signature is valid. However, there may be situations where the authenticity of kernel modules is verified at the file system level. The device from which modules are loaded may be read-only, so tampering of files is impossible.

Policy of Loadpin security module is focused on loading modules from such file systems. First kernel module loaded causes originating file system to be *pinned* and further modules are inserted **only** from that same file system. After the file system is unmounted, no other modules are allowed to be inserted without rebooting the system.

Lockdown

Lockdown provides an interface for the system administrator to disable (*lock down*) various features of the kernel that would allow the user to modify it somehow. As of Linux v6.2, it supports 29 reason levels of lockdown operations. Sensitive operations in the kernel are allowed only if their level is lower than the current level set in the lockdown module.

Security levels offered by the lockdown module can be grouped into two categories. First one is integrity, that keeps kernel safe from userland modifications. Second one is confidentiality that denies accesses that would leak information from the kernel.

SafeSetID

SafeSetID is a minor LSM module that prevents change of process UID/GID values. It can also prohibit assigning capabilities to change these values (`CAP_SETUID`, `CAP_SETGID`). SafeSetID maintains a list of approved UID/GID transitions and only these are allowed.

Yama

One of the main security issues of Linux is that processes of one user may examine memory of other processes owned by that same user. This can be performed using the `ptrace()` system call which is used by debuggers and system call trace tools.

This means that once a process is compromised, attacker may compromise other processes of that user and gather more confidential data. For example, by compromising a web browser, attacker may read memory of `ssh-agent` and steal keys.

Processes themselves may disable `ptrace()` attachment to them by calling `prctl(PR_SET_DUMPABLE, 0)`. This is done by the `ssh-agent` and other programs that wish to be protected against this vulnerability. However, many programs don't do this step and are left vulnerable to `ptrace()` malicious attacks.

Access control of Yama security module focuses on restricting access to `ptrace()` system call to limited set of processes. The exact set of processes is determined by a `ptrace_scope` level kernel parameter. Yama defines four levels of `ptrace_scope`: [39]

0. **classic `ptrace()` permissions** This security level adds no restrictive access control to `ptrace()` system call. `ptrace()` is still available to execute on processes which run under the same user, are not privileged and have not disabled `PR_SET_DUMPABLE` option using `prctl()`.

1. **restricted `ptrace()`** With this setting, processes can `ptrace()` only their children or processes that explicitly set the PID of the tracer using `prctl(PR_SET_TRACER, debugger)`.

2. **admin-only attach** On this level, `ptrace()` may be executed only by processes holding the `CAP_SYS_PTRACE` capability.

3. **no attach** `ptrace()` is completely denied. Once this setting is set in kernel, it cannot be reverted without rebooting the machine.

BPF LSM

BPF LSM doesn't provide any security model out-of-the box. It just provides an interface for BPF programs so they can hook into LSM hooks. BPF programs are byte-code programs that are loaded from user space by a privileged user.

Capability

Linux capabilities (see section 2.2) are implemented as a minor LSM module. Although they were not designed as a separate module, their implementation is a good example of code refactoring, where code previously interwoven is extracted using a generic API.

2.4 Automatic policy creation

In this section, we present tools for creating new security policies for existing security modules in Linux. They are usually referred to as *learning modes* in the documentation. At the end of each subsetion, we also summarize key points notable to our goal in the dissertation thesis.

2.4.1 SELinux

Tools helping with the creation of a new security policy, for example `audit2why` or `audit2allow`.

if the system is running a custom application, the system administrator has to create a new policy module for that application. There are two ways of doing that. The administrator might write the policy files from scratch using the knowledge of what the program does. This requires a very good knowledge of the reference policy and also of the confined program.

The second approach, which might be used along with the first one is to change SELinux enforcing mode to permissive. This allows the program to function normally and all operations that would be denied are stored in the audit log. Now the administrator has to execute all operations of the program, so that everything needed will included in the finished policy. After this, the administrator needs to filter the output of audit log to include events just for the program in question and redirect it into the `audit2allow` script with switch `-M` to create a new policy module. Note that this doesn't contain file context. If new labels are needed, administrator has to create these manually. After the policy module was created, it can be loaded into SELinux using `semodule -i`.

2.4.2 AppArmor

Following tools can be used to generate new profiles or update existing ones:

`aa-autodep` Generates minimal AppArmor profile for one or multiple executables. It doesn't perform complete static analysis of an executable, so that resulting profile might not be complete, but it's a good starting point for creating a new profile. Internally it works by calling `ldd` and generating a profile based on linked libraries [40].

`aa-logprof` Interactive tool that can be used to modify existing profiles that are running in complain mode. User will be presented with denied accesses which can be added to the policy, denied in the profile, or they can manually create a new globbing rule for detected path. Globbing rule can also be automatically generated from the last part of the path, if user selects that option.

Similar tool for generating file globbing rules can also be found in TOMOYO user space utilities suite (see 2.4.3).

`aa-genprof` Interactive utility that executes new process in complain mode, detects all events and allows user to store generated rules in a new profile.

AppArmor allows policy to be developed one process at a time. Administrator starts the procedure by running the `aa-genprof` command with the name of the executable to be confined. This creates a new profile and sets it to complain mode. The application can then be executed and all of its legitimate features have to be used, so that the final policy contains all necessary permissions. After all events are logged, the administrator presses `S` to scan the system log using `aa-logprof` utility.

This utility then interactively presents all logged events and provides various options. In the case of starting a new process, it asks if the new process should inherit the current profile, use a new profile or a subprofile.

AppArmor also provides suggested glob rules and abstractions. An abstraction provides a reusable set of access rules grouping together multiple resources that are commonly used together [41]. Administrator can further specify the exact type of the added rule:

allow to add an allow rule to the policy

deny to add a deny rule to the policy, meaning it won't be logged next time

ignore to ignore the event, meaning it will be denied and logged in the enforcing mode

new user has to enter new globbed entry to include the path

glob globs the last element of the path `/*`, which includes all of the files in the directory

2.4.3 TOMOYO

Automatic policy creation from system call logs was TOMOYO's design goal. When creating a new policy, administrator has to use the policy editor to analyze a behavior of an application. Standard process for this goes as follows:

First, the application to be confined has to be running. Then, a new domain for the application should be initialized using the `initialize_domain` directive. This is to make sure that the same domain will be used for this application no matter how the application was executed. Then the profile for this domain should be set to learning mode. TOMOYO provides standard profile with learning mode stored at position 1. After the profile has been set, application in question should be *restarted*, so that TOMOYO can detect behavior during shutdown and startup of the application and add it to the security policy. After the program is restarted, administrator should leave it running for some time, or execute all its features that might occur during its normal operation. It is important to do this step thoroughly, because all unrecorded behaviors will be denied after the profile is set to *enforcing* mode.

Once the behavior has been recorded, the policy can be stored on the hard drive, since now it exists just in the operating memory. Policy can be stored on hard drive using the `tomoyo-savepolicy` tool.

Memory requirements

Since TOMOYO stores a list of actions one process (domain) can execute, its memory usage can be quite high compared to other security modules. This usage is especially evident with learning mode.

Memory statistics are available in `/sys/kernel/security/tomoyo/stat` file. This interface provides summary of memory usage by each subsystem.

Patterning domain policy rules

Domain policy access rules in TOMOYO consist of full pathnames along with corresponding operations. However, it is not always possible to generate all possible pathnames a program can access. Temporary files are a perfect example. Their file name is usually a sequence of random characters prefixed with a constant string.

TOMOYO allows to create patterns in access rules to handle all possibilities. It offers a number of wildcard rules similar to regular expressions, but with special focus on often occurring patterns in filenames. For example, `\$` means 1 or more repetitions of decimal digits and `\A` means 1 or more repetitions of alphabet characters.

TOMOYO also offers extensive tools to work with patterning rules. One of them is user space tool `tomoyo-findtemp` which can be used to find potential temporary files suitable for patterning. Utility `tomoyo-patternize` uses rewrite rules from `/etc/tomoyo/tools/patternize.conf` to automatically generate patterns from domain policy generated from learning mode.

2.4.4 Smack

Smack doesn't provide any tools for automatic policy creation. Administrator has to manually label files and directories using the `chsmack` utility and create configuration files in `/etc/smack/accesses.d`. Smack doesn't support configuration file inclusion, but there may be more then one configuration file.

The only distribution that offers ready Smack policies is Tizen which focuses on embedded devices like TVs, wearables and mobile phones. Other desktop distributions don't contain Smack policy packages. Hence, the administrator has to create his own policy.

Chapter 3

Introduction to Medusa

Medusa is a security module for the Linux operating system that was developed at the Faculty of Electrical Engineering and Information Technology during the turn of the century by Marek Zelem and Milan Pikula [42, 43, 44].

3.1 Overview of the Medusa system

The complete Medusa system is defined by these parts:

1. Kernel module that represents the authorization module and partially policy store of a reference monitor. Architecture of Medusa allows multiple kernel modules that don't have to be a part of the operating system (e.g. a reference monitor implemented inside a database server). Currently there is one implementation for the Linux operating system integrated using LSM framework. Kernel module implements the Medusa security model (see section 3.2).

   Kernel module is structured into four layers. Lower layers are closer to the kernel and higher layers are closer to the authorization server.

2. Communication protocol used to transfer information from multiple kernel modules to an authorization server. This protocol is generic enough so that it's able to serialize any entity or programming object that needs to be examined for access control (see section 3.3).

3. Authorization server that represents the policy store component of the reference monitor. It is the main component that parses and stores access control policy. Functions $vsPermission_a$ and *vsMember* are defined by the authorization server based on the loaded policy. It is a requirement from the kernel module that any previously unencountered entity has to be sent to the authorization server for a process of labelling (assigment

of *virtual spaces*, see section 3.5.2. One authorization server can control one or several kernel modules.

3.2 Medusa Security Model

This section is based on an unpublished article by Zelem and Pikula [42], in which the security model of Medusa security module was presented under its original name — *ZP Security Framework*.

Terminology 1. The state of Medusa system is a tuple $(\mathcal{S}, \mathcal{O}, m, \mathcal{VS}, \mathcal{A})$, where $\mathcal{S}$ is a set of subjects, $\mathcal{O}$ is the set of objects, m is the access function that represents access matrix, $\mathcal{VS}$ is a set of virtual spaces and $\mathcal{A}$ is a set of access permissions that are recognized by Medusa.

Terminology 2. Let $\mathcal{S}$ be a set of subjects and $\mathcal{O}$ be a set of objects. We define a set of **entities** $\mathcal{E}$ as the union of $\mathcal{S}$ and $\mathcal{O}$, i.e., $\mathcal{E} = \mathcal{S} \cup \mathcal{O}$.

Virtual spaces are a key concept in Medusa that makes it specific from other security models. Virtual space groups together a set of related system entities. Note that one entity can be present in multiple virtual spaces (suggesting that the object can be accessed from different contexts). Entities in one virtual space have the same access rights (see below for detailed explanation). Such notion is similar to namespaces or containers.

Definition 1. Function $m : \mathcal{S} \times \mathcal{O} \to 2^{\mathcal{A}}$, where $\mathcal{S}$ is a set of subjects, $\mathcal{O}$ is the set of objects and $\mathcal{A}$ is a set of access permissions, assigns set of allowed permissions to pairs of subjects and objects. Subject $s \in \mathcal{S}$ can invoke access $a \in \mathcal{A}$ on object $o \in \mathcal{O}$ iff $a \in m(s, o)$.

Function m can be thought of as the function that gives contents of cells inside access matrix (see section 1.2.1).

Note that $\mathcal{S} \cap \mathcal{O} \neq \emptyset$. Subjects may pose as objects in some operations, for example when process sends a signal to another process.

Definition 2. Function $\mathit{vsMember} : \mathcal{E} \to 2^{\mathcal{VS}}$ returns a set of virtual spaces for a given entity, where $\mathcal{E}$ is a set of entities and $\mathcal{VS}$ is a set of virtual spaces. Entity $e \in \mathcal{E}$ is a member of a virtual space $v \in \mathcal{VS}$ iff $v \in \mathit{vsMember}(e)$.

Definition 3. Function $\mathit{vsPermission}_a : \mathcal{S} \to 2^{\mathcal{VS}}$ where $\mathcal{S}$ is a set of subjects and $\mathcal{VS}$ is a set of virtual spaces returns a set of virtual spaces that subject is allowed to act upon using access permission a. Subject $s \in \mathcal{S}$ can invoke access $a \in \mathcal{A}$ on object that is a member of virtual space $v \in \mathcal{VS}$ iff $v \in \mathit{vsPermission}_a(s)$.

In other words, the access is allowed if the object is a member of at least one virtual space that the subject has permission to access.

3.3 Medusa Communication Protocol

The Medusa communication protocol (MCP) is used to mediate communication between the kernel module and the authorization server. Since the authorization server manages the security policy, it needs to be aware of kernel status and events that change the state of the system. To inform about events and represent kernel objects, the communication protocol defines a way to serialize this data.

3.3.1 Data types

This section presents data types that are used in the MCP.

K-class K-class is used to internally represent a class of a kernel data structure. It consists of a definition of data (see **K-object** below) and pointers to functions that are used to invoke operations on the k-class. There is a constructor and destructor function, functions for fetch and update operations (explained below) and *unmonitor* function used to disable monitoring of operations of objects of the class. K-class design is indicative of an object-oriented design that is present in Medusa.

We will use `file_kobject` for code examples of data types. See listing 3.1 for definition of `file_kobject` k-class. `MEDUSA_KCLASS_HEADER` is a macro that connects this k-class with its k-object definition and definition of its attributes. See below for description of these objects.

```
MED_KCLASS(file_kobject) {
    MEDUSA_KCLASS_HEADER(file_kobject),
    "file",

    NULL,          /* init kclass */
    NULL,          /* destroy kclass */
    file_fetch,    /* fetch kobject */
    file_update,   /* update kobject */
    file_unmonitor, /* disable all monitoring on kobj. */
};
```

Listing 3.1: Definition of `file_kobject` k-class

Attribute type Attribute types are data types used to define attributes of k-objects. They specify a common data format used by the kernel module and the authorization server to exchange information about k-objects. There are also special key attributes that are used to find the kernel object to update (they serve as unique identifiers).

When updating an object, attributes that are not read-only are updated to attribute values sent by the authorization server.

K-object K-object internally represents data of a kernel structure. There are two views of a k-object: view of the kernel module, which may contain additional information for internal purposes and view of the authorization module which is determined by the attributes. Listing 3.2 contains the layout of the `file_kobject` structure. This is an example of a kernel-side k-object structure. It mimics contents of `inode` structure and contains only fields that are needed for the authorization server. Compare it with the definition of attributes of the same k-object in listing 3.3 — that is how the authorization server will see the k-object.

For each k-object, there are two important conversion routines, conventionally called `kern2kobj` and `kobj2kern`.[1] The first one is used usually by code in layer L2 before transferring kernel object to the authorization server. The second one is used usually by the `update` operation, i.e., when the authorization server updates values of k-object attributes.

```
struct file_kobject {
   unsigned long dev;
   unsigned long ino;

   umode_t mode;
   nlink_t nlink;
   kuid_t uid;
   kgid_t gid;
   unsigned long rdev;

   struct medusa_object_s med_object;
};
```

Listing 3.2: Definition of `file_kobject` structure

Event type Event types define structure of objects used to inform the authorization server about an event in the kernel. There are three main components of an event type: subject k-object, object k-object and attributes of the event itself.

Definition of `getfile` event is shown in listing 3.4. Starting from the top, it begins with a definition of a structure `getfile_event` that is used to represent the event in the kernel. Below it is a definition of

[1]Full names include name of the k-object as a prefix. For file k-object that is `file_kern2kobj` and `file_kobj2kern`.

```
MED_ATTRS(file_kobject) {
    MED_ATTR_KEY_RO(file_kobject, dev, "dev", MED_UNSIGNED),
    MED_ATTR_KEY_RO(file_kobject, ino, "ino", MED_UNSIGNED),
    MED_ATTR(file_kobject, mode, "mode", MED_UNSIGNED),
    MED_ATTR_RO(file_kobject, nlink, "nlink", MED_UNSIGNED),
    MED_ATTR(file_kobject, uid, "uid", MED_UNSIGNED),
    MED_ATTR(file_kobject, gid, "gid", MED_UNSIGNED),
    MED_ATTR_RO(file_kobject, rdev, "rdev", MED_UNSIGNED),
    MED_ATTR_OBJECT(file_kobject),
    MED_ATTR(file_kobject, user, "user", MED_UNSIGNED),
    MED_ATTR_END
};
```

Listing 3.3: Definition of attributes for file k-object

attributes used when transferring data to and from the authorization server and definition ends with macro `MED_EVTYPE` that defines the event itself. Name of its subject is `file` and name of its object is `parent`.

```
struct getfile_event {
    MEDUSA_ACCESS_HEADER;
    char filename[NAME_MAX + 1];
    int pid;
};

MED_ATTRS(getfile_event) {
    MED_ATTR_RO(getfile_event, filename, "filename", MED_STRING),
    MED_ATTR_RO(getfile_event, pid, "pid", MED_SIGNED),
    MED_ATTR_END
};

MED_EVTYPE(getfile_event, "getfile", file_kobject, "file",
      file_kobject, "parent");
```

Listing 3.4: Definition of a `getfile` event

When browsing source code of Medusa, one can notice that there are 5 event types defined.[2] These are used to notify authorization server about an event that is not a security access. Events starting with `get` are generated when kernel module encounters an entity that is not classified into virtual spaces. One of the tasks of the authorization server is to set the correct virtual spaces according to the identity of

[2]These event types are: fuck (files under critical kidnapping), getfile, getipc, getprocess and getsocket.

the entity and update it (see section 3.5).

There is also a special case of event types, called *access types*. In the MCP, there is no difference between event and access types. There is however an important semantic difference — access types represent actual security accesses. If such event is sent to the authorization server, it has to decide whether this access is permitted or not.

3.3.2 Operations

This subsection describes operations that are used by the authorization server to access or modify data in the kernel.

Fetch Fetch operation is used by the authorization server for getting k-object from the kernel. Authorization server has to create new k-object and fill key attributes. Kernel will use these attributes to locate the object and if found, it will be sent back to the authorization server.

Fetch is seldom used in the authorization server, as k-objects needed for an event are already included in the event.

Update Update operation is used by the authorization server to modify data of some kernel object by modifying attributes of its corresponding k-object.

As these operations are defined as methods of each k-class, they can perform any action inside the kernel. For example, `printk` k-class uses the update operation to print a message to the kernel log from the authorization server.

3.4 Kernel module

To explain the functionality of the kernel module, we will use the `mkdir` operation as an example. LSM path hook `path_mkdir` for this operation is registered on layer L1. Hooked function calls `medusa_mkdir` on L2. Functions of access types on L2 are the main place where the access decision is implemented. Necessary actions of such functions are shown in algorithm 2. Specific implementation depends on the access. For example, `link` needs to validate two objects and also check virtual spaces of these objects.

Procedure starts by checking validity of subject and object of the access. Validity means that the entity is classified into virtual spaces (i.e., it has its security context set). If the entity is not valid, a new `get` event is generated and sent to the authorization server. The task of the authorization server

is to assign virtual spaces to the entity according to the configured security policy.

After it is confirmed that both the subject and the object of the access have virtual spaces assigned to them, the intersection of the virtual spaces of the subject and the object is calculated. If there is no intersection, the access is denied without consulting the authorization server. Notice that this is similar to a pattern of UGO and MAC checks in Linux — if UGO denies the operation, MAC is not considered.

After virtual space intersection check, an access decision request may be generated and sent to the authorization server if the access is monitored. Whether the access is monitored is determined either from the subject or the object of the access. Which entity is used depends on the specific access.[3] For the `mkdir` operation, the object determines if the access will be sent to the authorization server (note that monitoring information has been assigned earlier by the authorization server together with classification into virtual spaces).

Algorithm 2 Typical processing of an access on layer L2

Input: *event*, *subject*, *object*
Output: 1 if *event* is allowed, 0 otherwise

1: **if** *subject* is invalid **then**
2:     *validate*(*subject*)
3: **end if**
4: **if** *object* is invalid **then**
5:     *validate*(*object*)
6: **end if**
7: **if not** virtual spaces of *subject* and *object* intersect **then**
8:     **return** 0
9: **end if**
10: **if** *event* is monitored **then**
11:     **return** *authserverDecide*(*event*, *subject*, *object*)
12: **else**
13:     **return** 1
14: **end if**

[3] This is a drawback in the Medusa design, which is planned to be removed in the future. However, its consequences are only performance-related and do not directly affect the policy mining.

3.5 Authorization server

The task of the authorization server is to assign virtual spaces to the entities according to the configured security policy. There are three authorization servers currently available: Constable (the original one), mYstable and Rustable. We will focus on Constable from now, as the other two are experimental and more suited to development purposes.

3.5.1 Unified namespace

The main data structure used by Constable is the unified namespace. It is a tree structure that unites several subtrees defined by the administrator in the configuration. Figure 3.1 shows a typical example of such a tree. There are two subtrees: `domain` and `fs`. Subtree `domain` contains processes. In this specific example, processes are assigned to tree nodes according to their category. Processes of PostgreSQL database daemon will be assigned under `pgsql` node and normal user processes will be assigned under `user` node. Notice that multiple processes can be assigned under one node. This applies to any subtree.

Subtree `fs` contains the filesystem as it's seen by the authorization server. Note that although it may resemble the filesystem structure of the protected machine, it doesn not get the structure from the actual filesystem. The structure is determined by the policy configuration only. Constable does not change the tree at runtime, so creating or deleting files is not reflected in the tree. Another example of a subtree in the unified namespace is the tree of roles in RBAC model (not shown in the figure).

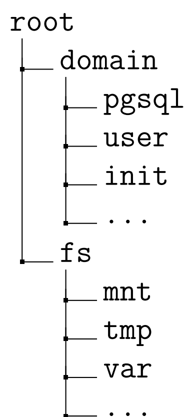

Figure 3.1: Tree structure of unified namespace in Constable.

3.5.2 Insertion into the tree

There are two basic ways of how an entity is classified into some node in the unified namespace. The first one is explicit classification. It is usually used when classifying processes. This is illustrated in listing 3.5. Line 5 defines `syslog` virtual space. Permissions for this virtual space are defined below on lines 6–10. Handler `fexec` on line 12 does the actual work of classifying a process into another domain. Access type *fexec* is invoked when process tries to execute a program file using the `exec` system call. In this case, the handler is invoked by any subject (denoted by a star `*`) and only one object — file located at `fs/usr/sbin/syslog` in the unified namespace.[4] Thus when some process tries to execute `syslogd` and it succeeds,[5] it is moved into `domain/syslog` node by the `enter_domain` function that is defined in the policy on line 1. This in turn classifies the subject into virtual spaces that are assigned to this node. In this case, that's the `syslog` virtual space.

```
1  function enter_domain {
2      enter(process, str2path("domain/" + $1));
3  }
4
5  primary space syslog  = "domain/syslog";
6  syslog     READ   tty, daemondev, temp, var, syslog, logs,
7             bin, etc, proc, home_public,
8         WRITE  tty, daemondev, temp, var, syslog, logs,
9         SEE tty, daemondev, temp, var, syslog, logs,
10            bin, etc, proc, home_public;
11
12 * fexec:NOTIFY_ALLOW "/usr/sbin/syslogd" {
13     enter_domain("syslog");
14 }
```

Listing 3.5: Snippet of Constable configuration for `syslog` domain

Second type of classification is automatic based on the position in the unified namespace tree hierarchy. It is executed by a special handler, see example in listing 3.6. This type of handler is also called a *hierarchy* handler.

The explanation of the handler now follows. The tree is called `fs` and it consists of `file` k-objects (`tree "fs" of file`). This handler is called when

[4]Curious readers may wonder why the root of the subtree `fs` is not present in the path in the listing. That's because Constable allows to label one subtree `primary` and all absolute paths will be automatically computed starting at the root of that subtree. This is normally used for the `fs` subtree so file paths may be defined naturally as absolute paths.

[5]The handler is executed after the operation is evaluated as allowed, due to the `NOTIFY_ALLOW` specifier.

```
tree "fs" clone of file by getfile getfile.filename;
```

Listing 3.6: Example of an automatic hierarchy handler in Constable

Constable receives `getfile` event from the kernel module (`by getfile`). Subject and object of the `getfile` event are important for the hierarchy handler to work. In this case, subject represents the file object that should be classified and object is parent folder of the subject that is already classified into some node.[6] Handler extracts node from the object due to `clone` handler modifier. Children nodes of the extracted node are search for a node with a name specified in the `filename` attribute of the event (`getfile.filename`). If found, subject will be classified into this node.

[6]Similar relation is also used for the `getprocess` event, where object is parent process of the subject.

Chapter 4

Related Work

This chapter introduces related work that concerns topics similar to this thesis. It can be roughly divided into two classes: **sandboxing**, that uses system call interposition to disallow certain processes from doing actions they are not allowed to and **policy mining** that pursues automatic creation of security policies for some security model from existing data, such as existing policy in a different security model or access logs.

4.1 Policy mining from logs

Molloy et al. [45] proposed an approach to producing usable RBAC models from usage of permissions. Such models reflect the actual pattern of usage and are therefore interpretable. Evaluation of the proposed algorithms shows improvement on previous approaches including exact mining, approximate mining, and probabilistic algorithms; the results are more temporally stable than exact mining approaches, and are faster than probabilistic algorithms while removing artificial constraints such as the number of roles assigned to each user.

Xu and Stoller [46] presented two solutions that are capable of creating ABAC policies from access logs. First one was an algorithm that worked by creating initial candidate rules from selected log entries and then iterating the rest of the log entries and generalizing and merging existing rules. Second one was implemented in the Progol inductive logic programming language. This work was the first published solution of such a problem.

Iyer and Masoumzadeh [47] proposed an approach to mine ABAC policies from access control logs that may contain both positive and negative authorization rules. They evaluated the approach using two different policies in terms of correctness, quality of rules (conciseness), and time. The algorithm outperformed previous approach to ABAC mining in terms of time.

Cotrini et al. [48] proposed an algorithm for mining ABAC rules from sparse logs that overcomes various problems that were present in previous solutions, such as overly permissive rules or large rules.

Karimi and Joshi [49] present a methodology for automatically learning ABAC policy rules from access logs in a system. The proposed approach uses an unsupervised learning-based technique for detecting patterns in a set of access records and extracting ABAC policy rules from these patterns. Two algorithms are presented, rule pruning, and policy refinement, to improve the quality of the mined policy. The proposed approach is evaluated on three different sample policies as well as a randomly synthesized policy.

The authors followed up with an unsupervised learning-based algorithm for detecting patterns in access logs and extracting ABAC authorization rules from these patterns [50]. In addition, they presented two policy improvement algorithms, including rule pruning and policy refinement algorithms to generate a higher quality mined policy.

Bui et al. [51] presented algorithms for mining ReBAC policies from information about entitlements together with information about entities. It presented the first such algorithms designed to handle incomplete information about entitlements, typically obtained from operation logs, and noise in information about entitlements. Two algorithms were presented: a greedy search guided by heuristics, and an evolutionary algorithm.

Cotrini et al. [52] presented a generalized method for policy mining supporting wide variety of policy languages, called universal access control policy mining (Unicorn). Supported models are ABAC, RBAC, RBAC with user-atribute constraints, RBAC with spatio-temporal constraints, and an expressive fragment of XACML. For the later two, no known policy miners were available previously. To design a policy miner using Unicorn, there are two requirements: a policy language and a metric quantifying how well a policy fits an assignment of permissions to users. Policy miners built on Unicorn were experimentally evaluated on logs from Amazon and access control matrices from other companies. Despite the genericity of the method, policy miners were competitive with and sometimes even beter than specialized state-of-the-art policy miners.

The problem that we set out to solve in this thesis is similar to these works in that they work with logs that might be incomplete. Thus we will need to deal with missing entries and control creation of over-assignments without compromising the system security. Other aspects, such as the security model (ABAC) is different from our pursuit. Since in our thesis we are devising a new system for automatic generation of security policies, we have to start with small steps and choose security model that is most suitable for operating system security, such as the domain model.

4.2 Automatic policy generation

One of the first works published about automatic policy creation is Polgen [53]. It is a tool for human-guided semi-automated SELinux policy generation. Polgen processes traces of the dynamic behavior of a target program. In that behavior, it observes instances of information flow patterns and based on these patterns, it creates new SELinux types and generates policy rules. The authors however deemed dynamic behavior as insufficient to determine security policy. Hence, Polgen uses a wizard-style interface for additional human input to the policy. They call this interaction "guided automatic policy generation."

Lachmund [54] presented a call graph based static analysis approach for application policy generation, which is augmented by an additional string analysis to identify user input propagating through the application's control flow, until it reaches permission checks. The main contribution of the method is that it distinguishes resource accesses initiated by the application from those initiated by the user and generates an application policy, which only contains access rights that are not derived from user interaction. The generated policy satisfies the principle of least privileges. User initiated accesses are handled separately at runtime.

Rauter et al. [55] provided a framework to automate generation of security policies and a proof-of-concept implementation that uses binary analysis to generate a model of the resource requirements of an application. They used a new approach to refine the policy by connecting different accesses to the same resource via their least common ancestor in the call graph. The proposed methods were tested with Nginx web-server and shown a high potential to significantly simplify the policy generation process.

Wang et al. [56] proposed EASEAndroid, an analytic platform for automatic policy analysis and refinement for SEAndroid, a SELinux implementation used in the Android operating system. It is not a fully automatic policy generator, as it needs an existing policy and a small set of known access patterns for operation. EASEAndroid dynamically processes new audit logs, producing suggestions for policy refinement. The solution was evaluated on 1.3 million audit logs from real-world devices discovering eight categories of attack access patterns.

Mocanu et al. [57] proposed work on the development of a deep learning technique to infer policies from logs. The proposal improves the state-of-the-art by supporting denied access requests and different types of noise in logs. It is based on restricted boltzmann machines.

In his dissertation, Sanders [58] explored the use of automated methods to create least privilege access control policies. Specifically, he developed a

framework for policy generation algorithms, developed two machine learning based algorithms for generating role based policies and developed a rule mining based algorithm to create attribute based policies.

Law et al. [59] presented a new ILP system, called FastLAS, that takes as input a learning task and a customised scoring function, and computes an optimal solution with respect to the given scoring function. While the system is generic and not specifically designed for generating access control policies, the authors have evaluated its accuracy on real-world access log datasets.

Jabal et al [60] proposes a framework for learning ABAC policies from data (inluding logs) named Polisma. It combines data mining, statistical, and machine learning techniques, capitalizing on potential context information obtained from external sources (e.g., LDAP directories) to enhance the learning process. The authors evaluate the approach empirically using two datasets (real and synthetic).

Li et al [61] proposed ASPGen, which is a framework for generating AppArmor security policies automatically. ASPGen can autogenerate security policy with the least privilege and RBAC (Role-Based Access Control) for applications, and effectively alleviate the complexity and subjectivity in manually configuring AppArmor's security policy, as well as the security threats that result from the improper policy. Inside ASPGen, a path-based confidence model for AppArmor is used to support the auto-generation of security policy. It is the first and so far the only public security framework supporting automated policy generation for AppArmor as far as we know.

4.3 System call interposition

Request decision can be made on various levels. The first one is on the level of a system call without considering its arguments. This is on the level of `seccomp` (without BPF). It's a good first step towards good direction, but consider that some system calls that might be necessary for the application might be misused by the attacker. Such system can't reliably manage system calls.

Second level is system call interposition with filtering based on the arguments of system calls. For example, seccomp-bpf uses bpf programs attached to seccomp system call filter. These programs may decide if the system call is allowed.

LSM works in a similar way, although it abstracts away system calls and instead uses the idea of hooks placed in the kernel when the kernel accesses some resource. Security modules can work with more information than simple syscall interposition — they can limit the access to an object based on the subject and object identity.

Golberg [62] built Janus, a secure environmment for untrusted helper applications. It used Solaris process tracing facililty. Pnacholi et al. [63] introduced TIMELOOPS, a novel technique for automatically learning system call filtering policies for containerized microservices applications. It used seccomp-BPF, systemd, and Podman containers.

4.4 Containerization and Sandboxing

One of the earliest works that considered sandboxing a program which receives untrusted data from the internet was by Goldberg et al. [64]. It intercepted and filtered dangerous system calls via the Solaris process tracking facility. System calls were allowed based on a configuration file that would have to be created by an administrator.

Another classic paper by Walker et al. [65] introduced domain type enforcement (DTE), a strong, configurable operating system access control technology that minimized the damage root programs could cause if subverted. The goals of the paper align with goals of LSM, such as protecting from attacks where malicious programs gain root privilege.

DTE was later implemented as LSM module to the Linux operating system by Hallyn [66] in his dissertation. Author implemented tools for creation and management of security policies, but no automatic policy creation was mentioned.

Acharya and Raje [67] presented MAPbox, a confinement mechanism for user applications. It is based on an idea to group application behaviors into classes based on their expected functionality and the resources required to achieve that functionality. However, the configuration in the form of allowed bahaviors had to be provided by the user of the application.

Fraser et al. [68] presented techniques for developing Generic Software Wrappers — protected, non-bypassable kernel-resident software extensions for augmenting security without modification of the application source code. The key elements of the work are: high-level Wrapper Definition Language, and framework for configuring, activating, and managing wrappers. Automatic generation of the wrappers was not discussed in the paper.

Ko et al. [69] built on the previous work to integrate intrusion detection techniques. Resulting implementation is a software layer dynamically inserted into the kernel that can selectively intercept and analyze system calls performed by processes as well as respond to intrusive events.

Provos [70] introduced an approach for application confinement that supports automatic and interactive policy generation, auditing, intrusion detection and privilege elevation and applies to both system services and user applications. Automatic generation of policies is provided by Systrace.

Jabłoński and Pawłowski [71] presented a sandboxing solution for GNU/Linux operating system. It didn't include facilities for automatic policy configuration.

Shan et al. [72] designed a MAC model incorporating intrusion detection and tracing in a commercial operating system, named Tracer. Their target users were common users who are not system experts. The model conceptually consists of three actions: detecting, tracing and restricting suspected intruders. One novelty is that it leverages light-weight intrusion detection and tracing techniques to automate security label configuration that is widely acknowledged as a tough issue when applying a MAC system in practice.

The other is that, rather than restricting information flow as a traditional MAC does, it traces intruders and restricts only their critical malware behaviors, where intruders represent processes and executables that are potential agents of a remote attacker. Experiments on Windows showed that Tracer can effectively defeat all malware samples tested via blocking malware behaviors while not causing a significant compatibility problem.

Chapter 5

Policy Mining

5.1 Problem Definition

The main objective of this thesis is to design and implement an algorithm that creates a functional security policy for the Medusa security module with minimal administrative intervention. This is a general objective that might be too overwhelming to achieve. To make it more focused and specific, we introduce these constraints:

1. Security policy will only take into account filesystem operations and accesses.

2. Creation of security policy will focus on limiting the set of available objects to a user or system application. This concept is similar to sandboxing. By limiting access of the application, we limit the *attack surface* of the application. This is useful for network daemons, as explained earlier in the thesis.

   In other words, our focus is on creating a policy for system services as opposed to creating a policy for individual users that use the system.

3. Created policy is static. This means that if the administrator wants to later update the policy, she has to run the policy mining algorithm again.

5.2 Research questions

The expected output of the thesis can be summarized in the following research questions:

1. If we construct an algorithm that would create security policy just from operation logs, how would it compare to an administrator-authored policy?

2. If we construct an algorithm that would create security policy from operation logs including some external information, how would it compare to an administrator-authored policy?

5.3 Solution Proposal

Our proposed solution takes inspiration from domain-type enforcement [65, 66], specifically it creates domains for each executed application. By observing activities of the application (on the premise that the application is not malicious), we can construct a list of objects that the application should have access to.

Brief summary of the proposed solution is:

1. Monitor operation of an application for which the security policy will be created.

2. Preliminary policy will be created for each subject (represented by an execution domain) based on the name of the object and requested operation.

3. Preliminary policy will be analyzed for missing rules that create underpermissions and additional rules will be added to the policy. We call this step *generalization*. After this step, a full usable policy for an application should be available.

In the following sections, each point of the proposed solution will be explained in more detail.

5.3.1 Basic Definitions

This section presents definitions that will be used later in the chapter.

Terminology 3. *Thread info* is a tuple $(exe, euid)$ that represents a running thread, where *exe* is the path to the program file and *euid* is the effective UID of the runnning thread.

Notation ti^{exe} represents the *exe* portion of the thread info tuple and ti^{euid} represents the *euid* portion respectively.

Terminology 4. *Domain* is a list of thread infos $[ti_1, \ldots, ti_n]$. ti_n contains thread info valid for the running thread. $ti_1, \ldots, ti_{n-1}$ represent the execution history of a thread.

Terminology 5. Domain transition is a change of domain for a particular process. There are three ways a domain transition can happen:

1. By calling `exec` system call. New domain of a thread will be $[ti_1, \dots, ti_{n+1}]$, where ti^{exe}_{n+1} is the path of the executed binary and $ti^{euid}_{n+1} = ti^{euid}_n$ (eUID remains unchanged).

2. When eUID is changed, for example by calling the `setresuid` system call. New domain of a thread will be $[ti_1, \dots, ti_{n-1}, ti_{n'}]$, where $ti_{n'} = (ti^{exe}_n, euid')$ and $euid'$ is the new euid.

3. By calling `exec` system call on a *setuid* binary. New domain of a thread will be $[ti_1, \dots, ti_{n-1}, ti_{n'}]$, where $ti_{n'} = (exe', euid')$, exe' is the path of the executed binary and $euid'$ is the new euid. Since setuid binaries are depracted in modern systems, we have not considered this way in our solution.

In all cases, $[ti_1, \dots, ti_n]$ is the original domain of the process.

Terminology 6. Policy rule is a tuple $(path, do, P', F)$, where $path \in PA$ is a string representing a path, do is the domain to which this rule applies, P' is a set of permissions that this rule grants and F is a set of flags that affects how the rule is applied when granting access (see section 5.3.2).

Definition 4. $dirname : PA \rightarrow PA$ is a function that for a given path returns path of its parent directory (PA is a set of file paths). It conforms to POSIX.1-2008.

For example, $dirname(\text{"/usr/bin/cat"}) = \text{"/usr/bin"}$.

Definition 5. $children : PA \rightarrow 2^{PA}$ returns a set of children paths of a given path, where PA is a set of paths.

$children(p) = \{q \in PA \mid dirname(q) = p\}$

Definition 6. $match : R \times PA \rightarrow \{0, 1\}$ returns 1, if regular expression from R matches path from PA, where R is a set of regular expressions and PA is a set of paths.

Definition 7. $prefix : PA \rightarrow 2^{PA}$ returns a set of all parent paths in the folder hierarchy, where PA is a set of paths. Example: $prefix(\text{"/usr/bin/cat"}) = \{\text{"/"}, \text{"/usr"}, \text{"/usr/bin"}\}$.

Definition 8. $matchPrefix : R \times PA \rightarrow \{0, 1\}$ returns 1, if $\exists p \in prefix(path) : match(regexp, p) = 1$, where $regexp \in R$ is a regular expression and $path \in PA$ is a path.

Definition 9. *recChildren* : $PA \to 2^{PA}$ returns all paths that are available under input path.

$$recChildren(p) = \bigcup_{p' \in children(p)} recChildren(p')$$

Definition 10. *numericRegexp* : $PA \to R$, where PA is a set of paths and R is a set of regular expressions, is a function that for a given path creates a regexp. This regexp keeps the path intact except for numeric characters. Each numeric character (or multiple characters if located next to each other) are replaced by `\d*`.

For example, *numericRegexp*("`/run/postgresql/.s.PGSQL.5432.lock`") = "`/run/postgresql/\.s\.PGSQL\.\d * \.lock`".

Definition 11. *unique* : $\mathcal{P} \to 2^{PA}$ returns a set of paths having multiplicity 1 in the input multiset, where $\mathcal{P}$ is a multiset of paths and PA is a set of paths.

Definition 12. Let $best(R', PA') : 2^R \times 2^{PA} \to R$ be a function returning $r \in R'$ that matches all paths in $PA' \subseteq PA$. If such r doesn't exist, it returns empty regexp.

5.3.2 Decision function in Medusa

Algorithm 3 shows how the decision function is computed for a specific access. How the access is computed depends on flags in the rule. There are two flags:

regexp Path in this rule is not matched literally, but as a regular expression.

recursive Rule applies to the specified path and all possible paths under it.

First, literal rules are applied (line 1). These rules target specific, literal paths within the file system. If a matching rule is found and its permissions cover the requested permissions, the access is allowed. If a matching literal path was not found, the algorithm searches for regexp rules (line 3). If that fails, search continues for matching recursive literal rules (line 5). The last rules searched are the recursive regexp rules (line 7). If no suitable rule was found, the access is denied.

5.3.3 Getting Logs

Input data for the algorithm should be complete, meaning that every system access should be logged without exceptions. Another requirement is that audit should be applicable only to a selected number of processes. This is mostly for performance reasons during development as logging every system call for every process might cause unacceptable decline of performance.

Algorithm 3 Computation of access

Input: access request $(do, path, P)$, policy Π
Output: $answer \in \{0, 1\}$
1: **if** $\exists P'' \in \{P' \mid (path, do, P', \emptyset) \in \Pi\} : P'' \supseteq P$ **then**
2: **return** 1
3: **else if** $\exists P'' \in \{P' \mid (path', do, P', \{regexp\}) \in \Pi \wedge match(path', path) = True\} : P'' \supseteq P$ **then**
4: **return** 1
5: **else if** $\exists P'' \in \{P' \mid (path', do, P', \{recursive\}) \in \Pi \wedge path \in recChildren(path') \cup \{path'\}\} : P'' \supseteq P$ **then**
6: **return** 1
7: **else if** $\exists P'' \in \{P' \mid (path', do, P', \{regexp, recursive\}) \in \Pi \wedge matchPrefix(path', path) = 1\} : P'' \supseteq P$ **then**
8: **return** 1
9: **else**
10: **return** 0
11: **end if**

Ideal way of getting logs on the Linux operating system is the audit system. It is able to log system calls and various security-related events in the operating system based on the settings provided by the `auditd` deamon.

Using contributions from [73], we modified Medusa security module to audit every hooked operation of a chosen process. Process to audit can be selected by a `fexec` handler in the authorization server configuration. Once Medusa-specific auditing is enabled for a thread, each hook call will create an audit record containing information about the thread, the operation, the object of the operation and any other useful information provided through the hook interface. This also applies to new processes forked by each audited process. Executing a new program file using `exec` system call doesn't change the audit state and process remains audited. Operations that are audited are listed in appendix A.

Listing 5.1 shows an example of operation as captured by the audit subsystem. It consists of three entries of different types. `AVC`[1] entries are

[1]The name `AVC` was taken from SELinux's *access vector cache* and it is a standard type for auditing LSM specific information, also used by AppArmor and Smack. During implementation of audit support for Medusa, we have closely followed SELinux's format. Not following the format may result in incompatibility with some system tools. For example, tool `ausearch` can't process AppArmor audit entries due to a long-standing bug, see `https://bugs.launchpad.net/ubuntu/+source/audit/+bug/1117804`.

generated from Medusa hooks, `SYSCALL`[2] and `PROCTITLE` entries are generated during exit from a system call. Value of `proctitle`[3] is encoded in Base16 binary to text encoding as *untrusted string*. If values may contain *unsafe* characters, they are represented in this encoding. Such representation can be distinguished from literal string by examining the first byte — literal strings start with double quotes (").

Audit system for Medusa was designed to generate `AVC` entries with useful information for the policy mining algorithm that are not present in other standard entries. For the `open` operation in the listing, that is the `file` value that specifies path to the file that was opened and `mode` that specifies the requested permissions on the file.

Preparation of this data directly in Medusa simplifies subsequent processing in policy mining. This advantage can be seen in the `SYSCALL` record, where the operation is represented by a number `syscall` that has to be mapped to a specific name (note that the `open` operation can be invoked by multiple system calls). The path to the open file is not possible to get from this record, since it is represented by a pointer to a string that no longer exists when the log is parsed.

```
type=AVC msg=audit(1679413684.051:1955): Medusa: op=open ans=ALLOW
    as_request=0 file="/var/lib/pgsql/data/global/1262" mode=6
type=SYSCALL msg=audit(1679413684.051:1955): arch=c000003e syscall=257
    success=yes exit=4 a0=ffffff9c a1=55bde45b6fc8 a2=2 a3=0 items=0
    ppid=20244 pid=20253 auid=4294967295 uid=26 gid=26 euid=26 suid=26
    fsuid=26 egid=26 sgid=26 fsgid=26 tty=(none) ses=4294967295
    comm="postmaster" exe="/usr/bin/postgres" key=(null)^]ARCH=x86_64
    SYSCALL=openat AUID="unset" UID="postgres" GID="postgres"
    EUID="postgres" SUID="postgres" FSUID="postgres" EGID="postgres"
    SGID="postgres" FSGID="postgres"
type=PROCTITLE msg=audit(1679413684.051:1955):
    proctitle=2F7573722F62696E2F706F73746D6173746572002D44002F76617...
```

Listing 5.1: Example of an event from audit log

After getting the audit log (for example by starting and stopping the service), we can assemble policy Π from the recorded accesses. The initial policy generated from the audit log will contain rules with literal paths. This process is presented in algorithm 4.

[2]Careful readers may notice `^]` characters inside the record. These represent the ASCII character “Information Separator Three” that separates raw numerical values from computed values.

[3]Value in the listing was shortened to fit on the page.

Algorithm 4 Creation of rules from audit log

Input: accesses A
Output: policy Π
1: $\Pi = \{\}$
2: **for all** $a = (do, path, P) \in A$ **do**
3: $\quad \Pi \leftarrow \Pi \cup (path, do, P, \{\})$
4: **end for**

5.3.4 Generalization

From the nature of the audit logs, we can identify the following problems that cause underpermission:

1. Based on the execution of the application, not all execution paths may have been executed and thus some accesses may not have manifested. These accesses will be denied once the policy will be enforced.

2. Accesses to temporary files or newly created files will refer to paths that were not captured in the original audit logs. Note that compared to the previous point, the access was requested, but the path is different in the next execution. The consequence is the same — after the policy is enforced, these accesses will be denied.

Solution to this problem is generalization — the policy mining module has to relax the generated rules so they will match a larger set of possible paths. This causes overpermission, which is undesirable. Policy mining has to solve an optimization problem — keep overpermission low while causing as few access misses as possible.

Generalization creates rules that apply to multiple paths which may not be present in the filesystem. This can be achieved using regexp and/or recursive rules. For example, rule ("`/var/log/pgsql/`$.*$", $do, P, \{regexp\}$) allows processes under domain do to execute operations that require permissions P on any file under `/var/log/pgsql` directory.

In the following subsections, we propose various generalization algorithms. We discuss their requirements and expected results. Experimental results along with testing methodology are then presented in section 5.4.

Tree coverage generalization

In this generalization, we assume that if all files in a folder have the same access permission, we can generalize this access permission for the entire content of the folder.

A new rule $(d \,\|\, "/.*", do, P, \{regexp\})$ is added for folder d if the following inequality holds:

$$\frac{|\{\, d' \in children(d) \mid (d', do, P', \{\}) \in \Pi \land P \subseteq P' \,\}|}{|children(d)|} \geq t$$

where do is the process domain, P is a set of access permissions, Π is a set of policy rules generated from the audit log and $t \in (0, 1]$ is a threshold constant that controls sensitivity of the generalization.

Note that this generalization takes into account only information from the audit logs. This means that if a process didn't access some path that exists in the filesystem, the generalization algorithm assumes the path doesn't exist.

We expect that this generalization will work well for services that store their files together in folders and they access all or most of these files. It won't work well for services that have many files in different folders and access them only sporadically.

Filesystem Hierarchy Standard generalization

This generalization takes into account standard hierarchy of folders in Linux systems as defined in File Hierarchy Standard [74] (FHS) and systemd's file-hierarchy [75] and Linux's hier(7) [76] manual pages.

For example, folder `/proc` contains information about running processes. This information is available under numerical subdirectories for each running process in the form of `/proc/<pid>`, where `<pid>` is the PID of the process. Files under `/proc/<pid>` provide various information about the specific process. Processes use these files to get information from the kernel about their current status in various Linux subsystems. For example, process may write to `/proc/<pid>/oom_score_adj` to adjust the heuristic used to select which process gets killed in out-of-memory conditions [77]. Such write may show up in the audit logs as write access to `/proc/1826/oom_score_adj`. As the `<pid>` portion of the path is variable, this results in a different path being generated each time the process runs. As a consequence, access to the path would be denied due to the inconsistency between the stored path in the policy and the actual path at runtime.

Similar reasoning can be used for wide-available files, such as libraries in `/usr/lib64` or binaries in `/usr/bin` that should automatically get *read* permission for all processes.

The goal of this generalization is to employ standard paths defined in FHS to automatically generalize dynamic parts of accessed paths. Note that there are Linux distributions such as NixOS [78] or GoboLinux [79] that don't utilize FHS and provide their own file hierarchy structure. For these systems, this specific generalization cannot be used, but administrators can

easily change the configuration file with standard permission to suit their system.

Generalization based on non-existing files

This generalization is based on two path sets. Path set PA' is created from the real filesystem before the service(s) for which the policy is mined are started. Path set PA is derived from access set A created from the audit log (see section 5.3.3).

Algorithm for this generalization takes every path from PA' that is not present in PA, computes its parent directory and applies read and write permission to it,[4] see algorithm 5.

Algorithm 5 Generalization based on non-existing files

Input: path set PA', domain do, policy Π
Output: policy Π
1: $PA = \{path \mid (do, path, P) \in A\}$
2: **for all** $p \in PA' \setminus PA$ **do**
3: $\quad \Pi \leftarrow \Pi \cup (dirname(p) \parallel \text{"/.*"}, do, \{R, W\}, \{regexp\})$
4: **end for**

This method compares accessed paths in the logs with a filesystem snapshot. If the accessed path doesn't exist in the snapshot, we can proclaim that this path is dynamic (as opposed to static files) and its name may be dynamic.

This generalization method is just supplementary with a specific focus on non-existent files, it cannot provide general generalization. Therefore it is expected that it might improve performance of other generalization algorithms, such as the tree coverage generalization, when used together.

Generalization based on UGO permissions

This generalization relies on external information stored on the filesystem. Most of the services on the Linux system are assigned a special user ID. These IDs are used as eUIDs when a service is running, but also as owner and group owner IDs of files associated with the service. We can use this information when constructing the policy. We propose four generalization strategies that can be used independently:

1. **Generalization by directory owner UID** Access to directories that

[4] *Write* permission to the parent directory is applied automatically after loading to audit log, since to create a file in the directory, process has to have a write permission to that directory.

are owned by eUID of the domain can be generalized so that the domain gets privilege to access (read/write) any file in those directories.

Rationale for this strategy is that it *approximates* standard UGO permissions. Namely, if a user owns a directory, he can access all files in this directory as well with high certainty.

Note that this strategy can't be used for *root* as most configuration directories in the system have zero UID owner and there is no special distinction accross various subsystems (for example, on a standard Debian system, folders `audit`, `alsa`, `dpkg` and `mysql` in `/etc` have all zero owner UID, but their purpose and importance varies — reading passwords from `mysql` configuration may have more dire consequences than changing configuration of `alsa`).

2. **Generalization by file UID** This strategy is similar to the previous one, with the difference that it considers files *inside* a directory. For an access to any file in a directory to be generalized, all accessed files have to be owned by the effective user of the running domain.

   Rationale is the same as in the previous strategy, but the heuristic is different. Root user is again ignored for the same reasons.

3. **Generalization by read access to files** If the directory contains items that are readable by the effective user of the process, read access to files in this directory can be generalized. This considers computation of DAC permissions according to UGO permissions of each file (as explained in section 2.1.1).

4. **Generalization by write access to files** If the directory contains items that are writable by the effective user of the process, write access to files in this directory can be generalized similarly as the read access.

Generalization based on owner directory

Unlike *generalization by directory UID*, this generalization relies just on external information and not on information from audit logs. Inputs to this generalization are: set of file paths PA' obtained from the real filesystem, set of user UIDs U, set of group IDs G and a set of domains D.

U and G contain IDs that are associated with the audited service (for example, the service daemon runs with $euid \in U$ and $gid \in G$). Similarly, domains in D are associated with the service, i.e., service daemons run under these domains. Generalization algorithm searches for folders that match provided UIDs or GIDs. Files inside these folders are then generalized for read and write access.

We presume that this generalization will achieve the best results, since it relies entirely on external information. However, it is not usable for services that don't run under some specialized user.

Generalization based on multiple runs

This generalization method takes advantage of the fact that temporary files change names across executions of a service. Thus we can start a service multiple times, get audit log information from each *run* and compare them. Paths that are unique across all runs can be considered to be ephemeral and can be generalized accordingly.

Interesting problem is the generalization of the paths. They usually consist of static and dynamic parts. The problem lies in identifying these dynamic parts and providing generalization of them.

Algorithm 6 shows this generalization. First, a multiset union of path sets from all runs is computed and unique paths are extracted (line 1). Function *group* groups paths according to similarity computed using cosine similarity of TF-IDF N-gram vectors (line 2). This is implemented using external library [80]. Each group is then processed separately. Paths that have not been grouped (a group that contains one path) are processed on lines 6–10. There are two ways how a regexp can be created from this path. Using *numericRegexp* function (see definition 10) that replaces numeric characters or a full regexp that replaces name of the file by `.*?` matching any other file in its parent directory.

When more paths are grouped together, a regexp is computed that matches all paths in the group. The first approach is to is to select one *control* path (line 13) and for each remaining path in the group, calculate the difference between the path and the control path. The result from the comparison algorithm is used as input to the *regexpFromDiff* function, which computes a regular expression based on the difference, see algorithm 7. Function *best* (line 18) returns the first regexp that matches all paths in P (see definition 12). If that fails (no regexp created from differences matches all paths in the group), function *prefixPostfixRegexp* (see algorithm 8) is called that searches for longest common substring in paths that are left by removing their common prefixes and suffixes. If this fails, a generic `.*?` regexp is used instead.

5.4 Evaluation

This section presents the evaluation of suggested algorithms. First, we introduce the methodology used to test our algorithms. Then, we present results from two tests: generating policy for individual services and generating policy for multiple services using the tree generalization algorithm (cumulative mining). For the individual services, we have evaluated four applications

Algorithm 6 Generalization based on multiple runs

Input: path sets $PA_1, \ldots, PA_n$, $type \in \{numerical, full\}$
Output: policy Π

1: $PA' \leftarrow unique(\bigcup_{PA=PA_1}^{PA_n} PA)$
2: $G \leftarrow group(PA')$
3: $\Pi \leftarrow \{\}$
4: **for all** $P \in G$ **do**
5:   **if** $|P| = 1$ **then**
6:     **if** $type = numerical$ **then**
7:       $\Pi \leftarrow \Pi \cup (numericRegexp(P[1]), do, \{R, W\}, \{regexp\})$
8:     **else if** $type = full$ **then**
9:       $\Pi \leftarrow \Pi \cup (dirname(P[1]) \parallel "/.*", do, \{R, W\}, \{regexp\})$
10:     **end if**
11:   **else**
12:     $R \leftarrow \{\}$
13:     $control \leftarrow p_1$
14:     **for all** $p \in \{p_2, \ldots, p_n\}$ **do**
15:       $d \leftarrow computeDiff(control, p)$
16:       $R \leftarrow R \cup \{regexpFromDiff(d)\}$
17:     **end for**
18:     $b \leftarrow best(R, P)$
19:     **if** $b \neq ""$ **then**
20:       $\Pi \leftarrow \Pi \cup (b, do, \{R, W\}, \{regexp\})$
21:     **else**
22:       $\Pi \leftarrow \Pi \cup (prefixPostfixRegexp(R, P), do, \{R, W\}, \{regexp\})$
23:     **end if**
24:   **end if**
25: **end for**

Algorithm 7 regexpFromDiff

Input: $DIFF$
Output: regexp $r \in R$

```
1: r ← ""
2: modifications ← []
3: for all (op, change) ∈ DIFF do
4:    if op = equal then
5:       if modifications ≠ [] then
6:          r ← r || ".*?"
7:          modifications ← []
8:       end if
9:       r ← r || escape(change)
10:   else if op = insert then
11:      modifications ← (op, change)
12:   else if op = delete then
13:      modifications ← (op, change)
14:   end if
15:   if modifications ≠ [] then
16:      r ← r || ".*?"
17:   end if
18: end for
```

Algorithm 8 prefixPostfixRegexp

Input: $PA' \subseteq PA$
Output: regexp $r \in R$

```
1: prefix ← commonprefix(PA')
2: postfix ← commonpostfix(PA')
3: common ← []
4: for all s ∈ PA' do
5:    common ← common || removeprefix(removepostfix(s, postfix), prefix)
6: end for
7: lcs ← longestCommonSubstring(common)
8: return prefix || ".*?" || lcs || ".*?" || postfix
```

that are standard components of a Linux server system. For the cumulative mining, we have evaluated three services in different combinations.

5.4.1 Methodology

Testing was performed on a Fedora Server 37 distribution with Medusa running on a 6.2 Linux kernel. Testing consisted of following operations:

1. System is booted with the Fedora kernel.
2. Filesystem snapshot is created before running any services, this results in a set of paths PA'. This set will be used when evaluating owner, owner directory and non-existent generalization algorithms.
3. System is rebooted with the Medusa kernel.
4. Constable configuration for a specific service is prepared and Constable is started (see `run_service.sh` script).
5. Service is started, it is left running for a few seconds and then stopped.
6. Constable is stopped and audit log is retrieved for analysis.
7. Steps 2.–4. are repeated to get an alternative audit log that will be used for multiple runs generalization.
8. System is rebooted with the Fedora kernel.
9. Policy mining is executed with specific test cases. These test cases are described in the following subsections.

Resulting mined policy is compared to the reference SELinux policy present in Fedora. This is done by comparing a specific set of paths (containing paths of files and directories) for two access types supported by both Medusa and SELinux: read and write. The evaluated set of paths consists of:

1. Literal paths from the policy Π that is created as a union of initial policy from audit log and generalized policies by individual generalization algorithms.
2. FHS generalization algorithm is automatically applied with a static policy for each case.

3. Paths determined by expanding regexp and recursive rules from policy Π applied on PA'. The role of these paths is to detect overpermission, since generalization rules cause overpermission.

   For example, if a rule in Π specifies regexp path `/etc/.*`, this takes all files located under `/etc/` from the filesystem snapshot PA'.

4. Paths that are associated with the service according to the SELinux reference policy. This has to be determined manually by consulting the reference policy. The role of these paths is to detect underpermission, as these paths will usually be accessible by the service according to the reference policy and the path may not have been captured in the logs or by any generalization rule.

   For example, for the Open SSH service, we get paths to files with SELinux types beginning with `sshd`, such as `sshd_exec_t`, `sshd_key_t`, `sshd_keygen_exec_t` and `sshd_keygen_unit_file_t`.

Resulting permission values are evaluated using standard binary classification techniques. Results can be classified into 4 categories:

hit (TP) Both mined policy and reference policy allow the operation.

overpermission (FP) Mined policy allows the operation while reference policy denies it.

underpermission (FN) Mined policy denies the operation while reference policy allows it.

correct denial Both mined policy and reference policy allow the operation.

Because of the evaluation methodology, absolute values of these four categories for each individual service can't be compared directly and a relative metric is needed. When searching for suitable metrics, we discarded metrics that determined the relative value from correct denials. This is because of the permissive nature of LSM — accesses that are not listed in the policy are automatically denied. Since we could put accesses to all the other files on the disk into correct denials and thus artificially inflate the metric, it is not usable for our purpose. Two basic metrics that are suitable to compare mined and reference policies are sensitivity (equation 5.2) and precision (equation 5.1). Note that sensitivity is more important since it represents underpermission — accesses that are not permitted cause denial of service. Overpermission, while undesirable, can be tolerated since it doesn't cause the program to stop functioning.

For a combined metric, we have chosen F_β (equation 5.3), specifically F_2. F_1 is a harmonic mean of precision and sensitivity. By using value of $\beta = 2$ we weigh sensitivity twice more than precision. This well expresses our intention to have sensitivity more important than accuracy.

$$PPV = \frac{TP}{TP + FP} \tag{5.1}$$

$$SEN = \frac{TP}{TP + FN} \tag{5.2}$$

$$F_\beta = (1 + \beta^2) \cdot \frac{PPV \cdot SEN}{\beta^2 \cdot PPV + SEN} \tag{5.3}$$

Table 5.1 shows short names of generalizations that are used in evaluation tables. Combinations of generalizations are represented by a plus sign in the order the generalizations were applied.

Table 5.1: Legend of generalization names used in evaluation tables

Generalization	Short name
Tree coverage	T
Owner	O
Owner directory	OD
Nonexistent	N
Multiple runs	M

5.4.2 Individual mining

This subsection presents result from evaluating single services.

PostgreSQL

Results for PostgreSQL mining are available in table 5.2. Generalization with lowest number of underpermission accesses (5) was combination OD+T. This is also the combination of generalizations with the best sensitivity. However, as it had more overpermissions (597), the best generalization according to the F_2 metric was OD (145 overpermissions, 17 underpermissions). This generalization was so effective because PostgreSQL contains a large number of files under `/var/lib/pgsql` that represent the database. The audit log covered only some of them and OD generalization was able to cover all except for a small anomaly in subfolders of `/usr/share/pgsql/timezonesets`, which is owned by root and not postgresql. This was improved by combining

OD and T generalizations (5 underpermissions), but at the cost of increased overpermission (597).

Table 5.2: Results of policy mining for PostgreSQL

Generalization	SEN	PPV	F_2
no gen.	0.7673	0.9778	0.8018
T	0.7770	0.9157	0.8013
O/M+O	0.8775	0.9806	0.8963
OD/OD+O/M+OD	0.9980	**0.9829**	**0.9949**
N/M+N	0.7704	0.9767	0.8044
M	0.7677	0.9779	0.8021
M+T	0.7774	0.9157	0.8016
N+T	0.7802	0.9160	0.8041
O+T	0.8856	0.9252	0.8933
OD+T	**0.9994**	0.9332	0.9854
O+N	0.8775	0.9795	0.8961
OD+N	0.9980	0.9820	0.9947

OpenSSH SSH Daemon

Results for OpenSSH SSH daemon mining are presented in table 5.3. In this case, only T and O generalizations had any effect on the generated policy. Other generalization algorithms didn't provide any improvement over policy with no generalization.

Tree coverage generalization had 11 underpermission accesses, mostly files related to the `/proc` filesystem. Owner generalization fixed 9 underpermission accesses compared to no generalization, with the total number of underpermissions of 362. However, most of these accesses were in `/usr/sbin` directory and it is assumed that after manual review these underpermissions can be ignored.

Table 5.3: Results of policy mining for OpenSSH

Generalization	SEN	PPV	F_2
no gen.	0.9209	**0.9769**	0.9316
T	**0.9977**	0.8902	**0.9741**
O	0.9228	0.9586	0.9297

Postfix

Results for the Postfix mail transfer agent policy mining are presented in table 5.3. The lowest number of overpermissions (662) was in the policy without generalization. Every other generalization algorithm increased the number of overpermission accesses, as expected. Non-existent generalization didn't provide any effect when used on it's own and also in most of the pairs. There is one interesting exception with combination of M+N, that achieved 882 underpermission accesses with F_2 metric considering this to be the best method for this service. However, it must be mentioned that this pair had the worst result in overpermission with 1655 accesses.

The most interesting thing about M+N method is that M nor N on its own couldn't provide such good results and this means that interactions between these two generalization algorithms produced this result. The underpermissions were mostly located in `/usr/libexec`.

Table 5.4: Results of policy mining for Postfix

Generalization	SEN	PPV	F_2
no gen./N	0.8484	**0.9509**	0.8671
T/N+T/OD+T	0.8886	0.9268	0.8960
O/O+N	0.8541	0.9487	0.8715
OD/OD+N	0.8484	0.9500	0.8669
M/M+OD	0.8510	0.9487	0.8689
M+T	0.8912	0.9256	0.8979
O+T	0.8943	0.9249	0.9003
OD+O	0.8541	0.9478	0.8714
M+O	0.8549	0.9464	0.8718
M+N	**0.9456**	0.8961	**0.9352**

Apache HTTP Server

For the Apache HTTP Server only algorithm capable of generalizing policy was the T algorithm with 336 overpermissions and 258 underpermission accesses. The T algorithm achieved F_2 metric of 0.95. Without generalization, the overpermission was just 85 accesses with F_2 score of 0.81.

Apache HTTP Server does not use a lot of owned files or temporary files, so the other generalization algorithms could not manifest themselves in the resulting policy.

Table 5.5: Results of policy mining for Apache HTTP Server

Generalization	SEN	PPV	F_2
no gen. and others	0.8090	**0.9800**	0.8382
T	**0.9500**	0.9358	**0.9471**

Individual mining summary

We can see that the policy mining results for individual services depended on the evaluated service. The best algorithm for generalization came out based on which files the service accessed and how the files are distributed in the file hierarchy, whether they have metadata, such as owners.

See table 5.6 for the summary of the expriments with the best algorithm for each metric. The best methods according to F_2 metric were tree coverage and owner directory. In one service, combination of multiple runs and non-existent files proved to be the best. On the contrary, other generalization methods did not show better results.

Table 5.6: Summary of policy mining for individual services

Service	Best SEN	Best PPV	Best F_2
PostgreSQL	OD+T	OD	OD
OpenSSH	T	no gen.	T
Postfix	M+N	no gen.	M+N
Apache HTTP Server	T	no gen.	T

5.4.3 Cumulative mining

This subsection contains evaluation of cumulative mining, meaning evaluating how the policy changes as more services are added to the mining algorithm. This evaluation is specifically intended for the tree algorithm, as its generalization is based on the coverage of the tree. The more services are used in the algorithm, more files and directories from the real filesystem are available for the algorithm to work with. Our hypothesis is that more services we use for the mining, the precision should go up. We will test this with services from the previous evaluation: PostgreSQL, OpenSSH, Postfix and Apache HTTP Server.

Results of the cumulative mining are presented in table 5.7.[5] This small

[5] Abbrevations used in the table — s: OpenSSH, postg: PostgreSQL, postf: Postfix, a: Apache HTTP Server.

example meets our hypothesis. By adding one or two service logs to the tree coverage algorithm, the resulting precision increases. However, this doesn't mean that any other combination will also show a similar pattern. We can prove this by adding the Apache service log, when precision drops to 0.972.

Table 5.7: Results of cumulative mining

Services	PPV
s	0.891
postg	0.916
postf	0.936
postg + s	0.965
postg + postf	0.968
s + postf	0.941
postg + s + postf	0.974
postg + s + postf + a	0.972

Conclusion

The goal of this dissertation was to design and implement algorithms that automatically generate a security policy for the Medusa security module. We managed to fulfill this goal and the result is a finished product in the form of an application that the user can use to configure the Constable authorization server. The resulting policy is created from an audit log of a running aplication, such as a system service. The solution is capable of creating security policy for multiple applications at once.

We compared the resulting implementation with the standard reference policy of the SELinux security module on the Fedora 37 distribution. The implementation was compared on four common services in the Linux operating system: PostgreSQL, Open SSH server, Postfix and Apache. Sensitivity results for the *best* algorithms ranged from 95.6% to 99.9%. These results show the good ability of our algorithm to cover the program accesses that should be allowed according to the priciple of least privilege. However, for the correct functionality of the program, the sensitivity must be 100%, and thus even after using our algorithm, manual intervention and correction of the security policy will be necessary. This correction should be simplified by the fact that most of the rules will be created automatically. The precision of our algorithm in experiments ranged from 93.4% to 98.3%.[6] The decrease in precision was caused by the generalization algorithms, which add paths to the policy that did not occur in the audit logs from which the policy was generated.

We must acknowledge that our research was limited in some respects. The evaluation of our algorithms was dependent on the manual setting of the generalization algorithm for FHS. Our results cannot be considered complete either, since we tested our application on only four system services.

With our work, we only scratched the surface of the problem. Future research may focus on new algorithms that better analyze the application requirements and improve the generated security policy so it complies with the principle of least privilege. Another direction in which the work can go

[6]As in multiple cases the best precision was achieved by using no generalization at all, we are listing the second best precision of a generalization algorithm.

is the static analysis of applications, which we did not deal with. Another benefit can be analyzing a larger number of services and finding mutual connections between them, for example by using machine learning, ontologies, or inductive logic programming.

The main contribution of this work is a finished application that can be used to configure the Medusa security module. Its modular design makes it possible to add additional generalization algorithms, where there are still open calls for research and improvement of the properties of the resulting generated security policy. After certain changes, it could be also used to generate policies for other security modules.

Kapitola 6

Rezumé

V tejto kapitole priblížime najdôležitejšie časti vybraných kapitol z práce v slovenskom jazyku. Netajíme sa ale tým, že pre úplne pochopenie problematiky preberanej v práci je potrebné prečítať originálnu verziu v anglickom jazyku.

6.1 Riadenie prístupu

Medzi základné modely riadenia prístupu patrí prístupová matica. Podľa toho, ako túto maticu rozdelíme, môžeme definovať ďalšie dva modely riadenia prístupu: zoznam riadenia prístupu (*Access control list*, ACL) a schopnosti (*capability*).

6.1.1 Matica riadenia prístupu

Maticu riadenia prístupu opísal Lampson. Pozostáva z riadkov reprezentujúcich subjekty[1] v systéme a stĺpcov reprezentujúcich objekty[2] v systéme.

Stav systému je definovaný trojicou (S, O, M), kde S je množina subjektov, O je množina objektov a M je matica riadenia prístupu. Jeden prvok matice je označený ako $M[s_i, o_j]$, kde s_i je subjekt v i-tom riadku a o_j objekt v j-tom stĺpci. Prvok matice obsahuje množinu oprávnení, ktoré s_i môže vykonať na o_j. Keďže na reálnych systémoch býva veľkosť tejto matice veľká a väčšina prvkov matice je prázdna, v tejto forme sa nepoužíva. Namiesto reprezentácie oprávnení pomocou matice sa používa jej rozklad na riadky alebo stĺpce. V prvom prípade sa povolenia ukladajú spolu so subjektmi, čím sa vytvárajú zoznamy schopností, ktoré má daný subjekt pre každý objekt v zozname (pozri 6.1.3). V druhom prípade sú oprávnenia uložené s každým objektom v systéme, čím sa vytvárajú zoznamy riadenia prístupu (pozri 6.1.2).

[1] Systémové entity, ktoré vykonávajú operácie na objektoch, napr. procesy.

[2] Systémové entity, na ktorých sú vykonávané operácie, napr. súbory.

6.1.2 Zoznam riadenia prístupu

Rozložením matice riadenia prístupu podľa stĺpcov získame zoznamy riadenia prístupu (ACL). Zoznam riadenia prístupu pre objekt o je reprezentovaný ako zoznam dvojíc $(s_i, \{a_1, \ldots, a_n\})$, kde s_i je subjekt a $\{a_1, \ldots, a_n\}$ je množina oprávnení, ktoré s_i môže vykonať na o. Keďže zoznam riadenia prístupu je uložený spolu s objektom, správcovi systému to uľahčuje zobrazenie úplného zoznamu subjektov, ktoré môžu narábať s konkrétnym objektom. Opak je zložitejší – aby sme mohli vidieť všetky operácie, ktoré môže konkrétny subjekt vykonávať, museli by sme hľadať tento subjekt v zoznamoch prístupových práv všetkých objektov v operačnom systéme.

6.1.3 Schopnosti

Rozložením matice riadenia prístupu po riadkoch získame zoznamy schopností. Zoznam schopností pre subjekt s je reprezentovaný ako zoznam dvojíc $(o_i, \{a_1, \ldots, a_n\})$, kde o_i je objekt a $\{a_1, \ldots, a_n\}$ je množina oprávnení, ktoré môže s vykonať na o_i.

Teraz sú výhody a nevýhody zoznamov schopností vymenené v porovnaní s prístupovými zoznamami. Zistiť všetky povolenia pre konkrétny subjekt je triviálne, pretože sú uložené spolu so subjektom. Zistiť zoznam subjektov, ktoré môžu vykonávať nejaké operácie na konkrétnom objekte nie je možné bez iterácie cez všetky subjekty v systéme.[3]

Schopnosti si môžeme predstaviť ako poverenia – ich vlastníctvo umožňuje vlastníkovi pristupovať k objektom uvedeným v zozname schopností a vykonávať povolené operácie. To ponúka možnosti, ktoré nie sú dostupné pri ACL, napríklad delegovanie. Na druhú stranu, systémy, ktoré podporujú delegovanie schopností musia vyriešiť problém odvolania schopnosti.

6.2 Bezpečnosť v systéme Linux

V tejto sekcii predstavíme komponenty systému Linux, ktoré sú zodpovedné za riadenie prístupu k systémovým objektom. Jedná sa o základný model UGO, ktorý Linux prevzal z UNIX-u, ďalej *zoznamy riadenia prístupu*, ktoré sú rozšírením UGO modelu, *schopnosti* a na záver *povinné riadenie prístupu*, ktoré je v Linuxe implementované cez rozhranie *Linux security modules* (LSM).

6.2.1 Model UGO

Hlavným účelom architektúry UGO je rozdeliť a zabezpečiť prístupy k súborom pre každého používateľa v systéme a poskytnúť metódy na zdieľanie

[3]Skutočný čas vykonania tejto operácie v porovnaní s vyhľadávaním zoznamov ACL pre konkrétny subjekt môže byť kratší, pretože vo väčšine systémov je podstatne viac objektov ako subjektov.

týchto súborov kontrolovaným spôsobom. V tomto prípade oprávnenia určujú vlastníci týchto súborov. Takýto prístup sa označuje ako *voliteľné riadenie prístupu* (DAC).

Každému súboru je priradený vlastník a skupina. Tieto sú identifikované číselnými identifikátormi UID a GID. Keď sa vytvára nový súbor, tieto identifikátory sa zvyčajne nastavia pomocou efektívneho UID a GID vlákna, ktoré ho vytvára.

Povolenia v modeli UGO pre jeden súbor môžu byť priradené trom množinám používateľov:

vlastniaci používateľ Používateľ, ktorý vlastní súbor (jednoprvková množina).

skupina Používatelia patriaci do skupiny, ktorá vlastní súbor.

ostatní používatelia Všetci ostatní používatelia.

Každá z týchto množín môže obsahovať tri oprávnenia: *čítanie*, *zápis* a *vykonanie*, označené r, w a x. Aktuálny stav oprávnení v danej množine reprezentujú tri bity. Povolenia sa kontrolujú podľa prvej vhodnej množiny v poradí, ako sú zapísané vyššie.

6.2.2 Zoznamy riadenia prístupu

Zoznamy riadenia prístupu používajú rovnaké povolenia ako model UGO, ale umožňujú granulárnejšie riadenie prístupu ďalších používateľov a skupín. Pozostávajú zo záznamov ACL. Jeden záznam obsahuje typ záznamu, kvalifikátor špecifikujúci identitu používateľa alebo skupiny a oprávnenia týkajúce sa záznamu. Platný ACL musí obsahovať aspoň tri záznamy týchto typov: `ACL_USER_OBJ`, `ACL_GROUP_OBJ` a `ACL_OTHER`. Tieto priamo odrážajú povolenia vlastníka, skupiny a ostatných používateľov.

Oproti štandardnému modelu UGO, je možné špecifikovať oprávnenia aj pre iných používateľov a iné skupiny pomocou záznamov typov `ACL_USER` a `ACL_GROUP`.

6.2.3 Schopnosti v Linuxe

V operačných systémoch podobných UNIX-u (bez podpory schopností) existujú dve základné úrovne privilégií. Keď proces beží pod efektívnym UID rovným nule, je privilegovaný a môže vykonávať akúkoľvek úlohu v systéme. V opačnom prípade je neprivilegovaný a nemôže vykonávať privilegované akcie (napr. otvorenie nízkeho portu alebo reštartovanie systému).

Schopnosti v Linuxe sú množina oprávnení, ktoré vznikli rozdelením oprávnení superužívateľa na menšie jednotky. Toto umožňuje explicitne špecifikovať,

ktoré oprávnenia z celkovej množiny sú potrebné pre spustený proces, čím sa dodržiava princíp najmenších privilégií.

Hlavnou motiváciou pre zavedenie schopností boli privilegované aplikácie (napr. sieťové služby), ktoré nepotrebovali všetky privilégiá superužívateľa. Keďže takéto programy boli často dostupné na diaľku cez otvorené porty na internet, stali sa terčom útokov. Ak bol útočník schopný spustiť ľubovoľný kód prostredníctvom takéhoto programu, automaticky získal všetky privilégiá superužívateľa a mohol vykonávať ľubovoľné akcie v systéme.

6.2.4 Povinné riadenie prístupu

Koncom 90-tych rokov sa ukázalo, že mechanizmy riadenia prístupu v Linuxe nie sú dostatočné na to, aby poskytovali silné zabezpečenie. Prvé projekty, ktoré zlepšili mechanizmy riadenia prístupu v Linuxe, sa spoliehali na intervenciu systémových volaní alebo vyžadovali záplatu, ktorá vložila rozhodovacie funkcie na vhodne vybrané miesta. Medusa, SELinux, grSecurity a ďalšie bezpečnostné riešenia boli pôvodne implementované ako takéto záplaty.

6.2.5 Linux security modules

Prvý bezpečnostný modul, ktorý bol zahrnutý v jadre, bol SELinux. Linus Torvalds[4] požadoval, aby sa namiesto implementácie jedného bezpečnostného modulu upravilo jadro, aby si používateľ mohol vybrať, ktorý bezpečnostný modul má byť aktivovaný. Požiadal teda vývojárov jadra, aby vytvorili generický rámec, ktorý by umožnil načítať a použiť akýkoľvek modul, ktorý implementuje dohodnuté rozhranie. To viedlo k vytvoreniu rámca Linux security modules (LSM).

Tento rámec umožňuje bezpečnostnému modulu zasahovať do operácií v kritických častiach jadra, zvyčajne počas systémových volaní. Je to vďaka umiestneniu záchytných funkcií tesne pred prístupom jadra k nejakému zdroju. Bezpečnostné moduly môžu do týchto záchytných funkcií zavesiť svoje vlastné funkcie. Po zavolaní záchytnej funkcie je každá zavesená funkcia zavolaná – vďaka tomu môže každý bezpečnostný modul vyhodnotiť prístup a rozhodnúť, či aplikácia dodržiava bezpečnostnú politiku. Aby bola operácia povolená, musia ju povoliť všetky bezpečnostné moduly. Odmietnutie z jedného bezpečnostného modulu postačuje na odmietnutie celej operácie. Verzia OS Linux 6.2 poskytuje LSM 247 záchytných funkcií. Záchytné funkcie delíme na dva druhy:

bezpečnostné záchytné funkcie Tieto funkcie sa používajú na rozhodovanie o bezpečnostných udalostiach v systéme. Sú umiestnené na miestach kódu, kde sa subjekty chystajú pristupovať z používateľského priestoru

[4]Tvorca a súčasný správca projektu Linux.

k objektom jadra. Bezpečnostný modul musí rozhodnúť, či je prístup povolený.

ovládacie záchytné funkcie Tieto funkcie slúžia na notifikáciu bezpečnostných modulov o dôležitých udalostiach v systéme, ktoré sa ich môžu týkať. Typické príklady sú funkcie *alloc* a *free*. Tieto funkcie môže bezpečnostný modul použiť na alokáciu alebo uvoľnenie bezpečnostnej položky (pozri nižšie). Tieto funkcie sa zvyčajne volajú, keď sa vytvorí alebo uvoľní nový objekt jadra.

Ďalšou dôležitou súčasťou LSM sú bezpečnostné položky, nazývané aj *security blobs*. Sú to smerníky na dátové štruktúry, ktoré používa bezpečnostný modul na ukladanie svojich vlastných informácií o entitách v jadre.

Medzi nevýhody LSM patrí to, že LSM nie je autoritatívny, teda po zamietnutí operácie politikou iného bezpečnostného mechanizmu jadra už LSM nemôže zmeniť toto rozhodnutie.

6.3 Bezpečnostný modul Medusa

Medusa je bezpečnostný modul pre operačný systém Linux, ktorý bol vyvinutý na Fakulte elektrotechniky a informatiky na prelome tisícročí Marekom Zelemom a Milanom Pikulom [42, 43, 44]. Celý systém sa skladá z týchto častí:

1. Modul jadra. Architektúra systému Medusa umožňuje viacero takýchto modulov, ktoré nemusia byť priamou súčasťou operačného systému (napr. rozhodovací modul implementovaný v databázovom serveri). V súčasnosti existuje jedna implementácia pre operačný systém Linux integrovaná pomocou LSM.

2. Komunikačný protokol, ktorý sa používa na prenos informácií z viacerých modulov jadra na autorizačný server. Tento protokol je schopný serializovať akúkoľvek entitu jadra, ktorú je potrebné preskúmať z hľadiska riadenia prístupu.

3. Autorizačný server, ktorý analyzuje a ukladá politiku riadenia prístupu. Funkcie, ktoré priraďujú členstvo vo virtuálnych svetoch (pozri nižšie) sú definované autorizačným server na základe načítanej politiky. Od modulu jadra sa očakáva, že každú neoznačenú entitu odošle autorizačnému serveru na označkovanie (zaradenie do virtuálnych svetov, pozri nižšie). Jeden autorizačný server môže ovládať jeden alebo niekoľko modulov jadra.

Hlavným princípom bezpečnostného modelu Medusy je priradenie systémoch entít do množín nazývaných virtuálne svety (VS). Subjekty majú niekoľko množín VS podľa typu prístupu (čítanie, zápis). Keď je subjekt vo virtuálnom svete v na čítanie, potom môže čítať všetky objekty, ktoré sa nachádzajú v tom istom svete.

6.4 Ťažba bezpečnostnej politiky

Hlavným cieľom tejto práce je navrhnúť a implementovať algoritmus, ktorý vytvorí funkčnú bezpečnostnú politiku pre bezpečnostný modul Medusa s minimálnym administratívnym zásahom. Ide o všeobecný cieľ, ktorého dosiahnutie by mohlo byť príliš náročné. Aby sme ho lepšie konkretizovali, zavádzame tieto obmedzenia:

1. Bezpečnostná politika bude brať do úvahy len operácie a prístupy k súborovému systému.
2. Generovanie bezpečnostnej politiky sa zameria na obmedzenie množiny dostupných objektov pre aplikáciu. Tento koncept je podobný technike *sandboxu*. Obmedzením prístupu aplikácie obmedzíme *priestor vektorov útokov* na aplikáciu. To je obzvlášť užitočné pre sieťové služby.
3. Vytvorená politika je statická. To znamená, že ak chce správca neskôr politiku aktualizovať, musí znova spustiť algoritmus na ťažbu politík.

6.4.1 Výskumné otázky

Očakávané výstupy práce možno zhrnúť do týchto výskumných otázok:

1. Ak by sme skonštruovali algoritmus, ktorý by vytváral bezpečnostnú politiku len zo záznamov operácií, vyrovnala by sa táto politika politike vytvorenej správcom?
2. Ak by sme skonštruovali algoritmus, ktorý by vytvoril bezpečnostnú politiku zo záznamov operácií vrátane externých informácií z prostredia systému, vyrovnala by sa táto politika politike vytvorenej správcom?

6.4.2 Návrh riešenia

Naše navrhované riešenie sa inšpiruje modelom *domain-type enforcement* [65, 66], ktorý vytvára *domény* pre každú spustenú aplikáciu. Pozorovaním činnosti aplikácie (za predpokladu, že aplikácia nie je škodlivá) môžeme vytvoriť zoznam objektov, ku ktorým by mala mať aplikácia prístup.

Stručné zhrnutie navrhovaného riešenia je takéto:

1. Budeme sledovať činnosť aplikácie, pre ktorú bude vytvorená bezpečnostná politika.

2. Vytvoríme predbežnú politiku pre každý subjekt (reprezentovaný doménou) na základe názvu objektu a požadovanej operácie.

3. V predbežnej politike budeme skúmať a dopĺňať chýbajúce pravidlá, ktoré spôsobujú zamietnutia legitímnych oeprácií. Tento krok nazývame generalizácia. Po tomto kroku by mala byť k dispozícii úplná použiteľná politika pre aplikáciu.

6.4.3 Získavanie záznamov

Vstupné údaje pre algoritmus by mali byť úplné, čo znamená, že každý prístup do systému by mal byť bez výnimiek zaznamenaný. Ideálnym spôsobom získavania záznamov v operačnom systéme Linux je subsystém *audit*. Dokáže zaznamenávať systémové volania a rôzne udalosti súvisiace s bezpečnosťou v operačnom systéme na základe nastavení poskytnutých službou `auditd`.

S využitím príspevkov z [73] sme upravili bezpečnostný modul Medusa tak, aby auditoval každú zachytenú operáciu vybraného procesu. Proces, ktorý sa má auditovať, možno vybrať pomocou obslužnej funkcie `fexec` v konfigurácii autorizačného servera. Operácie, ktoré sú auditované, sú uvedené v prílohe A.

6.4.4 Generalizácia

Z povahy záznamov z auditu sme identifikovali tieto problémy, ktoré spôsobujú *chýbajúce povolenia*:[5]

1. Na základe vykonávania aplikácie sa nemuseli vykonať všetky cesty vykonávania, a preto sa niektoré prístupy nemuseli prejaviť v záznamoch. Tieto prístupy budú po uplatnení politiky zamietnuté.

2. Prístupy k dočasným alebo novo vytvoreným súborom sa budú týkať ciest, ktoré neboli zachytené v pôvodných protokoloch auditu. Zdôrazňujeme, že v porovnaní s predchádzajúcim bodom bol prístup požadovaný, ale cesta je pri ďalšom vykonaní iná. Dôsledok je rovnaký – po vynútení politiky budú tieto prístupy zamietnuté.

Riešením tohto problému je zovšeobecnenie – modul na generovanie politík musí uvoľniť vygenerované pravidlá tak, aby zväčšili množinu pokrytia

[5] Prístupy, ktoré v politike chýbajú, aby mohla aplikácia normálne fungovať.

možných ciest. Generalizácia zároveň vytvorí nadbytočné povolenia,[6] ktoré sú nežiaduce. Generovanie politiky musí vyriešiť optimalizačný problém – udržiavať nadbytočné povolenia na nízkej úrovni a zároveň mať čo najmenej chýbajúcich povolení, ideálne žiadne.

Generalizáciou sa vytvárajú pravidlá, ktoré sa vzťahujú na viacero ciest, ktoré sa v súborovom systéme nemusia vyskytovať. To sa dá dosiahnuť pomocou regulárnych výrazov a/alebo rekurzívnych pravidiel. Napríklad pravidlo ("`/var/log/pgsql/`$.*$", do, P) umožňuje procesom v doméne do vykonávať operácie, ktoré vyžadujú oprávnenia P na ľubovoľnom súbore v adresári `/var/log/pgsql`.

Navrhli sme tieto generalizačné algoritmy:

- generalizácia podľa pokrytia súborovej hierarchie,
- generalizácia podľa štandardu *Filesystem Hierarchy* (FHS),
- generalizácia na základe neexistujúcich súborov,
- generalizácia podľa oprávnení UGO,
- generalizácia podľa vlastníka priečinka,
- generalizácia na základe viacnásobného spustenia.

6.4.5 Zhrnutie výsledkov

Výslednú implementáciu sme porovnali so štandardnou referenčnou politikou bezpečnostného modulu SELinux v distribúcii Fedora 37. Implementáciu sme porovnávali na štyroch bežných službách: PostgreSQL, Open SSH server, Postfix a Apache HTTP server. Výsledky citlivosti pre najlepšie algoritmy sa pohybovali od 95,6 % do 99,9 %. Tieto výsledky preukazujú dobrú schopnosť nášho algoritmu pokryť prístupy programov, ktoré by mali byť povolené podľa *princípu najmenších privilégií.* Pre správnu funkčnosť programu však musí byť citlivosť 100 %, a preto aj po použití nášho algoritmu bude potrebný manuálny zásah a korekcia bezpečnostnej politiky administrátorom. Táto korekcia ale bude zjednodušená tým, že veľká časť pravidiel sa bude vytvárať automaticky. Presnosť nášho algoritmu sa pri experimentoch pohybovala od 93,4 % do 98,3 %.[7] Zníženie presnosti spôsobili generalizačné algoritmy, ktoré

[6] Pravidlá v politike, ktoré povoľujú operácie, ktoré aplikácia nepotrebuje ku svojej bežnej činnosti.

[7] Keďže v niekoľkých testoch mal najlepšiu presnosť prípad bez generalizácie, uvádzame druhú najlepšiu presnosť generalizačného algoritmu.

do politiky pridávajú cesty, ktoré sa nevyskytovali v záznamoch operácií, z ktorých bola politika vytvorená.

Je potrebné uznať, že náš výskum bol v niektorých ohľadoch obmedzený. Vyhodnotenie našich algoritmov záviselo od manuálneho nastavenia generalizačného algoritmu pre *File Hierarchy Standard*. Naše výsledky tiež nemožno považovať za úplné, pretože sme našu aplikáciu testovali len na štyroch systémových službách.

Budúci výskum sa môže zamerať na nové algoritmy, ktoré lepšie analyzujú požiadavky aplikácie a vylepšujú generovanú bezpečnostnú politiku tak, aby bola v súlade s princípom najmenších privilégií. Ďalším smerom, ktorým sa práca môže uberať, je statická analýza aplikácií, ktorou sme sa nezaoberali. Iným prínosom môže byť analýza väčšieho počtu služieb a hľadanie vzájomných súvislostí medzi nimi, napríklad pomocou strojového učenia, ontológií alebo induktívneho logického programovania.

Hlavným prínosom tejto práce je hotová aplikácia, ktorú možno použiť na konfiguráciu bezpečnostného modulu Medusa. Jej modulárna konštrukcia umožňuje pridávať ďalšie generalizačné algoritmy, čo otvára priestor skúmaniu ďalších metód na zlepšenie vlastností výslednej generovanej bezpečnostnej politiky. Po určitých úpravách by sa dala použiť aj na generovanie politík pre iné bezpečnostné moduly.

Bibliography

1. 5200.28-STD, DoD. *Trusted Computer System Evaluation Criteria*. Dod Computer Security Center, 1985.
2. TILBORG, Henk C. A. van and JAJODIA, Sushil (eds.). *Encyclopedia of Cryptography and Security, 2nd Ed.* Springer, 2011. ISBN 978-1-4419-5905-8. Available from DOI: `10.1007/978-1-4419-5906-5`.
3. LAMPSON, Butler W. Protection. *ACM SIGOPS Operating Systems Review*. 1974, vol. 8, no. 1, pp. 18–24. Available from DOI: `10.1145/775265.775268`.
4. BELL, E. D. and LA PADULA, J. L. *Secure computer system: Unified exposition and Multics interpretation*. Bedford, MA: Mitre Corporation, 1976. Available also from: `http://csrc.nist.gov/publications/history/bell76.pdf`.
5. RAY, Indrakshi and KUMAR, Mahendra. Towards a location-based mandatory access control model. *Computers & Security*. 2006, vol. 25, no. 1, pp. 36–44. ISSN 0167-4048. Available from DOI: `https://doi.org/10.1016/j.cose.2005.06.007`.
6. RAY, Indrakshi and KUMAR, Mahendra. Towards a location-based mandatory access control model. *Computers & Security*. 2006, vol. 25, no. 1, pp. 36–44.
7. BIBA, J. K. *Integrity Considerations for Secure Computer Systems*. Bedford, MA: U.S. Air Force Electronic Systems Division, 1977-04. Tech. rep. The MITRE Corporation.
8. FERRAIOLO, D. F. and KUHN, D. R. Role-based access control. *15th National Computer Security Conference*. 1992.
9. SANDHU, Ravi S, COYNE, Edward J, FEINSTEIN, Hal L and YOUMAN, Charles E. Role-based access control models. *Computer*. 1996, vol. 29, no. 2, pp. 38–47.

10. *Overview of role-based access control in Azure Active Directory* [online]. 2023-04-10. [visited on 2023-05-14]. Available from: `https://learn.microsoft.com/en-us/azure/active-directory/roles/custom-overview`.
11. THE KUBERNETES AUTHORS. *Using RBAC Authorization* [online]. 2022-01-08. [visited on 2023-05-14]. Available from: `https://kubernetes.io/docs/reference/access-authn-authz/rbac/`.
12. HU, Vincent C, FERRAIOLO, David F, CHANDRAMOULI, Ramaswamy and KUHN, D Richard. *Attribute-Based Access Control*. Artech House, 2017.
13. BISWAS, Prosunjit, SANDHU, Ravi and KRISHNAN, Ram. Label-Based Access Control: An ABAC Model with Enumerated Authorization Policy. In: *Proceedings of the 2016 ACM International Workshop on Attribute Based Access Control*. New Orleans, Louisiana, USA: Association for Computing Machinery, 2016, pp. 1–12. ABAC '16. ISBN 9781450340793. Available from DOI: `10.1145/2875491.2875498`.
14. GATES, Carrie. Access Control Requirements for Web 2.0 Security and Privacy. 2007.
15. PANG, Ruoming, CACERES, Ramon, BURROWS, Mike, CHEN, Zhifeng, DAVE, Pratik, GERMER, Nathan, GOLYNSKI, Alexander, GRANEY, Kevin, KANG, Nina, KISSNER, Lea, KORN, Jeffrey L., PARMAR, Abhishek, RICHARDS, Christina D. and WANG, Mengzhi. Zanzibar: Google's Consistent, Global Authorization System. In: *2019 USENIX Annual Technical Conference (USENIX ATC '19)*. Renton, WA, 2019.
16. OORSCHOT, Paul C. van. *Computer Security and the Internet - Tools and Jewels from Malware to Bitcoin, Second Edition*. Springer, 2021. Information Security and Cryptography. ISBN 978-3-030-83410-4. Available from DOI: `10.1007/978-3-030-83411-1`.
17. SPEIGHT, Toby. *Answer to question: What is the reason for having or restricting file owner's permissions?* [online]. 2022. [visited on 2023-02-12]. Available from: `https://unix.stackexchange.com/a/699436`.
18. IEEE. *Draft Standard for Information Technology—Portable Operating System Interface (POSIX)—Part 1: System Application Program Interface (API)— Amendment #: Protection, Audit and Control Interfaces [C Language]*. New York, NY, USA: IEEE, 1997.
19. GRUENBACHER, Andreas. *acl(5) — BSD File Formats Manual* [online]. 2002. [visited on 2023-01-25]. Available from: `https://man7.org/linux/man-pages/man5/acl.5.html`.

20. KERRISK, Michael. *capabilities(7) — Linux manual page* [online]. 2021. [visited on 2023-01-11]. Available from: https://man7.org/linux/man-pages/man7/capabilities.7.html.
21. HALLYN, Serge E and MORGAN, Andrew G. Linux capabilities: making them work. In: *Proceedings of the Linux Symposium*. 2008.
22. CORBET, Jon (ed.). *Kernel development* [online]. 1999. [visited on 2023-02-06]. Available from: https://lwn.net/1999/1202/kernel.php3.
23. LUTOMIRSKI, Andy. *capabilities: Ambient capabilities* [online]. 2015. [visited on 2023-02-10]. Available from: https://lwn.net/Articles/636533/.
24. WRIGHT, C., COWAN, C., MORRIS, J., SMALLEY, S. and KROAH-HARTMAN, G. Linux Security Modules: General Security Support for the Linux Kernel. *Security 2002*. 2002. Available also from: https://www.kernel.org/doc/ols/2008/ols2008v1-pages-163-172.pdf.
25. HAINES, Richard. *The SELinux Notebook*. 2014.
26. VERMEULEN, Sven. *Selinux System Administration*. Packt Publishing, 2016. ISBN 978-1-78712-695-4.
27. VERMEULEN, Sven and EVANS, Brian. *SELinux/Users and logins* [online]. Gentoo Foundation, Inc., 2015-01-13. [visited on 2020-01-27]. Available from: https://wiki.gentoo.org/wiki/SELinux/Type_enforcement#Permissions.
28. *SELinux Reference Policy* [online]. [visited on 2018-12-10]. Available from: https://github.com/SELinuxProject/refpolicy.
29. *TOMOYO Linux 2.6.x : The Official Guide* [online]. [visited on 2020-01-27]. Available from: http://tomoyo.osdn.jp/2.6/index.html.en.
30. *Policy specification* [online]. .62nd ed. NTT DATA Corporation, 2019-02-06. [visited on 2020-01-22]. Available from: https://tomoyo.osdn.jp/2.6/policy-specification/index.html.en.
31. CONTRIBUTORS & OTHERS, openSUSE. *SDB:AppArmor* [online]. SUSE LLC, 2017-08-10. [visited on 2020-01-22]. Available from: https://en.opensuse.org/SDB:AppArmor.
32. *AppArmor Profiles* [online]. [visited on 2019-01-15]. Available from: https://wiki.ubuntu.com/SecurityTeam/KnowledgeBase/AppArmorProfiles.
33. LARABEL, Michael. *Solus 3 Linux Distribution Released For Enthusiasts* [online]. [visited on 2019-01-15]. Available from: https://www.phoronix.com/scan.php?page=news_item&px=Solus-3-Released.

34. *AppArmor/HowToUse* [online]. 2019-03-13. [visited on 2020-01-22]. Available from: `https://wiki.debian.org/AppArmor/HowToUse#Enable_AppArmor`.

35. ACIICMEZ, O. and BLAICH, A. *Understanding the Permission and Access Control Model for Tizen Application Sandboxing* [online]. Samsung, 2012-05-09. [visited on 2020-01-22]. Available from: `http://download.tizen.org/misc/media/conference2012/wednesday/seacliff/2012-05-09-0945-1025-understanding_the_permission_and_access_control_model_for_tizen_application_sandboxing.pdf`.

36. *Integrity subsystem* [online]. Linux Kernel Organization, 2019 [visited on 2020-01-31]. Available from: `https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/security/integrity/Kconfig?h=v5.5`.

37. *Extended Verification Module* [online]. Linux Kernel Organization, 2019 [visited on 2020-01-31]. Available from: `https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/security/integrity/evm/Kconfig?h=v5.5`.

38. *Integrity Measurement Architecture* [online]. Linux Kernel Organization, 2019 [visited on 2020-01-31]. Available from: `https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/security/integrity/ima/Kconfig?h=v5.5`.

39. *Yama* [online]. [visited on 2020-01-27]. Available from: `https://www.kernel.org/doc/html/v5.5/admin-guide/LSM/Yama.html`.

40. [online]. Canonical Ltd. [visited on 2019-01-15]. Available from: `http://manpages.ubuntu.com/manpages/cosmic/en/man8/aa-autodep.8.html`.

41. HERTZOG, Raphaël and MAS, Roland. *The Debian Administrator's Handbook*. Freexian SARL, 2015. ISBN 9791091414043.

42. ZELEM, Marek and PIKULA, Milan. *ZP Security Framework* [online]. Faculty of Electrical Engineering and Information Technology, Slovak University of Technology in Bratislava, 2000 [visited on 2023-02-26]. Available from: `http://medusa.terminus.sk/English/medusa-paper.ps`.

43. PIKULA, Milan. *Distribuovaný systém na zvýšenie bezpečnosti heterogénnej počítačovej siete*. Bratislava: Bratislava: FEI STU, 2002.

44. ZELEM, Marek. *Integrácia rôznych bezpečnostných politík do OS Linux*. Bratislava: Bratislava: FEI STU, 2001.

45. MOLLOY, Ian, PARK, Youngja and CHARI, Suresh. Generative Models for Access Control Policies: Applications to Role Mining over Logs with Attribution. In: *Proceedings of the 17th ACM Symposium on Access Control Models and Technologies*. Newark, New Jersey, USA: Association for Computing Machinery, 2012, pp. 45–56. SACMAT '12. ISBN 9781450312950. Available from DOI: 10.1145/2295136.2295145.

46. XU, Zhongyuan and STOLLER, Scott D. Mining Attribute-Based Access Control Policies from Logs. In: *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2014, pp. 276–291. Lecture Notes in Computer Science. Available from DOI: 10.1007/978-3-662-43936-4_18.

47. IYER, Padmavathi and MASOUMZADEH, Amirreza. Mining Positive and Negative Attribute-Based Access Control Policy Rules. In: *Proceedings of the 23nd ACM on Symposium on Access Control Models and Technologies*. 2018, nil. Available from DOI: 10.1145/3205977.3205988.

48. COTRINI, Carlos, WEGHORN, Thilo and BASIN, David. Mining ABAC Rules from Sparse Logs. In: *2018 IEEE European Symposium on Security and Privacy (EuroS&P)*. 2018, nil. Available from DOI: 10.1109/eurosp.2018.00011.

49. KARIMI, Leila and JOSHI, James. An Unsupervised Learning Based Approach for Mining Attribute Based Access Control Policies. In: *2018 IEEE International Conference on Big Data (Big Data)*. 2018, nil. Available from DOI: 10.1109/bigdata.2018.8622037.

50. KARIMI, Leila, ALDAIRI, Maryam, JOSHI, James and ABDELHAKIM, Mai. An Automatic Attribute Based Access Control Policy Extraction From Access Logs. *IEEE Transactions on Dependable and Secure Computing*. 2021, pp. 1–1. Available from DOI: 10.1109/tdsc.2021.3054331.

51. BUI, Thang, STOLLER, Scott D. and LI, Jiajie. Mining Relationship-Based Access Control Policies from Incomplete and Noisy Data. In: *Foundations and Practice of Security*. Springer International Publishing, 2019, pp. 267–284. Foundations and Practice of Security. Available from DOI: 10.1007/978-3-030-18419-3_18.

52. COTRINI, Carlos, CORINZIA, Luca, WEGHORN, Thilo and BASIN, David. *The Next 700 Policy Miners: A Universal Method for Building Policy Miners*. 2019. Available from arXiv: 1908.05994 [cs.CR].

53. SNIFFEN, Brian T, HARRIS, David R and RAMSDELL, John D. Guided policy generation for application authors. In: *SELinux Symposium*. 2006.

54. LACHMUND, Sven. Auto-generating access control policies for applications by static analysis with user input recognition. In: *Proceedings of the 2010 ICSE Workshop on Software Engineering for Secure Systems - SESS '10*. 2010. Available from DOI: 10.1145/1809100.1809102.

55. RAUTER, Tobias, HOLLER, Andrea, KAJTAZOVIC, Nermin and KREINER, Christian. Towards an automated generation of application confinement policies with binary analysis. In: *2015 International Symposium on Networks, Computers and Communications (ISNCC)*. 2015, nil. Available from DOI: 10.1109/isncc.2015.7238568.

56. WANG, Ruowen, ENCK, William, REEVES, Douglas, ZHANG, Xinwen, NING, Peng, XU, Dingbang, ZHOU, Wu and AZAB, Ahmed M. EASEAndroid: Automatic Policy Analysis and Refinement for Security Enhanced Android via Large-Scale Semi-Supervised Learning. In: *24th USENIX Security Symposium (USENIX Security 15)*. Washington, D.C.: USENIX Association, 2015, pp. 351–366. ISBN 978-1-939133-11-3. Available also from: https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/wang-ruowen.

57. MOCANU, Decebal Constantin, TURKMEN, Fatih and LIOTTA, Antonio. Towards ABAC Policy Mining from Logs with Deep Learning. In: FOMICHOV, V.A. and FOMICHOVA, O.S. (eds.). *Proceedings of the 18th International Multiconference - Intelligent Systems, IS 2015*. Jožef Stefan Institute, 2015. 18th International Multiconference - Intelligent Systems, IS 2015, IS ; Conference date: 12-10-2015 Through 13-10-2015.

58. SANDERS, Matthew W. *Automated methods for generating least privilege access control policies*. Colorado School of Mines. Arthur Lakes Library, 2019. Available also from: https://hdl.handle.net/11124/173028. PhD thesis.

59. LAW, Mark, RUSSO, Alessandra, BERTINO, Elisa, BRODA, Krysia and LOBO, Jorge. Fastlas: Scalable Inductive Logic Programming Incorporating Domain-Specific Optimisation Criteria. *Proceedings of the AAAI Conference on Artificial Intelligence*. 2020, vol. 34, no. 03, pp. 2877–2885. Available from DOI: 10.1609/aaai.v34i03.5678.

60. JABAL, Amani Abu, BERTINO, Elisa, LOBO, Jorge, LAW, Mark, RUSSO, Alessandra, CALO, Seraphin and VERMA, Dinesh. Polisma - A Framework for Learning Attribute-Based Access Control Policies. In: *Computer Security - ESORICS 2020*. Springer International Publishing, 2020, pp. 523–544. Computer Security - ESORICS 2020. Available from DOI: 10.1007/978-3-030-58951-6_26.

61. LI, Yun, HUANG, Chenlin, YUAN, Lu, DING, Yan and CHENG, Hua. ASPGen: an Automatic Security Policy Generating Framework for AppArmor. In: *2020 IEEE Intl Conf on Parallel & Distributed Processing with Applications, Big Data & Cloud Computing, Sustainable Computing & Communications, Social Computing & Networking (ISPA/BDCloud/-SocialCom/SustainCom)*. 2020, pp. 392–400. Available from DOI: `10.1109/ISPA-BDCloud-SocialCom-SustainCom51426.2020.00075`.

62. GOLDBERG, Ian, WAGNER, David, THOMAS, Randi and BREWER, Eric A. A Secure Environment for Untrusted Helper Applications. In: *6th USENIX Security Symposium (USENIX Security 96)*. San Jose, CA: USENIX Association, 1996. Available also from: `https://www.usenix.org/conference/6th-usenix-security-symposium/secure-environment-untrusted-helper-applications`.

63. PANCHOLI, Meghna, KELLAS, Andreas D., KEMERLIS, Vasileios P. and SETHUMADHAVAN, Simha. Timeloops: Automatic System Call Policy Learning for Containerized Microservices. *CoRR*. 2022. Available from arXiv: `2204.06131v3 [cs.CR]`.

64. GOLDBERG, Ian, WAGNER, David, THOMAS, Randi and BREWER, Eric A. A Secure Environment for Untrusted Helper Applications Confining the Wily Hacker. In: *Proceedings of the 6th Conference on USENIX Security Symposium, Focusing on Applications of Cryptography - Volume 6*. San Jose, California: USENIX Association, 1996, p. 1. SSYM'96.

65. WALKER, Kenneth M., STERNE, Daniel F., BADGER, M. Lee, PETKAC, Michael J., SHERMAN, David L. and OOSTENDORP, Karen A. Confining Root Programs with Domain and Type Enforcement. In: *6th USENIX Security Symposium (USENIX Security 96)*. San Jose, CA: USENIX Association, 1996. Available also from: `https://www.usenix.org/conference/6th-usenix-security-symposium/confining-root-programs-domain-and-type-enforcement`.

66. HALLYN, Serge E. *Domain and Type Enforcement for Linux*. 2003. PhD thesis. The College of William & Mary in Virginia.

67. ACHARYA, Anurag and RAJE, Mandar. MAPbox: Using Parameterized Behavior Classes to Confine Untrusted Applications. In: *9th USENIX Security Symposium (USENIX Security 00)*. Denver, CO: USENIX Association, 2000. Available also from: `https://www.usenix.org/conference/9th-usenix-security-symposium/mapbox-using-parameterized-behavior-classes-confine`.

68. FRASER, T., BADGER, L. and FELDMAN, M. Hardening COTS software with generic software wrappers. In: *Proceedings DARPA Information Survivability Conference and Exposition. DISCEX'00*. 2000, vol. 2, 323–337 vol.2. Available from DOI: 10.1109/DISCEX.2000.821530.
69. KO, Calvin, FRASER, Timothy, BADGER, Lee and KILPATRICKV, Douglas. Detecting and Countering System Intrusions Using Software Wrappers. In: *9th USENIX Security Symposium (USENIX Security 00)*. Denver, CO: USENIX Association, 2000. Available also from: https://www.usenix.org/conference/9th-usenix-security-symposium/detecting-and-countering-system-intrusions-using-software.
70. PROVOS, Niels. Improving Host Security with System Call Policies. In: *12th USENIX Security Symposium (USENIX Security 03)*. Washington, D.C.: USENIX Association, 2003. Available also from: https://www.usenix.org/conference/12th-usenix-security-symposium/improving-host-security-system-call-policies.
71. JABŁOŃSKI, Jedrzej and PAWŁOWSKI, Marcin. *Secure sandboxing solution for GNU/Linux*. 2011. MA thesis. University of Warsaw, Faculty of Mathematics, Informatics and Mechanics.
72. SHAN, Zhiyong, WANG, Xin and CHIUEH, Tzi-cker. Tracer. In: *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security - ASIACCS '11*. 2011, nil. Available from DOI: 10.1145/1966913.1966932.
73. ŇAŇKO, Peter. *Podpora audit systému pre bezpečnostný model Medusa*. 2020. Available also from: https://opac.crzp.sk/?fn=detailBiblioForm&sid=201CE56335A527AB040B96791929. Bc. pr. Ústav informatiky a matematiky, Fakulta elektrotechniky a informatiky Slovenskej technickej univerzity v Bratislave. EČ: FEI-5382-86243.
74. YEOH, Christopher, RUSSELL, Rusty and QUINLAN, Daniel (eds.). *Filesystem Hierarchy Standard* [online]. The Linux Foundation, 2015-03-19 [visited on 2023-04-18]. Available from: https://refspecs.linuxfoundation.org/FHS_3.0/fhs-3.0.pdf.
75. *file-hierarchy — File system hierarchy overview* [online]. 2023. [visited on 2023-04-18]. Available from: https://www.freedesktop.org/software/systemd/man/file-hierarchy.html.
76. KERRISK, Michael. *hier(7) — Linux manual page* [online]. 2021. [visited on 2023-04-18]. Available from: https://man7.org/linux/man-pages/man7/hier.7.html.

77. KERRISK, Michael. *proc(5) — Linux manual page* [online]. 2021. [visited on 2023-04-18]. Available from: `https://man7.org/linux/man-pages/man5/proc.5.html`.
78. NIXOS CONTRIBUTORS. *Nix & NixOS | Reproducible builds and deployments* [online]. 2023. [visited on 2023-04-18]. Available from: `https://nixos.org/`.
79. *GoboLinux - the alternative Linux distribution* [online]. 2022. [visited on 2023-04-18]. Available from: `https://gobolinux.org/`.
80. BERG, Chris van den. *Super Fast String Matching in Python* [online]. 2017-10-14. [visited on 2023-05-03]. Available from: `https://bergvca.github.io/2017/10/14/super-fast-string-matching.html`.

Appendix A

Audited operations

Following list shows types of operations that are audited from Medusa. Operations are grouped together based on their arguments. Arguments `dir`, `old_dir` represent parent folder (*basename*) of the object in question. Arguments `name`, `old_name` represent dentry of the object in question. Argument `path` represents whole path to the object of the operation (dentry and parent directory). Symbol RW next to a list of operations signifies that the operation has read and write access types on the object of the operation. Other operations don't use access types except for `open`, which access type is determined according to the `mode` argument.

- `unlink`, `rmdir` RW
  - `dir`
  - `name`
- `mkdir`, `mknod`, `truncate`, `symlink`, `chmod` RW
  - `dir`
- `link` RW
  - `old_dir`
  - `dir`
- `rename` RW
  - `old_dir`
  - `old_name`
  - `dir`

Audited operations

- `chown`, `path` RW
  - `path`
- `exec`
  - `path`
  - `pid`
- `open`
  - `dir`
  - `mode` (permissions are based on the mode)
- `setresuid`
  - `pid`
  - `euid`

Appendix B

Source code

- Source code of the Medusa policy mining application, along with logs and results that were evaluated in this thesis can be found at `https://github.com/Medusa-Team/medusa-policy-mining-hub`.
- Source code of Medusa kernel is at `https://github.com/Medusa-Team/linux-medusa`.
- Authorization server Constable is available at `https://github.com/Medusa-Team/Constable`.