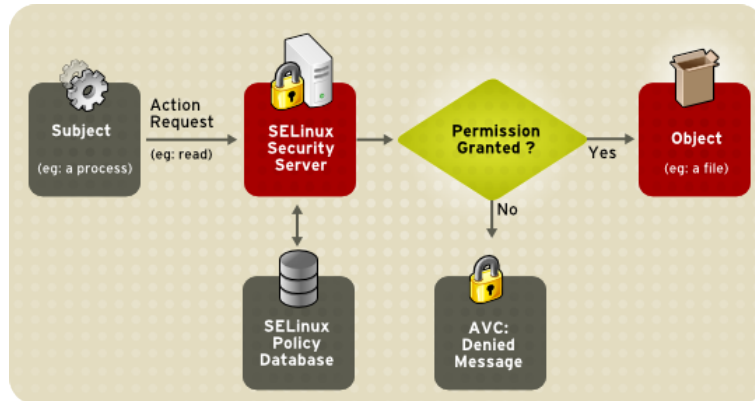


Mandatory Access control in Distributed Environment



By

Patel Nainesh (ID No. - 10CE10)

A Thesis

Submitted in Partial fulfillment of the requirements for the degree of

Master of Technology in Computer Engineering

Internal Guide

PROF. P. M. JADAV

(Associate Prof.)

Department of Computer Engineering



Faculty of Technology

Department of Computer Engineering

Dharmsinh Desai University

April 2013

CANDIDATE'S DECLARATION

I declare that the dissertation for M.Tech. (C.E.) entitled "*Mandatory Access Control in Distributed Environment*" is my own work conducted under the guidance of Prof. P. M. Jadav.

I further declare that to the best of my knowledge, the dissertation for M.Tech. (C.E.) does not contain any part of the work, which has been submitted for the award of any degree either in this University or in other University/Deemed University without proper citation.

Patel Nainesh

Candidate

(ID No. : 10CE10)

Internal Guide

PROF. P. M. JADAV

(Associate Prof.)

Department of Computer Engineering

CERTIFICATE

This is to certify that the dissertation entitled "*Mandatory Access Control in Distributed Environment*" is a bonafied record of the research work carried out by Mr. Nainesh Patel (ID No. : 10CE10) under my guidance and supervision for the partial fulfillment of the award of the degree of Master of Technology of Dharmsinh Desai University, Nadiad, Gujarat.

To the best of my knowledge and belief, the dissertation:

1. Embodies the work of the candidate himself,
2. Has duly been completed,
3. Fulfills the requirements of the ordinance relating to the M.Tech. degree of the University and
4. Is up to the standard in respect of contents, presentation and language for being referred to the examiner.

Prof. P. M. Jadav

Assoc. Prof.,

Dept. of Computer Engg.

Prof. C. K. Bhensdadia

Head,

Dept. of Computer Engg.

Prof. D. G. Panchal

Dean,

Faculty of Technology



Faculty of Technology

Department of Computer Engineering

Dharmsinh Desai University

Abstract

Significant progress toward general acceptance of applying mandatory access control to systems has been made. Security Enhanced Linux (SELinux), in particular, has been enabled by default in some Linux distributions for several years. However, current SELinux deployments only effectively control a single system. Inter-system and remote resource access control capabilities are starting to appear in SELinux, but extending policy management capabilities to cover these networked systems remains an open problem. This Project discusses issues that must be addressed to support security policies distributed across a network, including policy development changes needed to be able to express a coherent security policy for a network of systems, managing the distribution of a multi-system policy, and synchronizing policy updates, and presents a distributed policy management architecture.

Acknowledgements

I could not have completed this work without my Mother, a constant gardener.

I wish to express my sincere thanks to Prof. C. K. Bhensdadia, for providing me with all necessary facilities.

I owe my deepest gratitude to Prof. P. M. Jadav and Mr. Niles Vaghela, for their guidance and support.

I also place on record, my sense of gratitude to one and all who, directly or indirectly, have helped in this thesis.

Last, but not the least I want to give grateful thanks to my institute, Dharmsinh Desai University, for giving me this opportunity to work in this environment. I also want to give my regards to all my school teachers.

Contents

List of Figures	viii
Glossary	x
1 Introduction	1
1.1 Evolution of Access Control Security in Operating Systems	1
1.1.1 Reference Monitor Concept	2
1.1.2 Problem with Discretionary Access Control	3
1.1.3 Concept of Mandatory Access Control	4
1.1.4 Better form of Mandator Access Control	6
1.1.5 Evolution of SELinux	7
1.2 Inadequacies of existing policies	9
1.2.1 Single System Access control	9
1.2.2 Expressing network-wide security goals	10
1.2.3 Analyzing policy domain interactions	10
1.2.4 Handling distributed subject contexts	11

1.2.5	Handling distributed object contexts	12
1.3	Goal	13
2	SELinux Overview	14
2.1	Introduction to SELinux	14
2.1.1	Comparing SELinux with Standard Linux	17
2.1.2	Flask Security Architecture and SELinux	18
2.2	SELinux, an Implementation of Flask	21
2.3	Userspace Object Managers	24
2.3.1	Kernel Support for Userspace Object Managers	25
2.4	Type Enforcement Access Control	26
2.4.1	The Problem of Domaini Transition	29
2.4.2	Review of SetUID Programs in Standard Linux Security	30
2.4.3	Domain Transition	32
2.4.4	Default Domain Transitions: type transition Statement	35
2.5	The Role of Roles	36
2.6	What is Policy	38
2.6.1	Policy Role in Boot	39
2.7	File System Security Contexts	41
3	SELinux Policy	42
3.1	Loadable Policy modules	42

3.1.1	Building and Installing Monolithic Policies	43
3.2	What is the Targeted Policy?	44
3.3	Reference Policy	45
3.3.1	Overview of Policy Source File Structure	46
3.3.1.1	Build and Support Files	47
3.3.1.2	Core Policy Files	48
3.3.2	Design Principles	49
3.3.2.1	Layering	49
3.3.2.2	Modularity	50
3.3.2.3	Encapsulation	50
3.3.2.4	Abstraction	51
3.3.2.5	Module Files	52
3.3.2.6	Interfaces	52
4	Proposed System	54
4.1	Expression of multi-system policy	54
4.1.1	Preserve equivalence where possible	54
4.1.2	Unified namespaces	55
4.1.3	The need for policy templating	55
4.1.4	Partitioning policy	56
4.2	The @ operator	57

4.2.1	Expressing multi-system policies	58
4.2.2	Partitioning policy with the @ operator	59
4.3	Distributed Homogeneous services	59
4.3.1	Automatic service Discovery	60
4.3.2	Gather Data from service and apply to application	61
4.3.3	Homogeneous Service	61
4.3.4	Override Atomic Service Discovery Decision	62
4.3.5	Hierarchy based data distribution	62
4.3.6	Failover	63
4.3.7	Activity Diagrams	64
5	Implementation	66
5.1	Software Requirements	66
5.2	Multi System Policy	67
5.3	Automatic Service Discovery	69
5.4	Client Server Architecture	70
5.4.1	Files & folder used for programming	72
5.4.2	Client algorithm	73
5.4.3	Server algorithm	79
5.5	Client Output	79
5.5.1	Normal client output	79

5.5.2	Client output in case of failover	81
5.6	Server Output	82
5.7	Libraries used by python programs	83
5.7.1	Client Program Libraries	83
5.7.2	Server Program Libraries	83
5.8	Updation of Policy on Destination Node	84
5.9	Override Automatic Discovered Parent	86
6	Conclusion and Future work	87
6.1	Conclusion	87
6.2	Future work	88
	Bibliography	89
A	SELinux Policy Objects	93
A.1	Object Classes and Permissions	93
A.2	TE Rules - Attributes	96
A.3	TE Rules - Types	104
A.4	TE Rules - Access Vectors	106
A.4.1	Understanding avc Message	108
A.5	Policy Macros	111
B	Installation steps	113
B.1	SELinux	113

B.2	Python	113
C	Contents of files	115
C.1	Client Program	115
C.2	Server Program	118
C.3	Avahi Service Entry File	120
C.4	SELinux Policy Updation Script	120
C.5	Host Dictionary file	121
C.6	README	121
C.7	HASH file	122

List of Figures

1.1	Reference Monitor Concept	2
1.2	Multilevel Security Model	5
2.1	LSM Hook Architecture	16
2.2	Flask Architecture	20
2.3	Userspace Object Manager	25
2.4	Depication of Allow Rule	27
2.5	Type Enforcement Example	28
2.6	Problem of Domain Transition	30
2.7	Password Program Security in Standard Linux	31
2.8	Passwd Program Security in SELinux	33
2.9	Roles in Domain Transition	37
3.1	Building Policy	43
4.1	Automatic Service Discovery	60
4.2	Initial Client Server	61

LIST OF FIGURES

4.3	Homogeneous Service	61
4.4	Failover Scenario	63
4.5	Activity Diagram of Initial Server	64
4.6	Activity Diagram of Distributed Service	65
5.1	Multi System Policy	68
5.2	Automatic service Discovery	71
5.3	Experimental Setup	72
5.4	Client Data Transfer Sequence	74
5.5	Step 1	75
5.6	Step 2	75
5.7	Step 3	76
5.8	Step 4 and 5	77
5.9	Step 6,7 and 8	77
5.10	Step 9	78
5.11	Failover	78
5.12	Server Data Transfer Sequence	80
5.13	Update Policy on Client Nodes	85
5.14	Overriding the parent address	86

Glossary

SELinux	Security Enhanced Linux
DAC	Discretionary Access Control
MAC	Mandatory Access Control
MLS	Multilevel Security
TE	Type Enforcement
LSM	Linux Security Module
ACL	Access Control List
RBAC	Role Based Access Control
EA	Extended Attributes
AVC	Access Vector Cache
UID	User Identity
GID	Group Identity
IPC	Interprocess Communication
NSA	National Security Agency
MCS	Multicategory Security

Chapter 1

Introduction

1.1 Evolution of Access Control Security in Operating Systems

Early operating systems had little or no security; a user could access any file or resource just by knowing how to name the resource. Fortunately, it was not long before access control mechanisms began to emerge to provide some sense of security. The predominant type of access control we have today is called discretionary access control (DAC). The primary feature of DAC is that individual users, often a resource “owner,” can specify who may or may not access the resource. As we can see, DAC has some fundamental security weaknesses that are intrinsic to its nature. To overcome these weaknesses, the computer security community has been trying to develop useful mandatory access control (MAC) mechanisms. MAC is intended to avoid the weaknesses of DAC while providing the security required(2). Unfortunately, creating a useful MAC mechanism that is secure yet flexible enough to address a wide range of problems has proven difficult. The primary value that SELinux brings to Linux is a flexible, configurable MAC mechanism. In the remainder of this section, we explore the strengths and weaknesses of various DAC and MAC mechanisms, as a means to

provide a context for understanding the true value that SELinux provides.

1.1.1 Reference Monitor Concept

In a reference monitor(2), the operating system isolates passive resources into distinct objects such as files and active entities such as running programs into subjects. The reference monitor mechanism (called a reference validation mechanism) would then validate access between subjects and objects by applying a security policy as embodied in a set of access control rules. In this manner, program access to system resources such as files can be limited to those accesses that accord with the security policy. Access control decisions are based on security attributes associated with each subject and object that represents the subject's/object's security-related characteristics. For example, in standard Linux, subjects (that is, processes) have real and effective user identifiers, and objects (for example, files) have access permission modes that are used to determine whether a process may open a file.

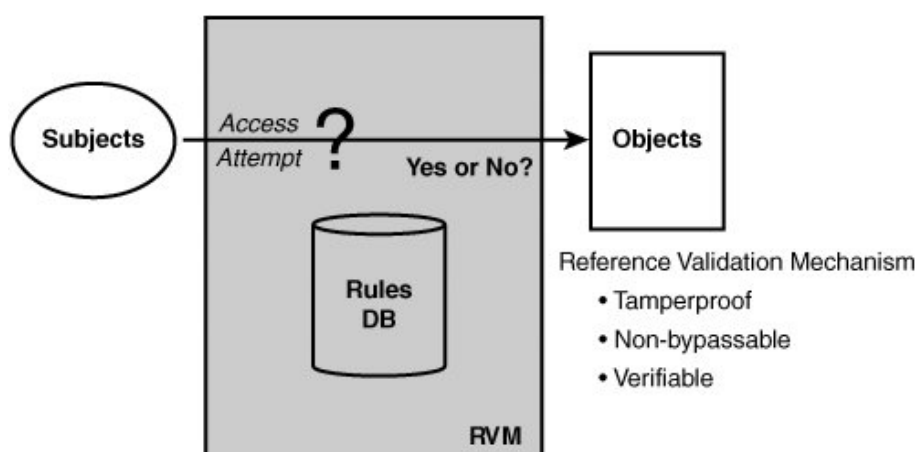


Figure 1.1: Reference Monitor Concept - Basic Validation Mechanism

Other than implementing the security policy, the fundamental design goals of an implementation of the reference monitor concept are that it be:

- Tamper-proof (cannot be maliciously changed or modified)

1.1 Evolution of Access Control Security in Operating Systems

- Nonbypassable (subjects cannot avoid the access control decisions)
- Verifiable (it is correct and implementation of the security policy can be demonstrated)
- Nearly all operating systems implement some form of a reference monitor and can be characterized in terms of subjects, objects, and security policy rules.

In standard Linux, subjects are generally processes, and objects are the various system resource used for information sharing, storage, and communication (files, directories, sockets, shared memory, and so on). In Linux, as in most other popular operating systems, the security policy rules enforced by the reference monitor (that is, the kernel) are fixed and hard-coded, whereas the security attributes that these rules use for validation (for example, access modes) can be changed and assigned. Standard Linux security is a form of DAC security(13).

1.1.2 Problem with Discretionary Access Control

As noted, DAC is a form of access control that usually allows authorized users (via their programs such as a shell) to change the access control attributes of objects, thereby specifying whether other users have access to the object. A simple form of DAC might be file passwords, where access to a file requires the knowledge of a password created by the file owner (and distributed by word of mouth to other users authorized to view the file). Most DAC mechanisms are based on user-identity access control attributes. Nearly all modern operating systems have some form of user-identity-based DAC(13). In Linux, the owner-group-world permission mode mechanism is prevalent and well known. Likewise, a more general access control list mechanism is also common.

All DAC mechanisms have a basic weakness in that they fail to recognize a fundamental difference between human users and computer programs. DAC typically tries to emulate an ownership concept where; for example, file owners have the right to

1.1 Evolution of Access Control Security in Operating Systems

specify access to files and only give access to other users they trust to access the file. Assuming that we can trust the human user (arguably an invalid proposition in general), the way computers work does not directly model the real world. Simply put, users rely on software, not of their own creation, to perform functions on the computer. So, we are not really giving users the ability to grant and use access. Instead, we are giving software programs this capability. As has become obvious in the age of the Internet, programs are often full of flaws or are downright malicious. This is the problem with Trojan horses, first recognized in the 1970s, of which today's modern viruses, worms, and spyware are just variants. In short, if a user is authorized access, that really means programs are authorized that access, and if programs are authorized that access, malicious programs will have that same access.

DAC assumes a benign environment where all programs are trustworthy and without flaws(2). Although the early computer research community, which largely lived in an academic world and from which so much of our current technology evolved, might have wished for such an environment; in reality, however, we know of no such benign computer environment in the entire history of computer science. Human nature will always have those who exploit weakness in flawed software.

1.1.3 Concept of Mandatory Access Control

Throughout the 1970s and 1980s, significant energy was exerted to address the problem of malicious and flawed software. The goal was to achieve MAC, where the basis of access control decisions was not at the discretion of individual users or even system administrators. We wanted to implement an organizational security policy to control access to objects that could not be affected by the actions of individual programs(14). The military funded most of this work, which focused on protecting the confidentiality of classified government data. In particular, the most common MAC mechanisms implemented to date address the problem of multilevel security, a simplified form of which is shown in Figure 1.2

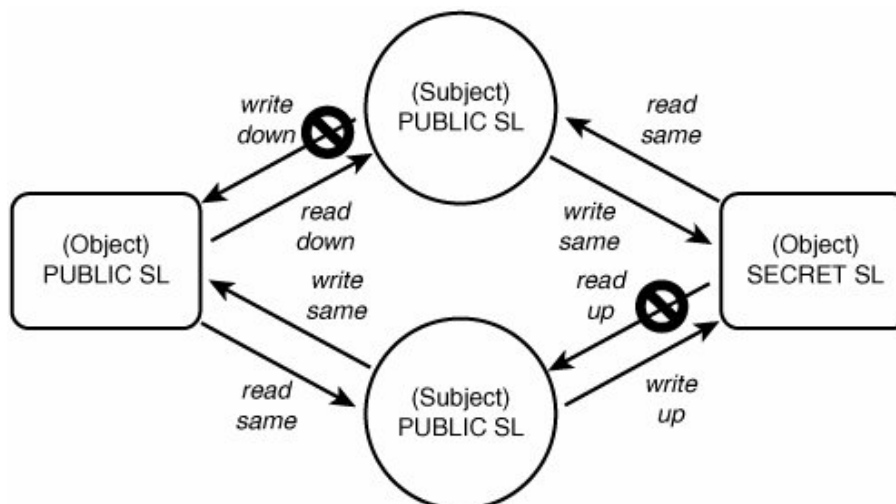


Figure 1.2: Multilevel Security Model - Subject and Object Based Validation

Multilevel security (MLS) is typically based on a formal model called the Bell-LaPadula model(15). In the MLS model, all subjects and objects are labeled with a security level. In our example, we have a PUBLIC and a SECRET sensitivity level. The levels represent the relative sensitivity of the data and the clearance of the user on whose behalf the subjects are operating (SECRET being data of “higher” sensitivity than PUBLIC). In MLS, subjects can always read and write objects at the same sensitivity. In addition, subjects can read lower-level objects (“read down”) and write higher-level objects (“write up”). However, a subject may never read higher-level objects (“no read up”) nor write lower-level objects (“no write down”). The idea being that information can flow from lower levels to higher levels, but not the reverse, thereby protecting the confidentiality of the higher-level data.

MLS was a radical change in the way we thought about access control. No longer are data owners arbitrarily determining who may access objects. Further, we could now have strong security assuming most software was untrusted, because the information flow rules prevent inappropriate data access. In MLS, the organization decides via fixed rules how data may be shared regardless of the desires of individual users (and more important, the programs they run). MLS is by far the most implemented MAC mechanism to date and is still prevalent in several niche operating systems. MAC

1.1 Evolution of Access Control Security in Operating Systems

mechanisms similar to MLS have also been contemplated and built, all of which share a common theme of implementing a small number of fixed security properties. The primary weakness of MLS is the fact that it implements a single security goal (that is, protecting the confidentiality of sensitive data using the model of government classified documents) in a strict, inflexible manner. Not all operating system security concerns are related to data confidentiality, and of those that are, most are not amenable to the rigid and simple model of classified government documents (including many, if not most, government systems dealing with classified data). To expand upon this goal in MLS (and similar MAC mechanisms), subjects must be given privilege to work outside the security policy (that is, violating the principle of nonbypassability) and trusted not to violate the intent of the policy. This inflexibility and narrow focus has kept MLS and similar MAC mechanisms from achieving broad appeal(2).

1.1.4 Better form of Mandator Access Control

SELinux implements a flexible MAC mechanism called type enforcement (TE). As we can see, type enforcement provides strong mandatory security in a form that is adaptable to a large variety of security goals, concurrently. Type enforcement provides a means to control access down to the individual program level, in a manner that allows an organization to define a security policy appropriate for their systems(15). In type enforcement, all subjects and objects have a type identifier associated with them. To access an object, the subject's type must be authorized for the object's type, regardless of the user identity of the subject. What makes the SELinux approach superior to a straight MLS solution is that the rules governing type-based access control are not predefined nor hard-coded in the kernel. By default, SELinux allows no access. An organization can develop any number of rules specifying what is allowed, making SELinux adaptable to a wide variety of security policies.

The collection of rules that determine allowed access for a system is called an SELinux policy. Physically, an SELinux policy is a special file that contains all the

1.1 Evolution of Access Control Security in Operating Systems

rules that the SELinux kernel will enforce. The policy file is compiled from a set of source files. As we can see, SELinux policies can vary from system to system. During the boot process, the policy is loaded into the kernel, where it is then used as the basis for access control decisions.

SELinux brings flexible type enforcement along with a form of role-based access control and the optional addition of traditional MLS to Linux. This flexible and adaptable MAC security, built in to the mainstream Linux operating system, is what makes SELinux such a promising technology for improved security.

1.1.5 Evolution of SELinux

SELinux has its origins in high-assurance operating system security and microkernel research from the 1980s. These two research threads came together in a project called Distribute Trusted Mach (DTMach), which merged the experiences of earlier research projects (LOCK, which involved a form of type enforcement in a high-assurance security kernel; and Trusted Mach, which incorporated multilevel security controls into the Mach microkernel). The U.S. National Security Agency's research organization participated in the DTMach effort and continued its participation through a number of subsequent secure microkernel projects. This work eventually resulted in a new security architecture, called Flask, that supported a more flexible and dynamic type of enforcement mechanism.

The various platforms upon which this work was performed were research microkernels not in wide market use. The NSA recognized a need to expose this technology to a broader community in hopes of demonstrating its viability and gaining broader support for its use. In the summer of 1999, the NSA began to implement the Flask security architecture(12) in the Linux kernel. In December 2000, the NSA made its first public release of this work, called Security Enhanced Linux. Being implemented in a popular mainstream operating system, SELinux started to attract the attention of

1.1 Evolution of Access Control Security in Operating Systems

the Linux community. SELinux was originally released as a collection of kernel patches for the 2.2.x kernel.

Following the 2001 Linux Kernel Summit in Ottawa, Canada, the Linux Security Module (LSM) project was started to create a flexible framework for the Linux kernel that allowed different security extensions to be added to Linux. The NSA and the SELinux community were major contributors to this effort, with SELinux helping to drive many of the requirements for LSM. Concurrent with the LSM effort, NSA started to adapt SELinux to use the LSM framework. The core LSM features were integrated into the mainline Linux kernel starting in August 2002, and were incorporated into the Linux 2.6 kernel. By August 2003, the NSA, with growing open source community help, had completed its migration of SELinux to the LSM framework, resulting in the inclusion of SELinux in the main Linux 2.6 kernel. SELinux had become a fully functional LSM module included in the core Linux code set.

Several Linux distributions began using the SELinux features in the 2.6 kernel to various degrees, but the primary effort to make SELinux ready for the enterprise was via the Red Hat-sponsored Fedora Core project. The NSA and Red Hat started a joint effort to integrate SELinux as part of the mainstream Fedora Core Linux distribution. Prior to Red Hat's interest, SELinux was always an add-on set of packages that required significant expertise to integrate. Red Hat took the initiative (and business risks) to make SELinux a part of a mainstream distribution(2), complete with modified user-space tools and services and enhanced security enabled by default. Starting with Fedora Core 2 and continuing with Fedora Core 3, SELinux and its supporting infrastructure and tools were improved for mainstream use. In early 2005, Red Hat released its Enterprise Linux version 4 (RHEL4) with SELinux as a fully enabled by default security enhancement. SELinux and mandatory access control had reached the mainstream operating system market at last.

1.2 Inadequacies of existing policies

Existing policies are created for a single system, i.e., they only express access control for subjects and objects on one system, whether it be an installation on a computer, in a virtual machine, or on an embedded device. The single system policy may specify network access controls, but it does not explicitly control access on subjects or objects of another system. Since the security contexts (the attributes used to identify subjects and objects to the security system) were confined to a single system policy development and analysis was always on a per-system basis(1). With the security contexts being extended to a network of systems, a single system policy no longer is sufficient.

1.2.1 Single System Access control

Before delving into the challenges of multi-system access control, it is important to define existing access control in SELinux systems. Access control systems specify the allowed access between subjects, which are the active entities i.e., processes on the system, and objects, which are the passive entities on the system such as files. With type enforcement (TE), the primary access control mechanism in SELinux, every subject and object is given a context that comprises the security attribute used to determine what access is allowed(1).

The Flask framework upon which SELinux is based already implements distributed access control within a single system; each object manager controls its own objects and asks the security server for access control decisions. In this way, the enforcement itself is distributed throughout the system. Distributed Trusted Operating System (DTOS) provides a good example of distributed access control since it is a microkernel architecture(24) and therefore has object managers running in separate memory spaces. SELinux uses the same concepts but most of the object managers are part of the monolithic kernel. Notable exceptions are the user space object managers X Windows, DBUS, and passwd, all of which control their own objects.

Since the underlying mechanism can already support distributed access control much of the initial work to support this has already been done. The effort then falls on effectively scaling this mechanism to entire networks where before it was only effectively applied to single systems(1).

1.2.2 Expressing network-wide security goals

The first step in policy development is that of determining security goals the policy must satisfy. Existing SELinux policy assumes these goals relate to single systems and network borders. However, with security becoming more of a concern on networks, the capability to define and enforce network-wide security goals must become prevalent. A network-wide security goal may involve separating internal only and external data or limiting data access to customers and employees or even locking down workstations(1). One must be able to express network-wide security goals without manually determining how to enforce the goals on a per-system basis, as well as writing and distributing the policies to those systems.

Trying to develop a network-wide security policy on a per-system basis may be difficult, and the developmental effort to keep per-system policies consistent with network-wide security goals may be great. However, the most significant problem is that analysis of the policy composed of various per-system policies after development may simply be impossible. Policy analysis relies on certain aspects of the policy being true, namely that security contexts are unique equivalence classes. If a single context means something different on two different systems, analysis becomes problematic.

1.2.3 Analyzing policy domain interactions

Types are defined as unambiguous security equivalence classes, and are the primary security attribute used by SELinux in the TE security model. Every subject and object has a type, and any subject or object with the same type is treated identically

by SELinux. When performing policy analysis, the TE mechanism and therefore the types are the primary consideration.

On a single system, suppose there are two Apache web server instances, one that was for internal use only and one that was for external use. Each Apache would need different types; otherwise the internal only Apache server could be accessible externally. Separating them into different types ensures that through analysis the policy will not allow internal Apache instances to be accessible externally. If the two web server instances reside on different machines and have the same type, even though they are on different systems this guarantee cannot be made.

In some cases, systems are identically configured. In cluster systems, for example, each system would have the same policy and it is likely that each instance of an application would share the same type. This is what gives SELinux most of its flexibility. Subjects that need to be treated the same use the same type; otherwise they get a different one(1). Obviously, one can have some portions that should be treated identically and other portions that must be handled distinctly across systems in the Apache example there may be an internal and external Apache that must be handled differently, but every DNS server may be the same and so the same type would be used for each of them.

In all cases, to analyze a network policy requires a single policy that covers all systems being protected. An analysis of the policy then can ensure the same guarantees that analyzing a single system policy can.

1.2.4 Handling distributed subject contexts

MAC has been used across systems for quite some time. Systems using the Bell LaPadula (BLP) model have been able to propagate contexts to other systems for enforcement using technologies such as CIPSO and RIPS0. These technologies are of limited

use for SELinux, however, as they were designed for Multi-Level Security (MLS) systems, which have inflexible policies and homogeneous security contexts(1). Supporting flexible MAC introduces a whole new set of issues that must be addressed.

Recent changes to SELinux have introduced the ability to propagate contexts to other systems. Using IPSEC protects both the integrity and confidentiality of the communication between SELinux machines and the context that is propagated. While this is a very significant advancement, it also means additional policy development and analysis effort. The single system policy development model is illustrated to be further insufficient in the networked environment, since a context that is used to access other systems may have different security properties than the local equivalent would have.

1.2.5 Handling distributed object contexts

In addition to subject contexts being used across systems, labeled network file systems bring the possibility of object contexts traversing the network. This means the network policy need not only treat subjects distinctively but also objects. Using the previous Apache example, if the two Apache instances are serving different data internally and externally, the data would need different types(1). This prevents the external Apache from accidentally or maliciously disclosing internal data whether it resides on the same system or a labeled network file system gets mounted to the system on which it runs.

Again, on single systems, assuming each Apache instance is on a different system, this would be unnecessary; on a distributed system the data and its context can be accessed on different systems, however. It would likely violate the policy domains security goals if the external web server could access the internal data, even if the data is on a network file system that is accessible by the external server. A network-wide policy must take these matters into consideration.

1.3 Goal

Distributed systems are becoming much more prevalent as cheaper hardware and technologies such as hypervisors make it more economical to run many systems for redundancy and integrity than in the past.

Current SELinux policy, assumes that all systems running the software protected by the policy are relatively the same. This may be a fair assumption for isolated systems but the ability to enforce inter-system access breaks that assumption. Therefore, the existing concept of security policy must be extended to handle a network of systems that belong within a single policy domain(1), but in which similar applications may be treated differently depending on the system upon which they run or the manner in which they are accessed (e.g., locally vs. from a remote machine). Within a network of systems, there will be many security servers, each responsible for providing security decisions to a set of object managers, each responsible for enforcing security decisions on their objects. The objects can also be very similar, for example `/etc/shadow` on system 1 and `/etc/shadow` on system 2 are different objects controlled by different object managers. Enforcing access control within a network of systems also introduces new challenges in policy distribution. Relevant portions of the coherent policy must be sent to the individual systems without unnecessary overhead. To this end, the policy must be partitioned and only the appropriate sections sent to individual systems. Finally, extra care must be taken in ensuring the atomicity of policy changes. While this is a concern on single systems, the problem is amplified when many systems must enforce a coherent policy on a high latency medium.

To effectively apply mandatory access control to networks and distributed systems, the single system policy paradigm must evolve. Furthermore, all policy distribution must be synchronized so that the policy across all systems is consistent at any given time. Finally, although the need to express distributed policies has prompted augmentation to the SELinux infrastructure and policy language, these changes will greatly facilitate the process of developing, distributing, and enforcing network policies.

Chapter 2

SELinux Overview

2.1 Introduction to SELinux

All operating system access control is based on some type of access control attribute associated with objects and subjects. In SELinux, the access control attribute is called a security context. All objects (files, interprocess communication channels, sockets, network hosts, and so on) and subjects (processes) have a single security context associated with them. A security context has three elements: user, role, and type identifiers. The usual format for specifying or displaying a security context is as follows:

```
user:role:type
```

The string identifiers for each element are defined in the SELinux policy language, which we discuss in greater detail later. For now, just understand that a valid security context must have one valid user, role, and type identifier, and that the identifiers are defined by the policy writer. The namespaces for each identifier are orthogonal. (So, for example, it is possible, but not usually advisable, to have the same string identifier for a user, a role, and a type.)

Security-enhanced Linux (SELinux) is an implementation of a mandatory access

control mechanism. This mechanism is in the Linux kernel, checking for allowed operations after standard Linux discretionary access controls are checked. To understand the benefit of mandatory access control (MAC) over traditional discretionary access control (DAC), we need to first understand the limitations of DAC. Under DAC, ownership of a file object provides potentially crippling or risky control over the object. A user can expose a file or directory to a security or confidentiality breach with a misconfigured `chmod` command and an unexpected propagation of access rights. A process started by that user, such as a CGI script, can do anything it wants to the files owned by the user. A compromised Apache HTTP server can perform any operation on files in the `Web` group. Malicious or broken software can have root-level access to the entire system, either by running as a root process or using `setuid` or `setgid`. Under DAC, there are really only two major categories of users, administrators and non-administrators.

In order for services and programs to run with any level of elevated privilege, the choices are few and course grained, and typically resolve to just giving full administrator access. Solutions such as ACLs (access control lists) can provide some additional security for allowing non-administrators expanded privileges, but for the most part a root account has complete discretion over the file system. A MAC or non-discretionary access control framework allows us to define permissions for how all processes (called subjects) interact with other parts of the system such as files, devices, sockets, ports, and other processes (called objects in SELinux). This is done through an administratively-defined security policy over all processes and objects. These processes and objects are controlled through the kernel, and security decisions are made on all available information rather than just user identity. With this model, a process can be granted just the permissions it needs to be functional. This follows the principle of least privilege. Under MAC, for example, users who have exposed their data using `chmod` are protected by the fact that their data is a kind only associated with user home directories, and confined processes cannot touch those files without permission and purpose written into the policy.

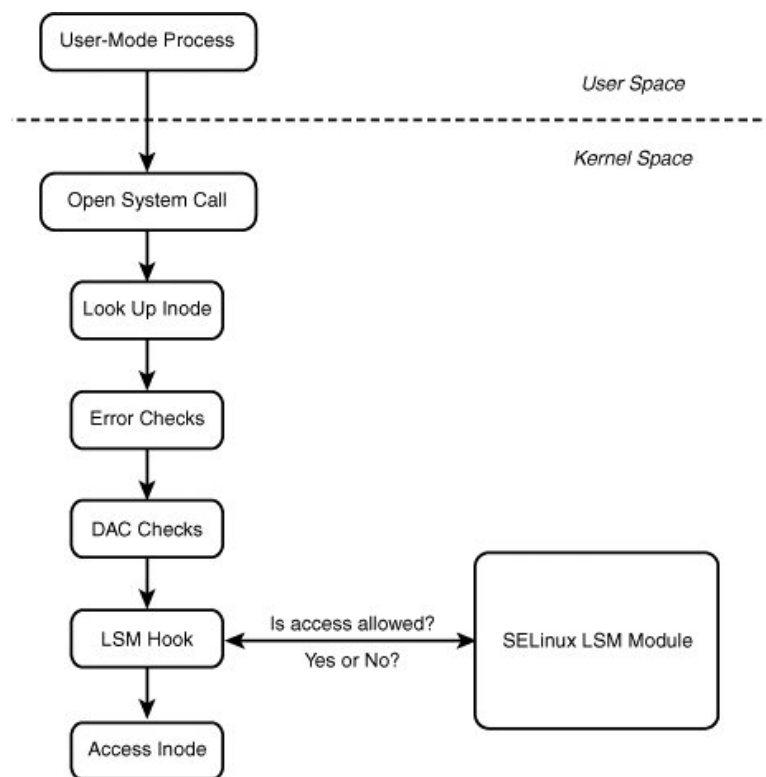


Figure 2.1: LSM Hook Architecture - Linux security Modules

SELinux is implemented in the Linux kernel using the LSM (Linux Security Modules) framework. To support fine-grained access control(11), SELinux implements two technologies: Type EnforcementTM (TE) and a kind of role-based access control (RBAC)(19). Type Enforcement involves defining a type for every subject, that is, process, and object on the system. These types are defined by the SELinux policy and are contained in security labels on the files themselves, stored in the extended attributes (xattrs) of the file. When a type is associated with a processes, the type is called a domain, as in, `httpd` is in the domain of `httpd_t`. This is a terminology difference leftover from other models when domains and types were handled separately. All interactions between subjects and objects are disallowed by default on an SELinux system. The policy specifically allows certain operations. To know what to allow, TE uses a matrix of domains and object types derived from the policy. The matrix is derived from the policy rules.

For example, `allow httpd_t net_conf_t:file { read getattr lock ioctl };` gives the domain associated with `httpd` the permissions to read data out of specific network configuration files such as `/etc/resolv.conf`. The matrix clearly defines all the interactions of processes and the targets of their operations. Because of this design, SELinux can implement very granular access controls. For Red Hat Enterprise Linux 4 the policy has been designed to restrict only a specific list of daemons. All other processes run in an unconfined state. This policy is designed to help integrate SELinux into our development and production environment. It is possible to have a much more strict policy, which comes with an increase in maintenance complexity.

2.1.1 Comparing SELinux with Standard Linux

At this point, it is useful to compare the access control attributes on standard Linux with those of SELinux. For simplicity, we stick to common filesystem objects such as files and directories. In standard Linux, the access control attributes of subjects are the real and effective user and group IDs associated with all processes via the

process structure in the kernel. These attributes are protected by the kernel and set via a number of controlled means, including the login process and `setuid` programs. For objects (for example, files), the inode of the file contains a set of access mode bits and file user and group IDs. The former controls access based on three sets of read/write/execute bits, one each for file owner, file group, and everyone else. The latter determines the file owner and group to decide which set of bits to use on a given access attempt. As noted, in SELinux, the access control attributes are always the security context triple. All objects and subjects have an associated security context. Where standard Linux uses the process user/group IDs, the file's access mode, and the file user/group IDs to grant or deny access, SELinux uses the security contexts of a process and the object the process accesses. More specifically, because the primary access control feature of SELinux is type enforcement, the type identifier from the security context is used to determine access.

2.1.2 Flask Security Architecture and SELinux

Flask was developed to work through some of the inherent problems with a MAC architecture. Traditional MAC is closely integrated with the multi-level security (MLS) model. Access decisions in MLS are based on clearances for subjects and classifications for objects, with the objective of no read-up, no write-down . This provides a very static lattice that allows the system to decide by a subjects security clearance level which objects can be read and written to. The focus of the MLS architecture(12) is entirely on maintaining confidentiality. The inflexible aspect of this kind of MAC is the focus on confidentiality. The MLS system does not care about integrity of data, least privilege, or separating processes and objects by their duty, and has no mechanisms for controlling these security needs. MLS is a mechanism for maintaining confidentiality of files on the system, by making sure that unauthorized users cannot read from or write to them.

Flask solves the inflexibility of MLS-based MAC(16) by separating the policy enforcement from the policy logic, which is also known as the security server. In traditional Flask, the security server holds the security policy logic, handling the interpretation of security contexts. Security contexts or labels are the set of security attributes associated with a process or an object. Such security labels have the format of `user : role : type`, for example, `system_u:object_r:httpd_exec_t`. The SELinux user `system_u` is a standard identity used for daemons. The role `object_r` is the role for system objects such as files and devices. The type `httpd_exec_t` is the type applied to the `httpd` executable `/usr/sbin/httpd`. Prior to full integration with the Linux kernel, security contexts were maintained separately in a file as a set of security identifiers or SIDs. Part of the change when moving to the Linux 2.6.x kernel is the usage of extended attributes (EAs) in the file system. SIDs are not entirely retired, but they are no longer exported to userspace from the kernel. For example, the kernel has some initial SIDs used by `init` during bootstrapping before the policy is loaded. In addition, `libselinux` provides a userspace SID abstraction for applications that enforce policy, such as `dbus-daemon` and `nscd`. Otherwise, users and other programs only interact with security contexts. To minimize confusion, from here forward in this guide, the term security context is used to include the SID. The security server need only do a look-up with a pair of contexts on a matrix of type-labeled subjects and objects, and the result is put in the access vector cache (AVC) for retrieval on subsequent matching requests. By adding in a generalized form of TE that is separated into its own security subsystem, Flask can be flexible in labeling for transition and access decisions.

Instead of being tied to a rigidly defined lattice of relationships, Flask can define other labels based on user identity (UID), role attributes, domain or type attributes, MLS levels, and so forth. Similarly, access decision computations can be made using multiple methods in the same decision. These methods could be lattice models, static matrix lookups, historical decisions, environmental decisions, or policy logic obtained in real time. These computations are all handled by the policy engine and cached, leaving the policy enforcement code available to handle requests. One other Flask

flexibility is that any of these subsystems can be swapped out for a new or different system, and none of the other systems are even aware of the change. The abstraction between policy enforcement and policy decision-making is what makes this possible. This flexibility gives Red Hat Enterprise Linux developers the control they need to make the best architecture decisions without being tied to a particular subsystem.

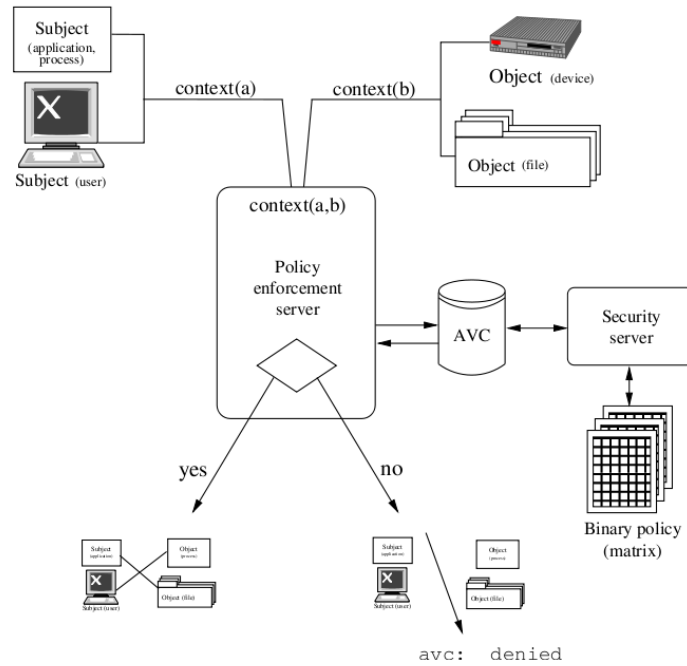


Figure 2.2: Flask Architecture - Core Concept of SELinux

Figure 2.2 describes the Flask architecture(7), showing the process of an operation. In this operation, standard DAC has occurred, which means the subject already has gained access to the object via regular Linux file permissions based on the UID1. The operation can be anything: reading from or writing to a file/device, transitioning a process from one type to another type, opening a socket for an operation, delivering a signal call, and so forth.

1. A subject, which is a process, attempts to perform an operation on an object, such as a file, device, process, or socket.

2. The policy enforcement server gathers the security context from the subject and object, and sends the pair of labels to the security server, which is responsible for policy decision making.
3. The policy server first checks the AVC, and returns a decision to the enforcement server. If the AVC does not have a policy decision cached, it turns to the security server, which uses the binary policy that is loaded into the kernel during initialization. The AVC caches the decision, and returns the decision to the enforcement server, that is, the kernel(4).
4. If the policy permits the subject to perform the desired operation on the object, the operation is allowed to proceed.
5. If the policy does not permit the subject to perform the desired operation, the action is denied, and one or more `avc: denied` messages are logged to `$AUDIT_LOG`, which is typically `/var/log/messages` in Red Hat Enterprise Linux.

With the security server handling the policy decision making, the enforcement server handles the rest of the tasks. In this role, we can think of the enforcement code as being an object manager. Object management includes labeling objects with a security context, managing object labels in memory, and managing client and server labeling.

2.2 SELinux, an Implementation of Flask

SELinux has been through several iterations as part of the process of being incorporated into the Linux kernel. During this time, the overall architecture has remained the same, but many of the programmatic details have changed. Some of the reasons for change were: requirements for upstream acceptance; changes in LSM as part of being accepted into the kernel; and the switch to using `xattrs`. As one example of the changes between kernel versions, originally security context was maintained through a mapping from context to SID, and managed by the security server. In the 2.6.x Linux kernel,

2.2 SELinux, an Implementation of Flask

the security context for a file is stored in the `xattrs`, allowing it to carry around its own SELinux context. As an implementation of the Flask architecture(7), SELinux also served as a reference implementation of LSM. Originally LSM and SELinux were patches to the 2.4. x series of kernels; SELinux was never able to work as a loadable security module. Therefore, a big part of gaining upstream acceptance into the mainline Linux kernel required everything from fixing coding practices to changing how SELinux interacted with the kernel.

Part of the SELinux development team was also instrumental in designing, building, and integrating LSM into the kernel. SELinux integration into the kernel was the motivation to start the LSM project. SELinux was an early proof of the ability of LSM to allow security-enhancements to be connected into, instead of strapped onto, the Linux kernel. Originally, SELinux was a loadable module, but it became statically compiled into the 2.6.x kernel. It is still an LSM module, using the LSM hooks in the kernel to control and label. Because of the abstraction layer provided by both the LSM and Flask frameworks, SELinux is highly configurable and modifiable. Flask is flexible enough to work in many different environments, and Linux is a natural fit for the Flask model. Access to the kernel source and a willing, community-driven development process allow for the best modification to fully support Flasks objectives. The wide range of platforms Linux runs on means SELinux is extensively tested. The consensus process of getting SELinux integrated into the kernel has improved the code and practices. Now that it is integrated, it has a better chance of long-term success than security-enhancement models that are strapped on-top of the operating system. There are a few more differences in the specific way SELinux implements Flask in the Linux kernel, compared to traditional Flask methodology and initial SELinux creation:

1. Under traditional TE, there is a distinction between types and domains. A type is the security context for a file object, and a domain is the security context for a process. In the SELinux implementation, there is no real distinction programmatically. In SELinux, domains are processes that have the attribute process,

so the term domain is used in the traditional way. Similarly, the term type is mostly applied to object types, but it can mean both domains and types.

2. The term security server is still used for the sake of clarity, but it is no longer a stand-alone service. The security server, the AVC, and the policy engine are now all parts of the kernel.

The Flask design makes a strong distinction between security policy decision making and enforcement functions. Policy decision making is the job of the security server. The name security server reflects SELinux's micorkernel roots, where the policy decision role was encapsulated in a userspace server. In Linux, the security server for kernel objects is located in the SELinux LSM module. The policy used for the security server is embodied in a set of rules that is loaded via the policy management interface. These rules can differ from system to system, making SELinux highly adaptable to various organizational security goals. The architecture is designed such that the security server could be completely replaced with logic that implements an entirely new access control policy without changing the rest of the architecture. In practice, new security servers are not needed because type enforcement provides sufficient flexibility for almost any access control security policy.

Object managers are responsible for enforcing the policy decisions of the security server for the set of resources they manage. For the kernel, we can think of object managers as kernel subsystems that create and manage kernel-level objects. Examples of kernel object managers include the filesystem, process management, and System V interprocess communication (IPC). In the LSM architecture, the object managers are represented by the LSM hooks; these hooks are scattered throughout the kernel subsystems and call the SELinux LSM module for access decisions. The LSM hooks then enforce those decisions by allowing or denying access to the kernel resource. The third component of the SELinux architecture is the access vector cache (AVC). The AVC caches decisions made by the security server for subsequent access checks and thus provides significant performance improvements for access validation. The AVC

also provides the SELinux interfaces for the LSM hooks and hence with the kernel object managers.

The AVC is invalidated when a policy is loaded, thereby keeping the cache coherent. However, SELinux does not fully implement access revocation on policy change. This is no worse than standard Linux, which does not access revocation at all. In standard Linux, if we have a file descriptor, we can use it regardless of the change in file access mode. In SELinux, for objects such as files where access is validated on all attempts to use (for example, every read system call is checked against the policy and not just open calls), access revocation works fine. Just having a file descriptor does not mean access to the file will be granted. For some resources, however, such as memory mapped-files and connection-oriented sockets, access is validated only when the resource is initially accessed and not on subsequent use. In these cases, existing access is not revoked. We expect that there will be further research to improve access revocation in SELinux.

2.3 Userspace Object Managers

One of the powerful features of the SELinux architecture is that it can be applied to userspace resources and to kernel resources. Indeed, its origins were in microkernel(8) research where most resource management was performed by userspace servers. Examples of userspace servers in Linux that can be enhanced to enforce access control over their resources include the X server and database services. Each of these servers provides abstract resources (windows, tables, and so on) over which mandatory security could be provided. This section examines two ways that the SELinux architecture supports userspace servers.

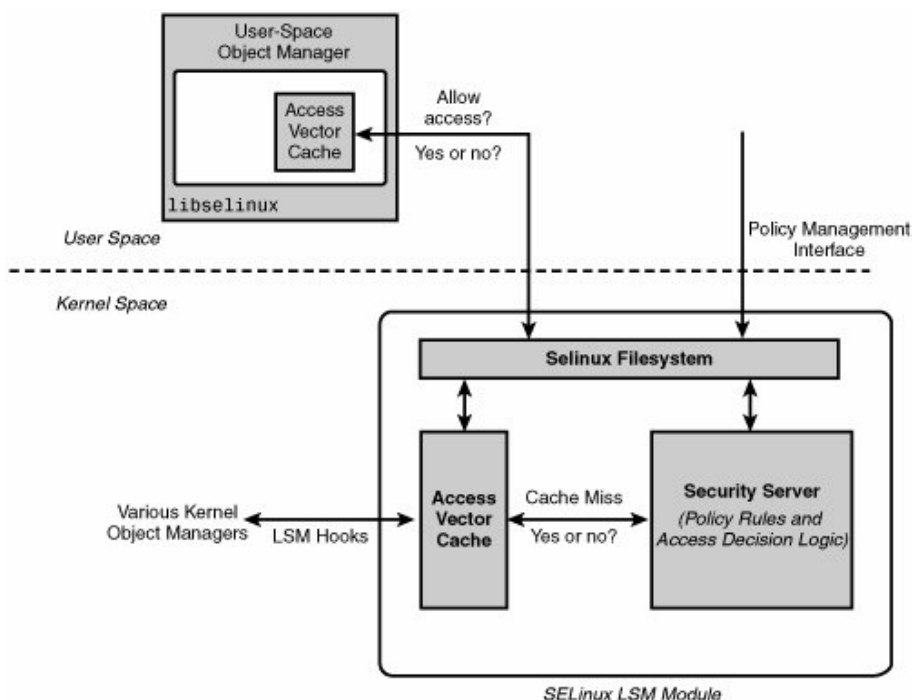


Figure 2.3: Userspace Object Manager - SELinux and userspace objects

2.3.1 Kernel Support for Userspace Object Managers

A simple way SELinux supports userspace objects is directly via the kernel security server, as depicted in Figure 2.3

In this method, the userspace object manager behaves much like the kernel object managers. The kernel security server contains the entire security policy, and the userspace object manager must query the kernel for access control decisions. The primary difference is that userspace object managers cannot use the kernel AVC. Each server must have its own, separate AVC that stores the past decisions it has requested from the kernel. The AVC functionality for userspace servers is contained in the library `libselinux`. Another difference is that userspace object managers do not have LSM hooks, which are a kernel-space concept. Instead, the object manager has internal interfaces with its AVC inside `libselinux`. The AVC handles cache misses and queries the kernel on behalf of the object manager. Although straightforward, this method for supporting userspace object managers has a number of weaknesses.

First, to use type enforcement, object managers must define object classes that represent their resources. For example, a database server might define object classes that include database, table, schema, entry, and so on. For kernel resources, object classes are fixed and correspond to hard-coded class offsets defined in SELinux LSM module header files. The relationship of class definitions in the policy and with those in the kernel code results in an unfortunate dependency between the userspace policy and the code. Specifically, two userspace servers must be careful not to both use the same object class offset in the kernel. The kernel provides no way to manage this possible conflict. The second weakness with this approach is that kernel security server is managing policy for object classes for object managers that are not in the kernel. This increases storage cost within the kernel for abstraction not related to the kernel and can negatively impact the cost of kernel policy validation for AVC misses.

2.4 Type Enforcement Access Control

In SELinux, all access must be explicitly granted. SELinux allows no access by default, regardless of the Linux user/group IDs. Yes, this means that there is no default superuser in SELinux, unlike root in standard Linux. The way access is granted is by specifying access from a subject type (that is, a domain) and an object type using an allow rule. An allow rule has four elements:

- Source type(s) Usually the domain type of a process attempting access
- Target type(s) The type of an object being accessed by the process
- Object class(es) The class of object that the specified access is permitted
- Permission(s) The kind of access that the source type is allowed to the target type for the indicated object classes

2.4 Type Enforcement Access Control

As an example, take the following rule: `allow user_t bin_t : file {read execute getattr};` This example shows the basic syntax of a TE allow rule. This rule has two type identifiers: the source (or subject or domain) type, `user_t`; and the target (or object) type, `bin_t`. The identifier `file` is the name of an object class defined in the policy (in this case, representing an ordinary file). The permissions contained within the braces are a subset of the permissions valid for an instance of the `file` object class. The translation of this rule would be as follows: A process with a domain type of `user_t` can read, execute, or get attributes for a file object with a type of `bin_t`. As we discuss later, permissions in SELinux are substantially more granular than in standard Linux, where there are only three (`rwX`). In this case, `read` and `execute` are fairly conventional; `getattr` is less obvious. Essentially, `getattr` permission to a file allows a caller to view (not change) attributes such as date, time, and discretionary access control (DAC) access modes. In a standard Linux system, a caller may view such information on a file with only search permission to the file's directory even if the caller does not have read access to the file. Assuming that `user_t` is the domain type of an ordinary, unprivileged user process such as a login shell process, and `bin_t` is the type associated with executable files that users run with the typical security privileges (for example, `/bin/bash`), the rule might be in a policy to allow users to execute shell programs such as the bash shell.

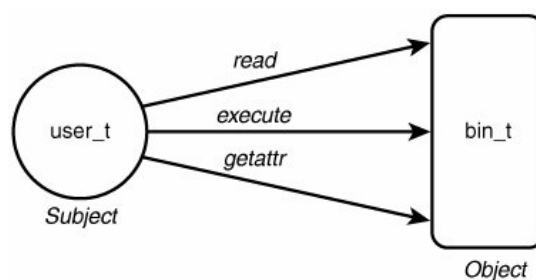


Figure 2.4: Depiction of Allow Rule - Allow Subject to access object(s)

SELinux allow rules such as the preceding example are really all there is to granting access in SELinux. The challenge is determining the many thousands of accesses one must create to permit the system to work while ensuring that only the necessary permissions are granted, to make it as secure as possible. To further explore type

enforcement, let's use the example of the password management program (that is, `passwd`). In Linux, the password program is trusted to read and modify the shadow password file (`/etc/shadow`) where encrypted passwords are stored. The password program implements its own internal security policy that allows ordinary users to change only their own password while allowing root to change any password. To perform this trusted job, the password program needs the ability to move and re-create the shadow file. In standard Linux, it has this privilege because the password program executable file has the `setuid` bit set so that when it is executed by anyone, it runs as root user (which has all access to all files). However, many, many programs can run as root (in reality, all programs can potentially run as root). This means, any program (when running as root) has the potential to modify the shadow password file. What type enforcement enables us to do is to ensure that only the password program (or similar trusted programs) can access the shadow file, regardless of the user running the program. Figure 2.5 depicts how the password program might work in an SELinux system using type enforcement.

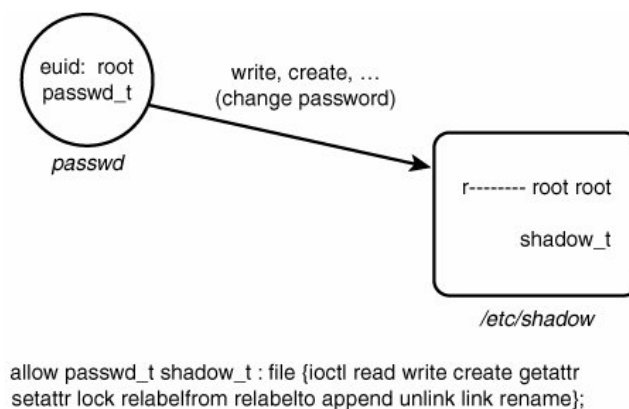


Figure 2.5: Type Enforcement Example - Type Based Access of object(s)

In this example, we defined two types. The `passwd_t` type is a domain type intended for use by the password program. The `shadow_t` type is the type for the shadow password file. If we examine such a file on disk, we would see something like this:

```
# ls -Z /etc/shadow
```

```
- r - - - - - root root system_u:object_r:shadow_t shadow
```

Likewise, examining a process running the password program under this policy would yield this:

```
# ps -aZ
```

```
joe:user_r:passwd_t 16532 pts/0 00:00:00 passwd
```

For now, we can ignore the user and role elements of the security context and just note the types. Examine the allow rule in Figure 2.5 The purpose of this rule is to give the passwd process' domain type (passwd_t) the access to the shadow's file type (shadow_t) needed to allow the process to move and create a new shadow password file. So, in reexamining Figure 2.5, we see that the depicted process running the password program (passwd) can successfully manage the shadow password file because it has an effective user ID of root (standard Linux access control) and because a TE allow rule permits it adequate access to the shadow password file's type (SELinux access control). Both are necessary, neither is sufficient.

2.4.1 The Problem of Domaini Transition

If all we had to do was provide allowed access for processes to objects such as files, writing a TE policy would be straightforward. However, we have to figure out a way to securely run the right programs in a process with the right domain type. For example, we do not want programs not trusted to access the shadow file to somehow execute in a process with the passwd_t domain type. This could be disastrous. This problem brings us to the issue of domain transitions.

To illustrate, examine Figure 2.6, in which we expand upon the previous password program example. In a typical system, a user (say Joe) logs in, and through the magic of the login process, a shell process is created (for example, running bash). In standard Linux security, the real and effective user IDs (that is, joe) are the same. In our

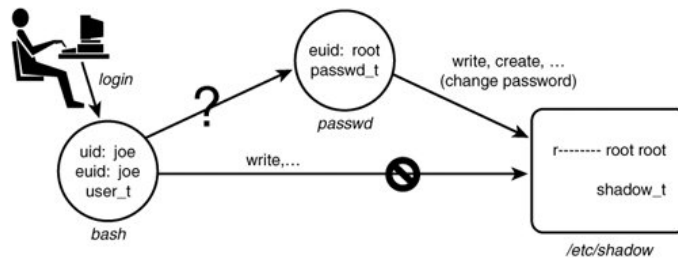


Figure 2.6: Problem of Domain Transition - Securely run right programs

example SELinux policy, we see that the process type is `user_t`, which is intended to be the domain type of ordinary, untrusted user processes. As Joe's shell runs other programs, the type of the new processes created on Joe's behalf will keep the `user_t` domain type unless some other action is taken. So how does Joe change passwords?

We would not want Joe's untrusted domain type `user_t` to have the capability to read and write the shadow password file directly because this would allow any program (including Joe's shell) to see and change the contents of this critical file. As discussed previously, we want only the password program to have this access, and then only when running with the `passwd_t` domain type. So, the question is how to provide a safe, secure, and unobtrusive method for transitioning from Joe's shell running with the `user_t` type to a process running the password program with the `passwd_t` type.

2.4.2 Review of SetUID Programs in Standard Linux Security

Before we discuss how to deal with the problem of domain transitions, let's first review how a similar problem is handled in standard Linux where the same problem of providing Joe a means to securely change his password exists. The way Linux solves this problem is by making `passwd` a setuid to the root program. If you list the password program file on a typical Linux system, you see something like this:

```
# ls -l /usr/bin/passwd

-r - s - - x - - x 1 root root 19336 Sep 7 04:11 /usr/bin/passwd
```

Notice two things about this listing. First the s in the x spot for the owner permission. This is the so-called setuid bit and means that for any process that executes this file, its effective UID (that is, the user ID used for access control decisions) will be changed to that of the file owner. In this case, root is the file owner, and therefore when executed the password program will always run with the effective user ID of root. Figure 2.7 shows these steps.

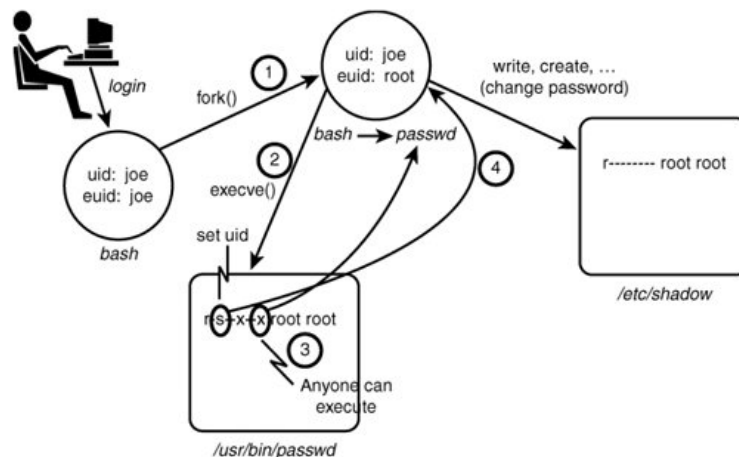


Figure 2.7: Password Program Security in Standard Linux - setuid

What actually happens when Joe runs the password program is that his shell will make a `fork()` system call to create a near duplicate of itself. This duplicate process still has the same real and effective user IDs (joe) and is still running the shell program (bash). However, immediately after forking, the new process will make an `execve()` system call to execute the password program. Standard Linux security requires that the calling user ID (still joe) have x access, which in this case is true because of the x access to everyone. Two key things happen as a result of the successful `execve()` call. First, the shell program running in the new process is replaced by the password program (passwd). Second, because the setuid bit is set for owner, the effective user ID is changed from the process' original ID to the file owner ID (root in this case). Because root can access all files, the password program can now access the shadow password file and handle the request from Joe to change his password.

Use of the `setuid` bit is well established in UNIX-like operating systems and is a simple and powerful feature. However, it also illustrates the primary weakness of standard Linux security. The password program needs to run as root to access the shadow file. However, when running as root, the password program can effectively access any system resource. This is a violation of the central security engineering principal of least privilege. As a result, we must trust the password program to be benign with respect to all other possible actions on the system. For truly secure applications, the password program requires an extensive code audit to ensure it does not abuse its extra privilege. Further, when the inevitable unforeseen error makes its way into the password program, it presents a possible opportunity to introduce vulnerabilities beyond accessing the shadow password file. Although the password program is fairly simple and highly trusted, think of the other programs (including login shells) that may and do run as root with that power.

What we would really like is a way to ensure least privilege for the password program and any other program that must have some privilege. In simple terms, we want the password program to be able to access only the shadow and other password-related files plus those bare-minimum system resources necessary to run; and we would like to ensure that no other program but the password (and similar) programs can access the shadow password file. In this way, we need only evaluate the password (and similar) programs with respect to its role in managing user accounts and need not concern ourselves with other programs when evaluating security concerns for user account management.

This is where type enforcement comes in.

2.4.3 Domain Transition

As previously shown in Figure 2.5, the allow rule that would ensure that `passwd` process domain type (`passwd.t`) can access the shadow password file. However, we still

have the problem of domain transitions described earlier. Providing for secure domain transition is analogous to the concept of setuid programs, but with the strength of type enforcement. To illustrate, let's take the setuid example and add type enforcement (see Figure 2.8).

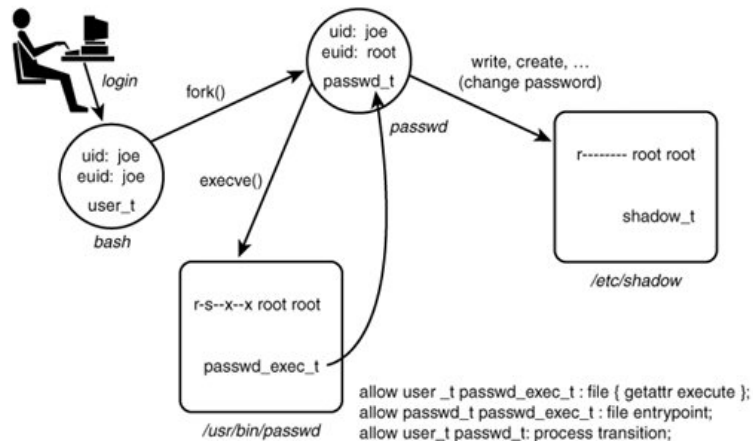


Figure 2.8: Passwd Program Security in SELinux - domain transitions

Now our example is more complicated. Let's examine this figure in detail. First notice that we have added the three types we showed previously, namely Joe's shell domain (user_t), the password program's domain type (passwd_t), and the shadow password file type (shadow_t). In addition, we have added the file type for the passwd executable file (passwd_exec_t). For example, listing the security context for the password program on-disk executable would yield a result something like this:

```
# ls -Z /usr/bin/passwd

-r - s - - x - - x root root system_u:object_r:passwd_exec_t /usr/bin/passwd
```

Now we have enough information to create the TE policy rules that allow the password program (and presumably only the password program) to run with the passwd_t domain type. Let's look at the rules from Figure 2.8. The first rule is as follows:

```
allow user_t passwd_exec_t : file { getattr execute };
```

What this rule does is allow Joe's shell (user_t) to initiate an execve() system call

on the `passwd` executable file (`passwd_exec_t`). The SELinux execute file permission is essentially the same permission as `x` access for files in standard Linux. (The shell “stats” the file before trying to execute, hence the need for `getattr` permission, too.) Recall our description of how a shell program actually works. First it forks a copy of itself, including identical security attributes. This copy still retains Joe’s shell original domain type (`user_t`). Therefore, the execute permission must be for the original domain (that is, the shell’s domain type). That is why `user_t` is the source type for this rule.

Let’s now look at the next allow rules from Figure 2.8:

```
allow passwd_t passwd_exec_t : file entrypoint;
```

This rule provides entrypoint access to the `passwd_t` domain. The entrypoint permission is a rather valuable permission in SELinux. What this permission does is define which executable files (and therefore which programs) may “enter” a domain. For a domain transition, the new or “to-be-entered” domain (in this case, `passwd_t`) must have entrypoint access to the executable file used to transition to the new domain type. In this case, assuming that only the `passwd` executable file is labeled with `passwd_exec_t`, and that only type `passwd_t` has entrypoint permission to `passwd_exec_t`, we have the situation that only the password program can run in the `passwd_t` domain type. This is a powerful security control.

Let’s now look at the final rule:

```
allow user_t passwd_t : process transition;
```

This is the first allow rule we have seen that did not provide access to file objects. In this case, the object class is `process`, meaning the object class representing processes. Recall that all system resources are encapsulated in an object class. This concept holds for processes, too. In this final rule, the permission is `TTransition` access. This permission is needed to allow the type of a process’ security context to change. The original type (`user_t`) must have `TTransition` permission to the new type (`passwd_t`) for the domain transition to be allowed.

These three rules together provide the necessary access for a domain transition to occur. For a domain transition to succeed, all three rules are necessary; alone, none is sufficient. Therefore, a domain transition is allowed only when the following three conditions are true:

1. The process' new domain type has enTRypoint access to an executable file type.
2. The process' current (or old) domain type has execute access to the entry point file type.
3. The process' current domain type has transition access to the new domain type.

When all three of these permissions are permitted in a TE policy, a domain transition may occur. Further, with the use of the entryptpoint permission on executable files, we have the power to strictly control which programs can run with a given domain type. The `execve()` system call is the only way to change a domain type, giving the policy writer great control

Now the issue is how does Joe indicate that he wants a domain transition to occur. The above rules allow only the domain transition; they do not require it. There are ways that a programmer or user can explicitly request a domain transition (if allowed), but in general we do not want users to have to make these requests explicitly. All Joe wants to do is run the password program, and he expects the system to ensure that he can. We need a way to have the system initiate a domain transition by default.

2.4.4 Default Domain Transitions: type transition Statement

To support domain transitions occurring by default (as we want in the case of the password program), we need to introduce a new rule, the type transition rule (`type_transition`). This rule provides a means for the SELinux policy to specify default transitions that

should be attempted if an explicit transition was not requested. Let's add the following type transition rule to the allow rules:

```
type_transition user_t passwd_exec_t : process passwd_t;
```

The syntax of this rule differs from the allow rule. There are still source and target types (`user_t` and `passwd_exec_t`, respectively) and an object class (`process`). However, instead of permissions, we have a third type, the default type (`passwd_t`).

Type transition rules are used for multiple different purposes relating to default type changes. For now, we are concerned with a type transition rule that has `process` as its object class. Such rules cause a default domain transition to be attempted. The type transition rule indicates that, by default on an `execve()` system call, if the calling process' domain type is `user_t` and the executable file's type is `passwd_exec_t` (as is the case in our example in Figure 2.8), a domain transition to a new domain type (`passwd_t`) will be attempted.

The type transition rule allows the policy writer to cause default domain transitions to be initiated without explicit user input. This makes type enforcement less obtrusive to the user. In our example, Joe does not want to know anything about access control or types; he wants only to change his password. The system and policy designer can use type transition rules to make these transitions transparent to the user.

2.5 The Role of Roles

SELinux also provides a form of role-based access control (RBAC)(19). The RBAC feature of SELinux is built upon type enforcement; access control in SELinux is primarily via type enforcement. Roles limit the types to which a process may transition based on the role identifier in the process' security context. In this manner, a policy writer can create a role that is allowed to transition into a set of domain types (assuming the type enforcement rules allow the transition), thereby defining the limits of the role.

Take our password program example in Figure 2.8. Although according to the type enforcement rules, the password program can be executed by the `user_t` domain type to enter the new `passwd_t` domain, Joe's role must also be allowed to be associated with the new domain type for the transition to occur. To illustrate, we extend the password program example in Figure 2.9.

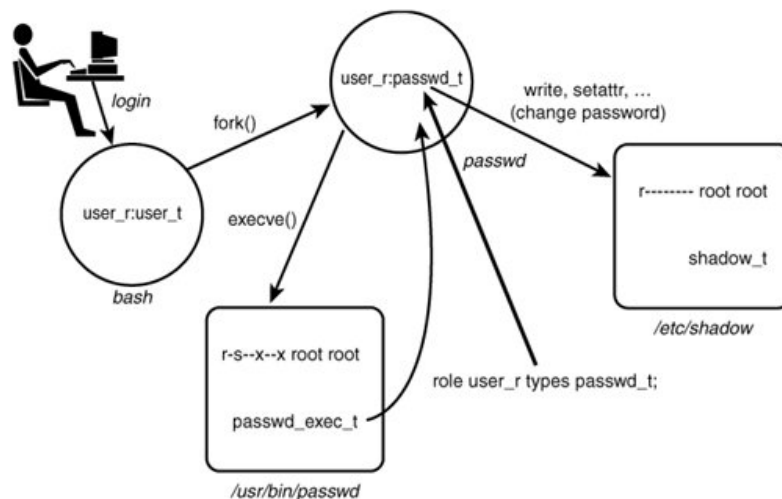


Figure 2.9: Roles in Domain Transition - Limit the types to which a process may transition

We have added the role portion (`user_r`) of the security contexts for the processes depicted. We also added a new rule, specifically the role statement:

```
role user_r type passwd_t;
```

The role statement declares role identifiers and associates types with the declared role. The previous statement declares the role `user_r` (if it has not already been declared in the policy) and associates the type `passwd_t` with the role. What this association means is that the `passwd_t` type is allowed to coexist in a security context with the role `user_r`. Without this role statement, the new context `joe:user_r:passwd_t` could not be created, and the `execve()` system call would fail, even though the TE policy allows Joe's type (`user_t`) all the necessary access.

A policy writer can define roles that are further constrained and then associate

these roles to specific users. For example, imagine that in our policy we also create a role called `restricted_user_r`, identical to `user_r` in all regards except that it is not associated with the `passwd_t` type. Thus, if Joe's role is `restricted_user_r` instead of `user_r`, Joe would not be authorized to run the password program even though the TE rules would allow his domain type the access.

2.6 What is Policy

Policy is the set of rules that guide the SELinux security engine. It defines types for file objects and domains for processes, uses roles to limit the domains that can be entered, and has user identities to specify the roles that can be attained. A domain is what a type is called when it is applied to a process. A type is a way of grouping together like items based on their fundamental security sameness. This doesn't necessarily have to do with the unique purpose of an application or the content of a document. For example, an object such as a file can have any type of content and be for any purpose, but if it belongs to a user and lives in that user's home directory, it is considered to be of a specific security type, `user_home_t`. These object types gain their sameness because they are accessible in the same way by the same set of subjects. Similarly, processes tend to be of the same type if they have the same permissions as other subjects.

In the targeted policy, programs that run in the `unconfined_t` domain have an executable with a type such as `sbin_t`. From an SELinux perspective, that means they are all equivalent in terms of what they can and cannot do on the system. For example, the binary executable file object at `/usr/bin/postgres` has the type of `postgresql_exec_t`. All of the targeted daemons have their own `*_exec_t` type for their executable applications. In fact, the entire set of PostgreSQL executables such as `createlang`, `pg_dump`, and `pg_restore` have the same type, `postgresql_exec_t`, and they transition to the same domain, `postgresql_t`, upon execution. The policy defines various rules that say how each domain may access each type. Only what is specifically allowed by the rules is

permitted. By default every operation is denied and audited, meaning it is logged in `$AUDIT_LOG`, such as `/var/log/messages`. Policy is compiled into binary format for loading into the kernel security server, and as the security server hands out decisions, these are cached in the AVC for performance. Policy can be administratively defined, either by modifying the existing files or adding local TE and file context files to the policy tree. Such a new policy can be loaded into the kernel in real time. Otherwise, the policy is loaded during boot by `init`, as explained in Next Section “Policy Role in Boot”. Ultimately, every system operation is determined by the policy and the type labeling of the files.

SELinux is an implementation of domain-type access control, with role-based limiting. The policy specifies the rules in that environment. It is written in a simple language created specifically for writing security policy. Policy writers use m4 macros to capture common sets of low-level rules. There are a number of m4 macros defined in the existing policy, which assist greatly in writing new policy. These rules are pre-processed into many additional rules as part of building `policy.conf`, which is compiled into the binary policy. The files are divided into various categories in a policy tree at `$SELINUX_SRC/`. Access rights are divided differently among domains, and no domain is required to act as a master for all other domains. Entering and switching domains is controlled by the policy, through login programs, userspace programs such as `newrole`, or by requiring a new process execution in the new domain, called a transition.

2.6.1 Policy Role in Boot

SELinux plays an important role early in system start-up. Since all of the processes must be labeled with their proper domain, `init` does some essential actions early in the boot process that keep labeling and policy enforcement in sync.

1. After the kernel has been loaded during boot, the initial process is assigned the

predefined initial SID kernel. Initial SIDs are used for bootstrapping before the policy is loaded.

2. `/sbin/init` mounts `/proc/`, then looks for the `selinuxfs` file system type. If it is present, that means SELinux is enabled in the kernel.
3. If `init` does not find SELinux in the kernel, finds it is disabled via the `selinux=0` boot parameter, or if `/etc/selinux/config` specifies that `SELINUX=disabled`, boot proceeds with a non-SELinux system. At the same time, `init` sets the enforcing status if it is different from the setting in `/etc/selinux/config`. This happens when a parameter is passed during boot. The default mode is permissive until the policy is loaded, then enforcement is set by the configuration file or by the parameters `enforcing=0` or `enforcing=1`.
4. If SELinux is present, `/selinux/` is mounted.
5. The kernel checks `/selinux/policyvers` for the supported policy version. `init` looks into `/etc/selinux/config` to see which policy is active, such as the targeted policy, and loads the associated file at `$SELINUX_POLICY/policy` version. If the binary policy is not the version supported by the kernel, `init` attempts to load the policy file if it is a previous version. This provides backward compatibility with older policy versions. If the local settings in `/etc/selinux/targeted/booleans` are different from those compiled in the policy, `init` modifies the policy in memory based on the local settings prior to loading the policy into the kernel.
6. Now that the policy is loaded, the initial SIDs are mapped to security contexts in the policy, as defined in `$SELINUX_SRC/initial_sid_contexts`. In the case of the targeted policy, the new domain is `user_u:system_r:unconfined_t`. The kernel can now begin to get security contexts dynamically from the in-kernel security server.

2.7 File System Security Contexts

SELinux stores file security labels in xattrs . Xattrs are stored as name-value property pairs associated with files. SELinux uses the security.selinux attribute. The xattrs can be stored with files on a disk or in memory with pseudo file systems. Currently, most file system types support the API for xattr, which allows for retrieving attribute information with getxattr. Some non-persistent objects can be controlled through the API. The pseudo-tty system controlled through /dev/pts is manipulated through setxattr, enabling programs such as sshd to change the context of a tty device.

There are two approaches to take for storing file security labels on a file system, such as ext2 or ext3. One approach is to label every file system object (all files) with an individual security attribute 2 . Once these labels are on the file system, the xattrs become authoritative for holding the state of security labels on the system.

The other option is to label the entire file system with a single security attribute. This is called genfs labeling. One example of this is with ISO9660 file systems, which are used for CD-ROMs and .iso files.

Chapter 3

SELinux Policy

3.1 Loadable Policy modules

A new method for creating a modular policy is called loadable modules, which uses recent extensions to `checkpolicy` and a module compiler (`checkmodule`) to construct loadable policy modules compiled independently of each other. In the loadable module case, there is no longer a monolithic binary policy constructed; instead, a (expectedly smaller) core subset of the policy is constructed called the base module. You create the base module much like you create the monolithic policy. With loadable modules, however, you can streamline the base module, including only rules relating to the core operating system. The rest of the policy is created as separate loadable modules. You can add all other policy rules in a modular fashion when you install their associated software package.

Loadable modules introduce policy syntax changes that are designed to ease the division of the policy into separate, individually distributable policy modules. These changes differ for base and nonbase modules(2). The base module uses the same policy language as monolithic policies with minor additions. Nonbase (that is, loadable) modules use a subset of the standard policy language with several additional language

features. The subset of the policy language includes most of the type enforcement, role, and user statements. The additional language features are used to manage dependencies between modules.

3.1.1 Building and Installing Monolithic Policies

Starting from the left side of this figure, you have the source files for the policy broken down into many tens of individual source modules. We will talk about various conventions for organizing these modules in the example policy. For now, just understand that these files are combined through a combination of scripts and macro processors into the single `policy.conf` file, which is a complete and syntactically correct statement of a SELinux source policy. You then compile the source policy using `checkpolicy` into a binary policy file (assuming no errors!) appropriate for the kernel. The `load_policy` program is then used to load the binary policy file into the kernel, which then enforces access control based on the policy rules.

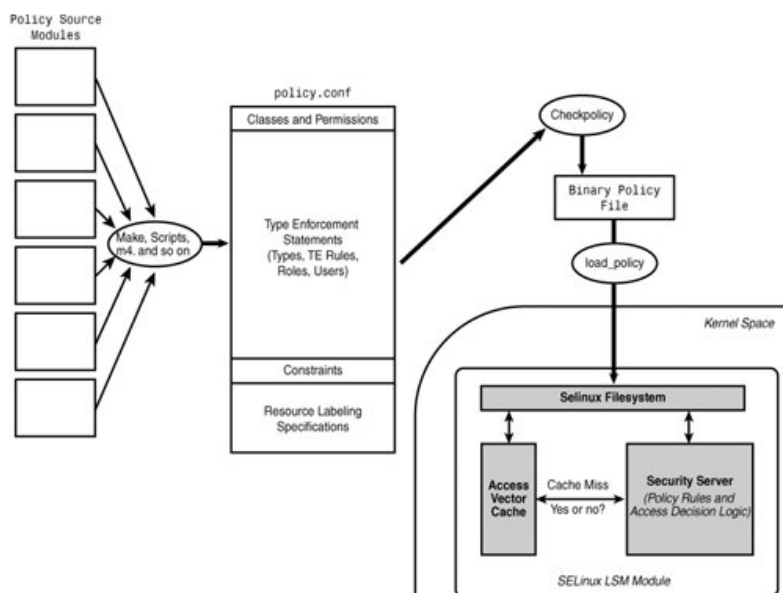


Figure 3.1: Building Policy - Creation of Policy from source

3.2 What is the Targeted Policy?

The SELinux policy is highly configurable. For Red Hat Enterprise Linux, Red Hat supports a single policy, the targeted policy. Under the targeted policy, every subject and object runs in the `unconfined_t` domain except for the specific targeted daemons. The objects on the system that are in the `unconfined_t` domain are allowed by SELinux to have no restrictions and fall back to using standard Linux security, that is, DAC. This policy is flexible enough to fit into enterprise infrastructures. The daemons that are part of the targeted policy run in their own domains and are restricted in every operation they perform on the system. This way daemons that are broken or exploited are limited in the damage they can do. The opposite of the targeted policy is the strict policy. This does not ship with Red Hat Enterprise Linux. In the strict policy, every subject and object are in a specific security domain, with all interactions and transitions individually considered within the policy rules(2). This is a much more complex environment. This guide focuses on the targeted policy that comes with Red Hat Enterprise Linux, and the components of SELinux used by the targeted daemons. The targeted daemons are:

- `dhcpd`
- `httpd`
- `mysqld`
- `named`
- `nsd`
- `ntpd`
- `portmap`
- `postgres`

- snmpd
- squid
- syslogd
- winbind

3.3 Reference Policy

The reference policy(6) is a newer method for building SELinux policies with the goal of making the policy easier to understand, modify, maintain, and validate. These goals are largely achieved through greater application of modern software engineering principles, such as modularity and encapsulation. The reference policy also allows strict and targeted policy variants to be built from the same source tree and incorporates support for emerging SELinux technologies, such as loadable modules.

The reference policy project is an effort to reengineer the existing policies derived from the National Security Agency (NSA) example policy into an easier to use, understand, and maintain policy. The primary goals are to create a strong design philosophy in policy development by applying well-understood software design principles, while retaining the years of experience learned by community effort in developing the existing policies. In other words, keep the good and fix the bad.

Chief among the “bad” with the existing example policy is its lack of strong modularity and the tight coupling of the policy source modules that results. Although macros add abstraction to the example policy, all policy identifiers (types, roles, attributes, and so on) are, in reality, global. Editing one policy module might require knowledge of many others and interdependency among modules is pervasive and poorly documented. Likewise, creating a new policy module requires detailed understanding of the implementation details of other policy modules.

Some of the key characteristics of the reference policy that make policy development easier and more understandable are as follows:

- A single source tree that supports (without destructive modification) strict and targeted policies, optional multilevel security/multicategory security (MLS/MCS) extensions, a single kernel policy file (called a monolithic policy), and the new loadable module infrastructure.
- Application of strong design principles, chiefly in the area of loosely coupled modules, with well-defined interfaces and no global use of type and other identifiers. (So, for example, all changes relating to a type are made entirely within a single module.)
- Integrated documentation support, capturing descriptions of module interfaces so that, for example, a policy module developer can use an interface without having to understand how the interface is implemented in the module.
- Simplify and standardize policy configuration and build options, so in general policy module writing and customization is easier and requires less expertise.

Besides making policy development easier, the reference policy also intends to make verifying the security properties of a policy easier to achieve (for example, for security certifications) and to increase support for high-level developments tools, such as graphical integrated development environments and sophisticated policy debuggers.

3.3.1 Overview of Policy Source File Structure

The file structure for the reference policy⁵ differs from the example policy. Before we describe the key implementation details of the reference policy, let's overview the layout of the reference policy source files to familiarize ourselves with its file structure.

3.3.1.1 Build and Support Files

The following files and directories are used for building or otherwise supporting the building of a reference policy:

build.conf This file defines the set of build options that we can change and set to control the build process. This file is included within the Makefile during the make process.

Rules.modular This file contains the make rules for building a policy that supports loadable modules. It supports building both the base policy module and loadable policy modules. Which modules are built as part of the base module, and which are built as loadable modules, is defined in `policy/modules.conf` (see below). The build option `MONOLITHIC` in `build.conf` controls whether a modular or monolithic policy is built.

Rules.monolithic If a monolithic policy is being built, this file (rather than `Rules.modular`) is included in the Makefile to define the rules for building a monolithic policy.

config/ This directory contains subdirectories for the application configuration files for every variety of policy that can be built with the reference policy. These configuration files are exactly the same as the files in the `appconfig/` directory for the example policy. These are files installed in the operational policy directory (for example, `/etc/selinux/refpolicy`) to support various services and

doc/ This directory contains files that support integrated documentation generation that is part of the reference policy. To see the resulting documentation generation, view the reference policy Web site (<http://serefpolicy.sourceforge.net/>) or run the command `make html` and look in the `doc/html/` directory.

support/ This directory contains source code and scripts for the tools used to support the build process.

3.3.1.2 Core Policy Files

In the reference policy, the primary files(2) used to create a policy (or loadable modules) are contained in the `policy/` directory. These are the files that we, as policy writers, will most commonly modify and examine:

policy/constraints This file is where all non-MLS constraints are defined. It is essentially identical to the same file in the example policy.

policy/flask/ This directory contains the Flask definitions identical with the example policy. “Original Example Policy,” for this directory and its files.

policy/mls and policy/mcs These two files define two configurations for the optional MLS features in SELinux. They are identical in intent to the same files in the example policy;

policy/global.booleans and policy/global.tunables These two files currently store defined Booleans and their default values. They are combined and installed in `/etc/selinux/refpolicy/booleans` and enable an administrator to change the default values of Booleans. The reason for two files is one of a philosophy that may eventually lead to a difference in implementation. The `global.booleans` file contains Booleans intended to support truly conditional policies that an administrator may want to toggle on and off in a production system. The

global.tunables contains Booleans that are build/runtime configuration options that are likely changed once during installation and never changed again. Some of these latter Booleans (that is, the tunables) may be implemented using features of loadable modules in the future.

policy/modules.conf This file configures which modules are to be included in a build process and in what form. A module can be built in to a monolithic policy or the base module for a loadable policy, built as a loadable module, or not built at all. The `modules.conf` file is created with the `make conf` command.

policy/modules/ This directory contains all the policy modules divided into subdirectories by layer. Most of the files that we would examine, edit, and change will be in this directory.

policy/support/ This directory contains macros used throughout the policy modules to aide in policy writing. For example, the file `policy/support/obj_perm_sets.spt` defines macros that define sets of permissions. We use these macros to simplify some of the policy writing steps and to create easier to read policy.

policy/users This file is the same as the `users` file in the example policy though it uses an interface (that is, a macro), `gen_user()`, to create the user statements;

3.3.2 Design Principles

3.3.2.1 Layering

In general, the reference policy tries to keep dependencies between modules within a layer or to a layer "below" the module's layer(2). We can find the layer directories, which contain the modules for each layer, in `policy/modules/`. The reference policy currently defines the following layers:

- **Kernel** This layer contains policy modules that directly relate to the Linux kernel. This is the lowest layer of modules. Modules at this layer include policy statements for the kernel, devices, filesystems, and basic networking. Most of these modules will always be included in any type of policy.
- **System** These are policy modules that are also usually included in a policy but do not directly support the kernel. Modules at this layer include policy for common libraries, login processes, and network management.
- **Services** This layer contains policy modules for all services and daemons not part of the system layer. These modules range from `cron`, to `sshd`, to `apache`.

- Admin This layer contains policy modules for administrative tools and commands that have their own domain type.
- Apps This layer contains policy modules for all other programs that have their own domain type and policy module.

3.3.2.2 Modularity

Modularity is the strongest design principle of the reference policy. In the reference policy, modules are required to be loosely coupled. This loose coupling is achieved through the enforcement of two strong design conventions: encapsulation and abstraction.

3.3.2.3 Encapsulation

Encapsulation is a reference policy modularity convention that requires that type and attribute names may only be used within a single module. In effect, type and attribute names may not be used as global names. Only the module that defines the type/attribute may reference the name directly. Any other module that would require the use of the type/attribute must do so through well-defined interfaces that the owning module defines.

For example, in the example policy, all types that are domain types are given the domain attribute. Every policy module simply has this knowledge built in and explicitly adds domain to the list of attributes for all domain types they define. If we decided to change how the concept of a domain was implemented in the policy (say by granting each type explicit rules or even by renaming the attribute), we would have to change every module that defines a domain type.

In the reference policy, a module called “domain” in the kernel layer defines the concept of a domain. It just so happens that this concept is implemented using the domain attribute as with the example policy. However, this implementation detail is

private to the domain module and could be changed (for example, renamed) without impacting any other module source. Any module that wants to make one of its types a domain type would call an interface defined in the domain module:

```
domain_type(my_type) # interface to make a type a domain type
```

Encapsulation enables us to make the reference policy modules' implementation details private to the module resulting in loosely coupled modules.

3.3.2.4 Abstraction

Abstraction is a design goal where interfaces describe what abstract access they provide and not how they do it(2). The intent of reference policy interfaces is to describe what abstract access is given or system capability is being enabled with the interface. The policy statements required to enable that access should not be a concern of the interface caller. For example, the macro we discussed previously to make a type a domain type is called `domain_type()` and not `add_domain_attribute()`. The intent of the interface is to make a type a domain type; doing this by adding the domain attribute is just the private implementation detail of the `domain_type()` interface. This interface could have instead simply added explicit rules for each individual type provided with the interface, and we can still change that implementation if we choose without impacting other modules that use this interface.

As another example, to allow a directory to be used as a mount point we would call the `file_mountpoint()` macro in the “files” module. We do not need to know that the implementation of this interface applies the attribute mountpoint to all directory types called with this interface and then defines rules for the attribute to allow the type to be used as a mount point. As a policy writer, all we need to know is that the `file_mountpoint()` interface is how we allow a directory type to be a mount point.

Currently, the reference policy has a low-level of interfaces implemented within each module. Eventually higher-level abstractions will be developed through additional

interfaces that combine the lower-level interfaces.

3.3.2.5 Module Files

As discussed earlier, within the reference policy source tree all modules are kept in `policy/modules/[layer]/` where the layer is a directory whose name coincides with one of the layers discussed previously. Each module must consist of three related files, all of which have the same root name (that is, module name):

- Private policy file (`.te`) This file contains the module private declarations and rules. In general, all module type and attribute declarations are contained in the `.te` file and the rules that give these types and attributes their core access.
- External interface file (`.if`) This file contains the module interfaces. These interfaces are the means by which other modules access the types and attributes of this module.
- Labeling policy file (`.fc`) This file contains the file context labeling statements relating to this module

Because a strong requirement is that no type or attribute be global, only the `.te` and `.if` file for a given module may use the module's type/attribute names explicitly. All other references to a module's types and attributes must be via the module's interfaces.

3.3.2.6 Interfaces

As discussed previously, one of the most significant improvements implemented in the reference policy⁵ is the use of interface macros for gaining access to a type outside of the module in which the type is defined. Interfaces provide access to a module's policy resources (for example, to its privately declared types and attributes). All other modules needing a particular access use the same interface; therefore, the policy rules

required for the access will be consistent across all users of the interface. Therefore, policy changes for access to a type require only a change in one place, rather than requiring changes to all the modules that use the type as is common in the example policy.

As noted above, interfaces are kept in a module's .if file and are implemented as macros. Currently, reference policy supports two kinds of interfaces: access interfaces and template interfaces.

The name we give each interface follows the convention of `modname_purpose`. So, for example, we can tell that the `domain_type()` interface is defined in the “domain” module and its purpose is to make a provided type a domain type. (We avoid the verbose name such as `domain_domain_type()` when the module name is also part of the purpose.)

Chapter 4

Proposed System

4.1 Expression of multi-system policy

We determined that a single policy for the entire distributed system is necessary, rather than having individual policies for each system. Further issues arise when determining the best way to express and develop a policy in this manner.

4.1.1 Preserve equivalence where possible

Although types must be different between systems in some cases, the policy should not introduce unnecessary differences. Types are security equivalence classes; if two subjects or objects have the same security properties on different systems, they should use the same type. For this reason, it is not advisable to adopt a policy-wide mechanism for separating all types on a per-system basis. One should not require, for example, that the shadow file on system 1 has the type `system1_shadow_t` and the shadow file on system 2 has the type `system2_shadow_t`. If these shadow files have the same security properties, they should use a single type. Requiring per-system types would result in a policy far bigger than necessary and much more difficult to manage inter-system access on a large scale.

Instead, if two or more systems have very similar duties, e.g., load balancing mirrors or redundant routers, it is likely that they will have very similar or even identical policies. This concept can, and should, be applied on a per-application basis. As mentioned earlier, if two Apache instances need different types but the DNS servers on the same systems do not, the DNS servers should share the same type while distinct types should separate the Apache instances.

4.1.2 Unified namespaces

Each part of an SELinux policy types, roles, users, etc. has a namespace that ensures uniqueness. When a single coherent policy is used for a network of systems, these namespaces must be unique across the entire network. Any type in the policy has the same policy associated with it no matter what system is enforcing access.

This follows the type enforcement methodology of using types as equivalence classes and is conducive to policy analysis. It should also make the policies and systems generally more legible and comprehensible. A policy analysis should not have to consider whether or not a subject type, for example `httpd.t`, on one system has the same security properties as that subject type on another system.

4.1.3 The need for policy templating

The currently available policies do not provide a means to differentiate instances of applications other than by reproducing the entire policy for that application. Instead, this should be done via the infrastructure rather than forcing users to develop policy. One possible way to do this is to extend the policy module concept to include module templates. These would not be standard loadable modules but would include everything necessary to create an entirely new set of subject and object types with associated policies for a specific application. For example, to make a new Apache policy that is separate from the standard Apache policy, simply instantiate the `httpd` policy with the

httpd.internal prefix. This would produce policy with httpd.internal_t as the subject type for the new Apache instance.

In addition to templating the policy, there must be a facility for adding and removing extra access or the templating is of limited use. It is unlikely that infrastructure alone can handle everything necessary to make this useful. The ability to load a module created from a template and make minor modifications would instead need to be added to the SELinux administration interfaces.

Analysis of the resulting policy would work very much the way it currently does. Since there is a single canonical version of the policy applied to the entire network, an analyst can use it to study information flow throughout the entire network instead of only a single system.

4.1.4 Partitioning policy

The SELinux policy is surprisingly easy to partition into chunks. The TE policy, which is the vast majority of the SELinux policy, is keyed on the source type, the target type and the object class. With the Flask architecture, each object manager enforces access on its objects. These objects in turn are part of some object class, for example file, shm (shared memory) or even window with X Windows as an object manager. Since object managers can only enforce on their objects, they would have no need for policy relating to other object classes. In this way, policy partitions can be made according to object classes.

The object managers ask the SELinux security server for access control(34) decisions, which in turn queries the policy. The security server, therefore, only needs policy for object classes that its object managers can enforce. If, for example, X Windows were not present on a web server, the security server on that system would not need policy for the window object class.

Since the policy uses object class as part of the key, a policy can be produced for each system that has inapplicable object classes removed, without changing the intent of the policy. If an object manager is subsequently added to a system, the policy for that system would need to be reproduced with the object classes that are necessary.

This model also allows for multiple security servers on a single system. For example, a system may have a kernel, user space, and hypervisor security server. Each of these security servers would provide access decisions to the object managers for which they are responsible. In this case, the systems policy would be a conjunction of the object classes needed by each of the security servers for their object managers.

4.2 The @ operator

As previously mentioned, the @ operator is a new policy mechanism for binding types to systems. For example, a policy author could refer to the `passwd_t` type on the `system_1` system as `passwd_t@system_1`. The operator is useful for determining which portions of policy are to be distributed to a particular security server. Additionally, it is a useful mechanism for specifying policy that is dependent on the systems on which the domains are running.

```
allow httpd_t@internal_webserver internal_www_files_t:file read;
```

```
allow httpd_t@internal_webserver external_www_files_t:file read;
```

```
allow httpd_t@external_webserver external_www_files_t:file read;
```

```
allow sshd_t@system_1 sshd_key_t@system_1:file read;
```

```
allow sshd_t@system_2 sshd_key_t@system_2:file read;
```

Above example shows two example usages of the @ operator. The first example shows the @ operator being used to restrict access to certain internal files by Apache running on the external web server, while permitting access to those files by Apache

running on the internal web server. The second example utilizes the @ operator to protect the private sshd key from access by sshd processes running on other systems. Both examples show that sometimes the location of the subject or object is an important factor in an access control decision. The previous examples could be accomplished within the current infrastructure without creating @ a special operator. By replacing @ with a character that was permissible in SELinux type names, underscore for instance, this policy would be possible today. Type Enforcement is quite flexible in this way, so treating sshd_t differently on each system requires only creating types for sshd on each system. Although this is true, there are several advantages of creating a new operator to perform this functionality, as described in the following subsections.

4.2.1 Expressing multi-system policies

Although one could accomplish the policy simply creating additional types, the resultant policy drastically reduces the usefulness of types as equivalence classes. It would be preferable instead to have a mechanism that provided a way to write rules about types on a system, on a group of systems, or on any system. Consequently, the system name after the @ operator, such as `internal.webserver` and `system_1`, can be an individual system or a group of systems (e.g. `workstation`). These system groups are analogous to type attributes, as they contain a number of systems and can be used in rules in place of an individual system. If a rule does not specify a location, such as the target in the first three rules, that rule applies to that type on all systems.

As an additional convenience mechanism similar to the `self` keyword in the current language, the location can also be `localhost`. This provides a mechanism for expressing policy that should apply to all systems but only between types on the local system. For example, the `sshd_t` rules would probably need to be repeated for every system in the network. With this mechanism instead, a single rule can replace these by using `sshd_t@localhost` as the subject and `sshd_key_t@localhost` as the object.

The @ operator provides a flexible mechanism for expressing location in a distributed policy. It extends the semantics of the policy language to incorporate the concept of location into the policy, which is critical in specifying distributed policy.

4.2.2 Partitioning policy with the @ operator

In addition to providing assistance in expressing distributed policy, the @ operator aids in distributing the policy. It is necessary to partition the policy into pieces that apply to a particular system. To do this, it is necessary to determine the types that apply to that particular system. The @ operator provides a convenient mechanism for doing this each system is given the policy applicable to the security servers it serves, including the 'global' policy, which is anything without an @ operator, and the specific policy, which is everything with an @ operator that specifies the system or groups of which the system is a member. This policy should be comprehensive enough to get each system up and running, and should be sufficient for all policy enforcement over local subjects and objects. It also may be desirable for a system to request all the types applicable to a different system. If, for example, a labeled network drive is mounted, the overhead associated with requesting policy updates could be minimized if the system were able to request the policy for all objects on the remote system. This would make it unnecessary for the security server to request policy many times for individual missing pieces of the remote policy as individual files on the mounted drive are accessed.

4.3 Distributed Homogeneous services

Following functionalities are required from our service to achieve the goal of providing data to every node, without dependency of a centralized server.

- Automatic Service Discovery,

4.3 Distributed Homogeneous services

- Should be able to fetch the service data (policy) file from discovered server and Apply the data to the desired service,
- Act as a Server for other Nodes on the network,
- Contain information of all the Servers providing distributed service in a given network,
- Super-user should be allowed to define the server and override Automatic Service Discovery Mechanism decision,
- In case of a failure of a parent node, service should connect to the parent of a parent to stay connected to the distributed service

In our example, Node A is initial distributor of service data, Node B is the first client connected, while Node C and D are derived clients of a distributed service.

4.3.1 Automatic service Discovery

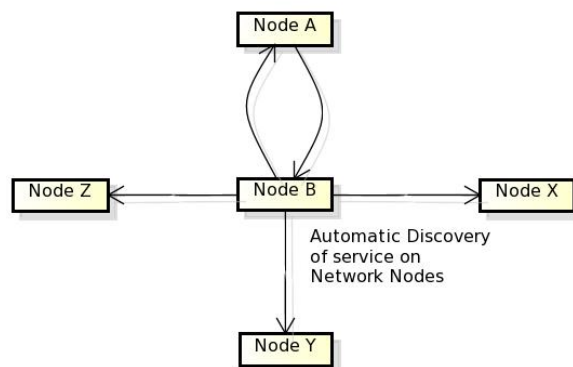


Figure 4.1: Automatic Service Discovery - Node B Automatically Discovers Node A Service

Initially Node A will be the only server in the above given scenario, while Node B will be the first client. Node B should be equipped with Automatic Service Discovery Mechanism to find out the Node A, without any configuration in Node B.

4.3.2 Gather Data from service and apply to application

Gathering the service data (policy) from Node A by Node B, also needs to apply this data to particular application. In our case, service data (policy) should be loaded into SELinux to update the application decision making functionality.

4.3.3 Homogeneous Service

In addition to creating services for distribution and gathering of service data (policy), the service should be able to act as a server for other nodes as well. For example Node A is the initial server which provides service data (policy) to Node B,

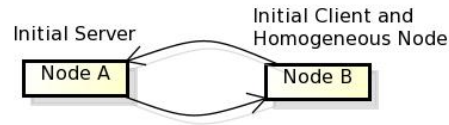


Figure 4.2: Initial Client Server - Node A servicing Node B

Node B should also be able to propagate the service data (policy) to other nodes in network to achieve actual distributed environment.

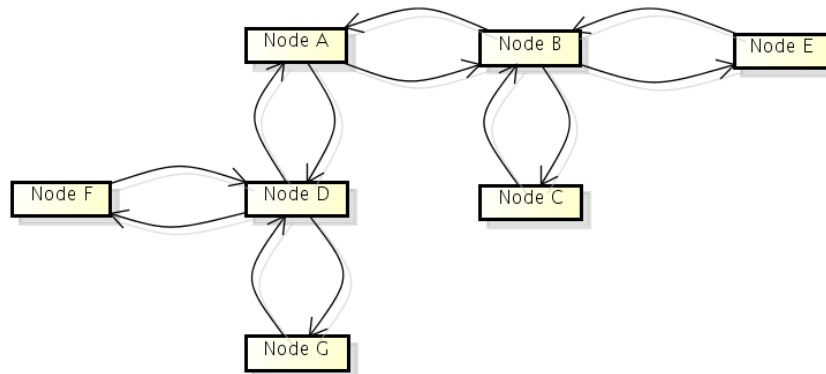


Figure 4.3: Homogeneous Service - Nodes are Distributively Providing Services

As show in above example, Node A has provided the service data (policy) component to Node B and D. While Node B also was able to provide the service data (policy) component to Node C. This is possible due to the binary policy which contains all the information required for every host on the network. In this was service data (policy) can be distributed without much load on any network node.

4.3.4 Override Atomic Service Discovery Decision

Super-user should be allowed to override the decision made by Automatic Service Discovery Mechanism and define the desired server to fetch data from or connect to.

4.3.5 Hierarchy based data distribution

Each Node running this service must have the information of all the nodes running this service in network, so in case of failure of a parent node, service can connect to parent of parent node which is available in the hierarchy. Also this information is required to propagate the updates in the service data (policy) to all nodes connected.

4.3.6 Failover

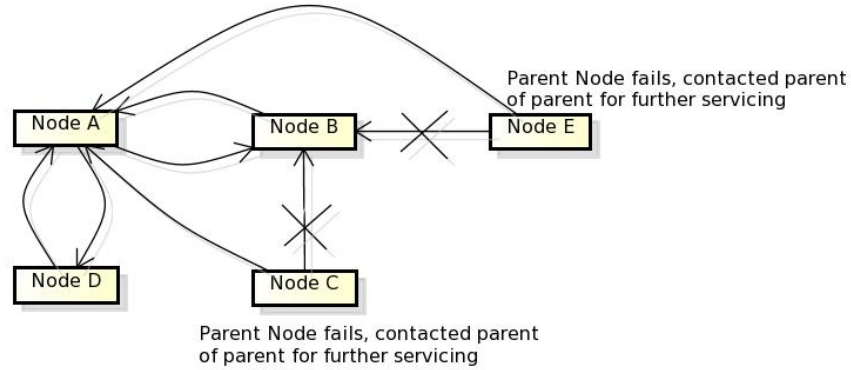


Figure 4.4: Failover Scenario - Nodes with failed parent nodes contacts parent of parent

In above example, if Node B Service fails to respond, Node C Service should contact Node A Service, to achieve failover. If even Node A service fails to respond, Node C service should contact Node D Service.

4.3.7 Activity Diagrams

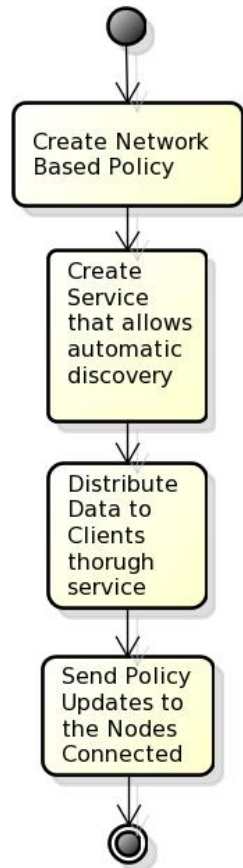


Figure 4.5: Activity Diagram of Initial Server - Creating Base Service

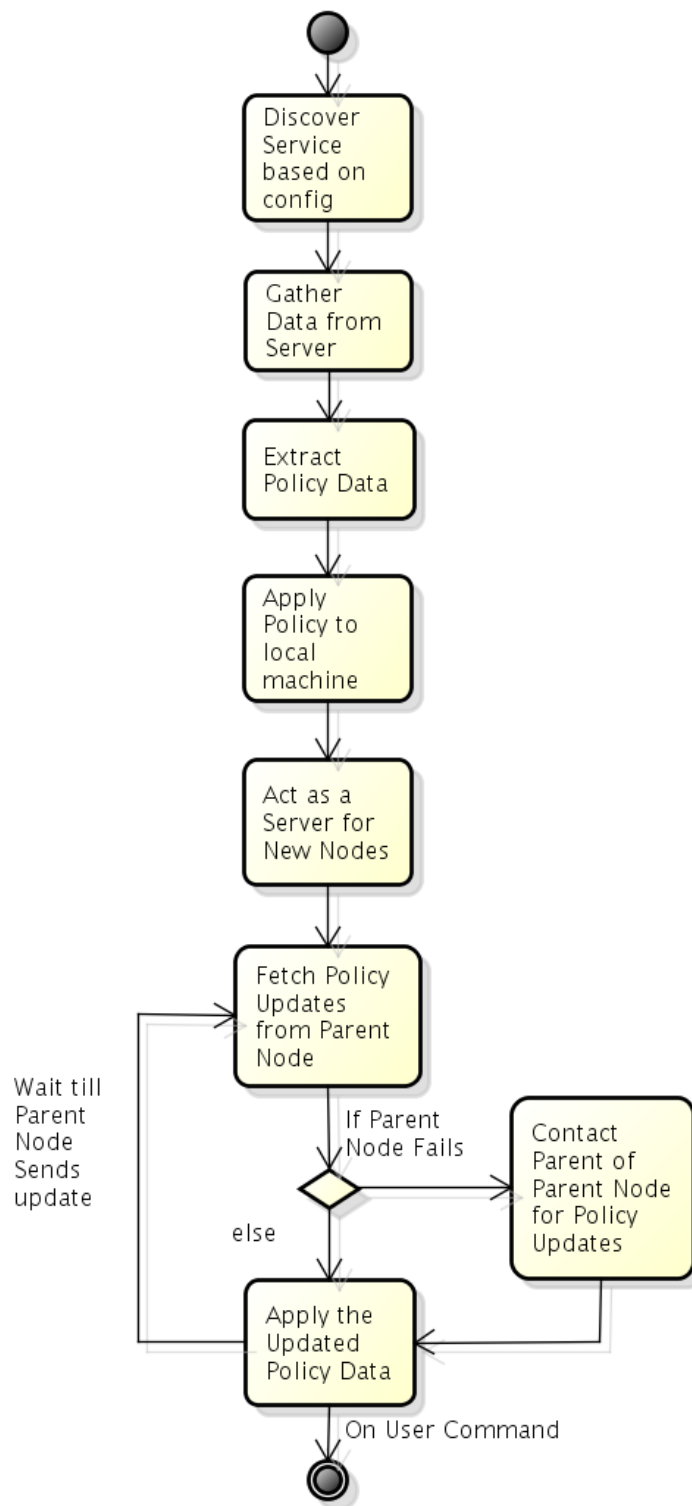


Figure 4.6: Activity Diagram of Distributed Service - Decentralized Approach

Chapter 5

Implementation

5.1 Software Requirements

- Any linux OS with kernel version 2.6.30 or higher with bash shell version 4 or higher
- Python version 2.6 (for network programming)
- SELinux version 3.7.19-195
- avahi 0.6.25 (to publish our service in a network)
- avahi-tools 0.6.25 (to fetch the published service from network)
- grep, sed, touch, cut sort, uniq, rm tools (for text/file manipulations)
- sha256sum tool (for HASH creation and checking)

Note :

- avahi-daemon should be running on client and server node
- port 1986 should be kept open in iptables for Service itself

- port 5353 should be kept open in iptables for avahi-daemon
- for installation steps of SELinux and python refer Appendix B

5.2 Multi System Policy

SELinux has multiple linux daemon policies integrated into targeted policy. This targeted policy is a binary policy which can be loaded into selinux directly. Although this policy is readable through some tools like apol and other selinux policy analysis tools, binary policy can not be modified directly by user through any interface.

In case, where we want to modify that binary policy, we need to have the policy source, which contains the Type Enforcement files for each daemon. For example, we want to modify the policy for apache daemon, we need the apache.te file which is the source for the Type Enforcement commands. The source code of Targeted Policy is available on RedHat source website. RedHat is the major implementor and contributor of the SELinux throughout the linux history.

This policy source need some work (run make commands for some procedures) to get the actual Type enforcement files. After getting the Type enforcement files, we can compile them and use them.

Before compiling the policy, the version of SELinux itself is to be matched with the policy Type Enforcement File. The reason is that, SELinux policy syntax and parameters are updated with version of SELinux itself. So unmatched version of SELinux with policy will not work.

In current implementation, the version of SELinux is 3.7.19-195. Reference Policy version 3.7.19 is compatible with this SELinux Version. The target operation system is CentOS 6.0 (i686 Architecture). It is to be notice that SELinux or SELinux policy are not dependent on any target system architecture. No change in SELinux or SELinux policy syntax is required according to the Destination Architecture.

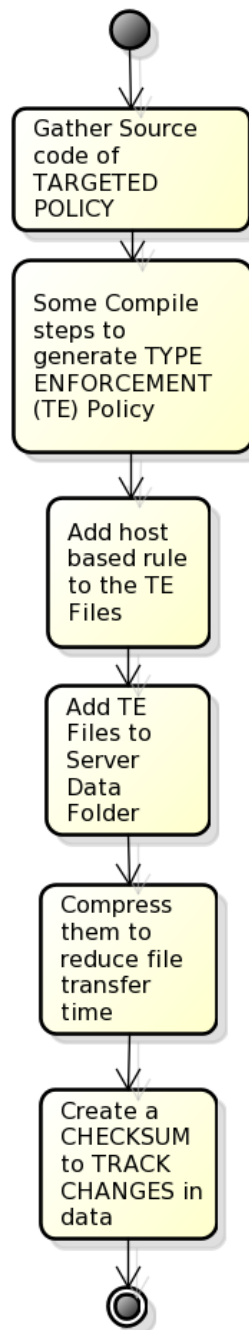


Figure 5.1: Multi System Policy - Create a Type Enforcement Policy from source

Defining the Multi-System Policy is the first thing to do, in the overall implementation. We define the Multi-system policy by @ operator in a very easy manner.

General SELinux Type Enforcement Statement is like :

```
allow httpd_t httpd_exec_t:file { getattr open read execute ioctl lock };
```

We change it to the Node based Syntax like :

```
allow httpd_t@node1 httpd_exec_t:file { getattr open read execute ioctl lock };
```

```
allow httpd_t@node2 httpd_exec_t:file { open read };
```

As the node based policy suggests, that node1 httpd_t subject gets the permissions like getattr, open, read, execute, ioctl and lock on the httpd_exec_t labeled file objects, while node2 httpd_t gets the permissions of open and read on httpd_exec_t labeled file objects.

This way we can define each individual node policy in a Type Enforcement file.

Although there are many rules which we want every node to apply. In that case, we do not use the @ operator, for example,

```
allow httpd_t httpd_exec_t:file entrypoint;
```

this rule should be applied to all the nodes, which use this policy, as there is no @ operator defined for node based policy.

5.3 Automatic Service Discovery

Automatic Service Discovery(ASD) is next step into implementing the service. Using Automatic Service Discovery, very less client configuration is required.

ASD is provided by the ZeroConf protocol, which broadcasts the services on network and allows the discovery of services on the network. Open Source implementation of

this protocol is avahi. Avahi is a collection of tools which implements all the required tools to establish the service discovery in network. Avahi is also provided as the basic package in almost every linux flavors, so no additional package is required to act as a client or server in our requirement.

To Participate into the service discovery, avahi daemon should be running on the node. This Avahi-Daemon will discover all the service information available on the network, and also add new information to the same according to the configuration.

To Publish a service on network, avahi daemon required the service file in the configuration directory of avahi itself. Generally this directory is `/etc/avahi/service`. In this directory we can create a new file ending with “service” extension.

After creating this service file, avahi-daemon load this file itself into the daemon. So avahi daemon does not require to reload when we add new service to the avahi configuration.

5.4 Client Server Architecture

Following experimental setup will be used:

- node A (IP : 172.16.10.81)(Initial Server),
- node B (IP : 172.16.10.91)(client of node A and server program running)
- node D (IP : 172.16.10.92)(client of node A and server program running)
- node C (IP : 172.16.10.71)(client of node B and server program running)
- node E (IP : 172.16.10.72)(client of node B and server program running)

Client server programming is done in Python. Python is the modern way to implement the system and web programming. It provides a very easy to use functions and interface, along with readily available libraries to achieve desired functionality.

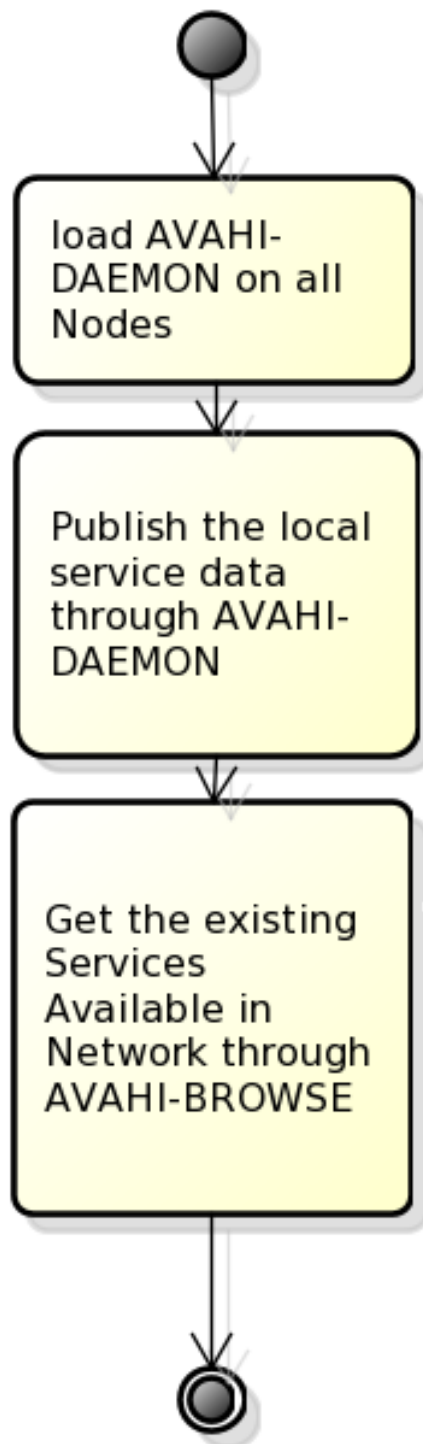


Figure 5.2: Automatic service Discovery - Using Avahi Daemon

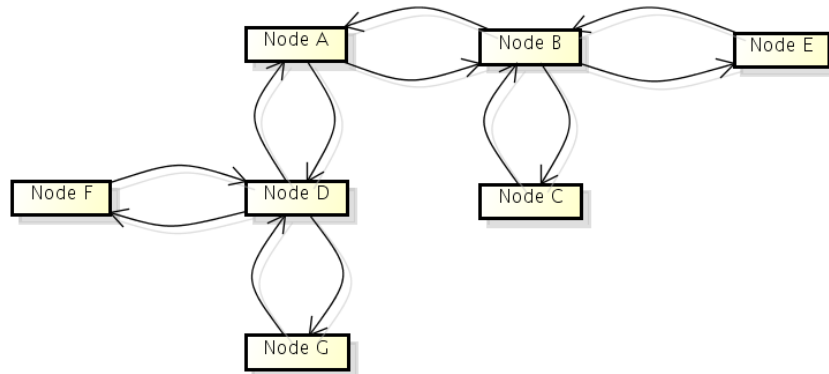


Figure 5.3: Experimental Setup - Server client relation for setup

Python Provides a very simple way to create a client server architecture using XMLRPC Library. This library is inbuilt into python, no additional packages are required for this client server programs to function.

5.4.1 Files & folder used for programming

Following is the list of files and folders used :

- /etc/diss/diss-client.py - Client Program {Refer Appendix C1}
- /etc/diss/disslinux.service - avahi daemon service file {Refer Appendix C3}
- /etc/diss/diss-server.py - Server Program {Refer Appendix C2}
- /etc/diss/host.dictionary.init - initial Host Dictionary {Refer Appendix C5}
- /etc/diss/README - Information about how to use the programs {Refer Appendix C6}
- /etc/diss/serveraddress - Address of permanent server address for client
for example, 172.16.10.91

- /etc/diss/serveraddresses.temp - Address of all available servers in network in new lines

for example,

172.16.10.91

172.16.10.81

172.16.10.71
- /etc/diss/selinux.sh - SELinux client policy updation script {Refer Appendix C4}
- /var/diss/selinux/SELINUX.HASH - hash file fetched from server {Refer Appendix C7}
- /var/diss/selinux/HASH - hash file of the current data directory {Refer Appendix C7}
- /var/diss/selinux/.LOCK - lock file this is a blank file
- /var/diss/selinux/data/ - directory to hold the data fetched from server
- /var/diss/selinux/incoming/ - data furnished to use on client
- /var/diss/selinux/patched/ - data applied to client service

5.4.2 Client algorithm

1. Client search for all available servers on network, if not already found (node C will use the avahi-browse and found node A,B and D as a service provider, node C will save the list of found clients in “serveraddress.temp” file)
2. If parent is already defined, client connect to the parent server using shared KEY/HASH, or client will randomly choose the parent from available list and try to connect (node C does not have file named “serveraddress” so parent is not already defined, node C will use the “serveraddresses.temp” file and choose the random address, for example it chooses node B as a server),

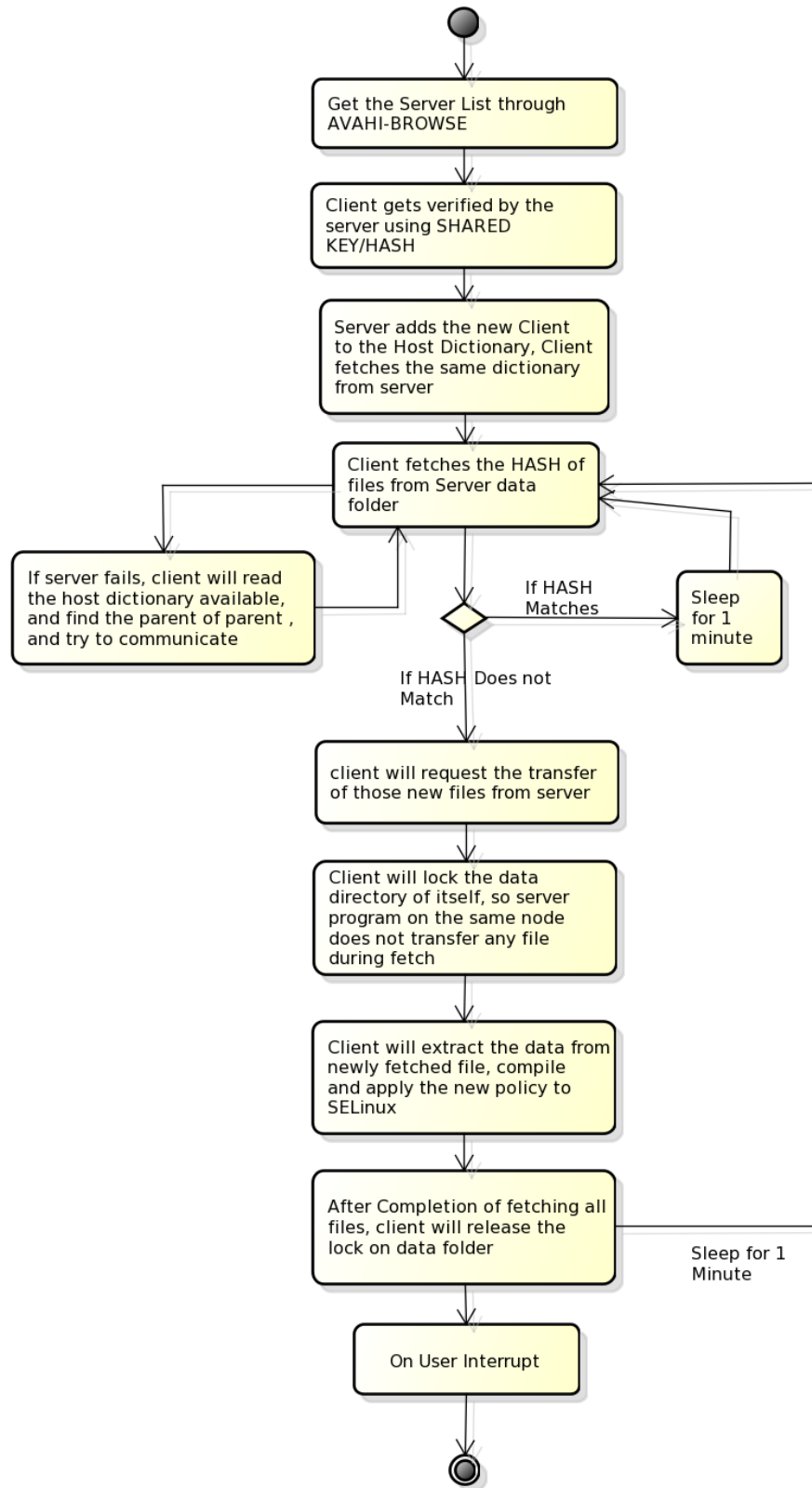


Figure 5.4: Client Data Transfer Sequence - Updation of data on client node

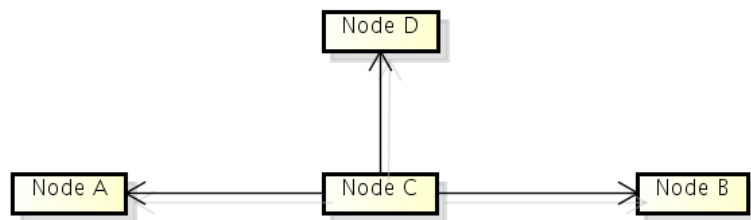


Figure 5.5: Step 1 - Search for available servers if not already defined

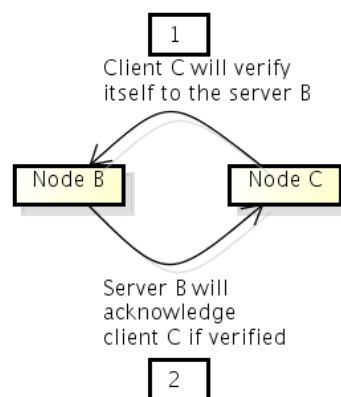


Figure 5.6: Step 2 - Client try to Connect to Server using Shared KEY

3. Once connection is established, client will ask for the host dictionary and hash file of the data on server, then that hash file is checked against the current data on client (node C will try to communicate with node B, if connection is established node B will add node C to the host dictionary file named "host.dictionary", node C will fetch HASH file from node B and also the host.dictionary file),

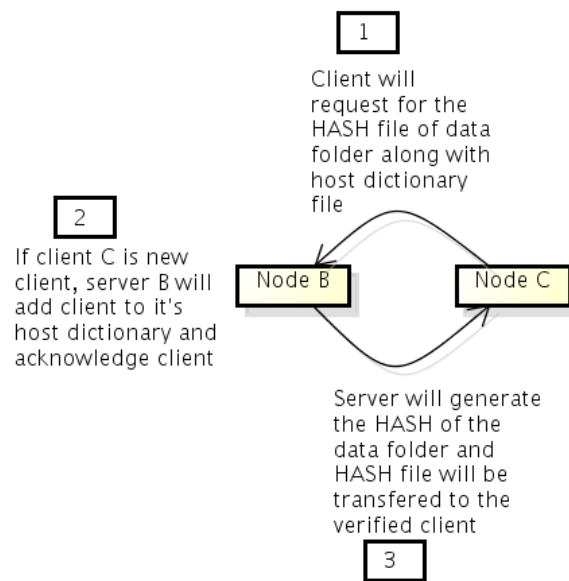


Figure 5.7: Step 3 - Client will request the HASH file of data folder

4. If hash does not match, client will create a lock on the data directory and ask for those files from server (node C checks the HASH file to the currently available data folder, if the HASH file does not match node C will create a lock file on the data folder of itself),
5. Server will transfer those files to client if no lock file present on server, after completion of transfer of files, client will unlock the data directory (node C will then fetch the mismatch hash files (for example apache.te) from node B),
6. Client will extract the specific data from the received data and put into incoming directory (node C will extract the data related to itself from the newly fetched file and put that file (apache.te) into incoming folder),

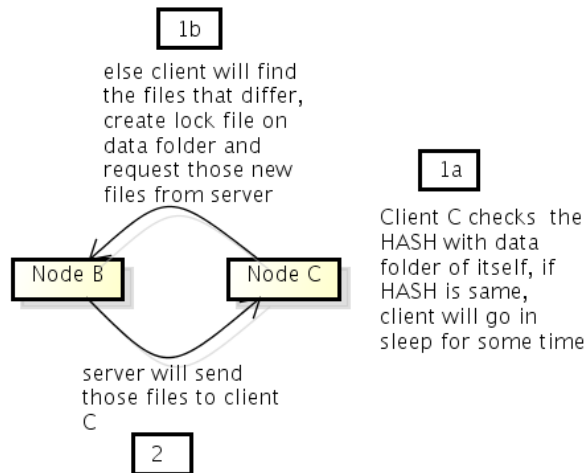


Figure 5.8: Step 4 and 5 - If HASH does not match, client will request those files

7. Client will build the policy from received file (node C will build a policy from newly fetched file, apach.te),
8. Client will load that built policy into SELinux (node C will use semodule command to load the policy into SELinux of itself),

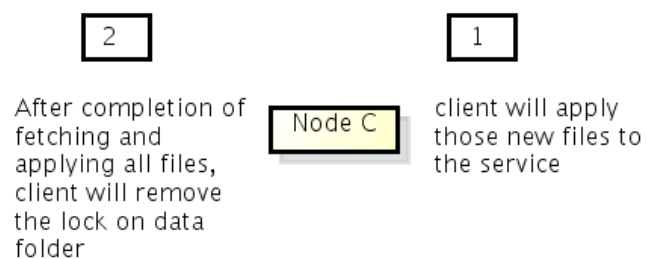


Figure 5.9: Step 6,7 and 8 - Client will apply those new file to the service

9. Client will stay in contact with the server, if server fails in between, client will load the host dictionary and find out the parent of parent, and try to connect to

that new parent (node C goes to sleep for some time, then repeat the above steps except first step, if node C gets the socket error in establishing communication with node B, node C will load the file “host.dictionary” and find the parent of parent of node B, which is node A, now node C will try to communicate with node A, until node C restarts the client program, it will stay in communication with node A, after restart of client program on node C, it will try to communicate with node B)

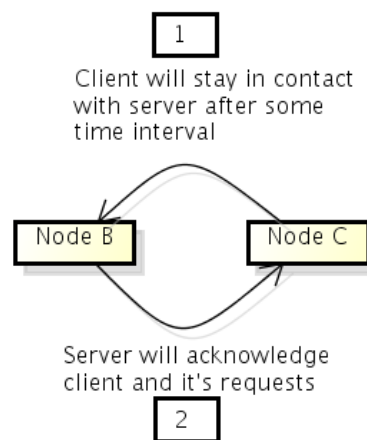


Figure 5.10: Step 9 - Client will stay connected to the server

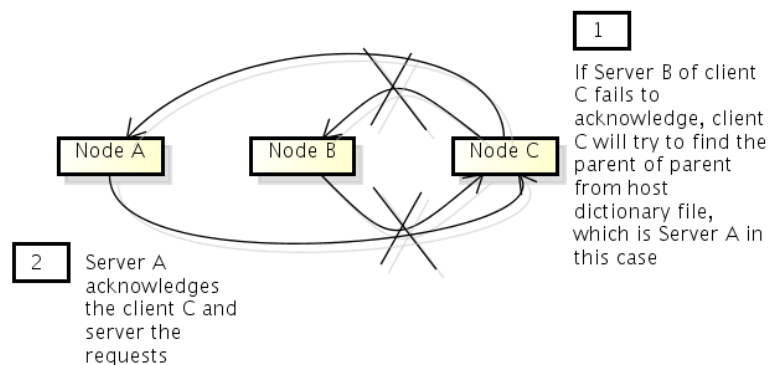


Figure 5.11: Failover - If server fails, client will connect to the parent of parent

5.4.3 Server algorithm

1. Add our service to avahi-daemon (for example node B, server program will add a file to avahi-daemon service folder, which will allow discovery of service on every node available in network),
2. Verify clients by Shared KEY/HASH (node B will verify every requests coming from client by shared KEY/HASH),
3. Add new clients to the host dictionary (if node B gets an authenticated request from new node, it will add that node to the “host.dictionary” file, else if client is already in the dictionary, node B will continue to next step),
4. Create a HASH of data folder and transfer that to client (when node B gets the HASH file request from any node, it will generate the HASH file of data folder, and transfer that file to the requested client),
5. On client request transfer the given filename to client (node B will transfer any requested file from data folder, to authenticated client node)

5.5 Client Output

5.5.1 Normal client output

```
root@nodeC# python diss-client.py
```

```
trying http://172.16.10.91:1986
```

```
Trying to register client
```

```
communication established
```

```
fetching hash file
```

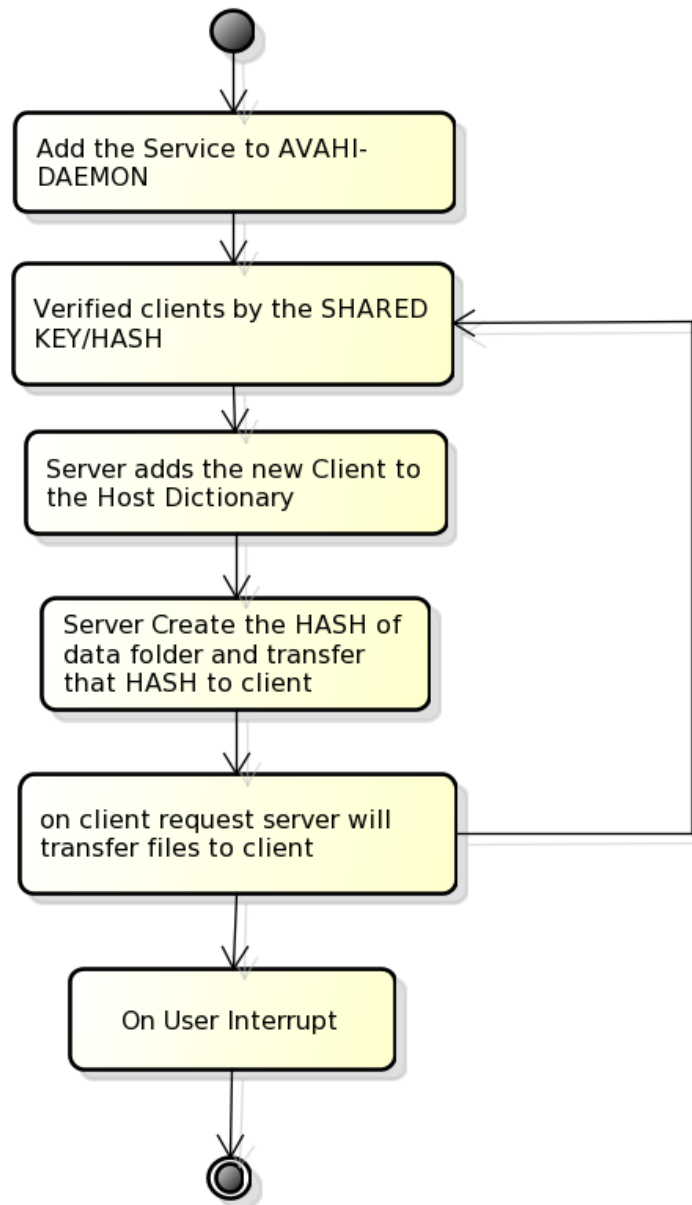


Figure 5.12: Server Data Transfer Sequence - data provided by server node

checking hash of local files
lock file created
file downloaded and applied : apache.te,gz
files download complete: now releasing lock
now going to sleep

5.5.2 Client output in case of failover

```
root@nodeC# python diss-client.py
```

trying http://172.16.10.91:1986
Trying to register client
(Original Parent Failed so trying the parent of parent : Node A)
Trying http://172.16.10.81:1986
communication established
fetching hash file
checking hash of local files
lock file created
file downloaded and applied : XYZ.te,gz
files download complete: now releasing lock
now going to sleep

5.6 Server Output

```
root@nodeB# python diss-server.py
```

```
Client not registered adding new entry
```

```
Adding Server Address :172.16.10.91
```

```
Adding client Address :172.16.10.71
```

```
Host array entry added to file
```

```
172.16.10.71 - - [14/Apr/2013 18:49:21] "POST /RPC2 HTTP/1.0" 200 -
```

```
Server Address :172.16.10.91
```

```
client Address :172.16.10.71
```

```
Client Already Registered OK
```

```
172.16.10.71 - - [14/Apr/2013 18:49:31] "POST /RPC2 HTTP/1.0" 200 -
```

```
Creating HASH FILE
```

```
Transferring HASH FILE
```

```
172.16.10.71 - - [14/Apr/2013 18:49:46] "POST /RPC2 HTTP/1.0" 200 -
```

```
Transferring DICTIONARY FILE
```

```
172.16.10.71 - - [14/Apr/2013 18:49:56] "POST /RPC2 HTTP/1.0" 200 -
```

```
Transferring FILE apache.te.gz
```

```
172.16.10.71 - - [14/Apr/2013 18:49:58] "POST /RPC2 HTTP/1.0" 200 -
```

5.7 Libraries used by python programs

5.7.1 Client Program Libraries

- xmlrpclib - for client server programming
- subprocess - to execute the shell commands within python code
- socket - for socket functions
- time - used in sleep code
- os - to perform os based functionality within python code
- pickle - to store hierarchical data of nodes
- random - to pick random server address and connect to it

5.7.2 Server Program Libraries

- xmlrpclib - for client server programming
- SimpleXMLRPCServer - to create and bind server program to a port
- subprocess - to execute the shell commands within python code
- time - used in sleep code
- os - to perform os based functionality within python code
- pickle - to store hierarchical data of nodes

5.8 Updation of Policy on Destination Node

Type Enforcement file from server provides the basis to create and compile a module for SELinux. From this Type Enforcement file, client will fetch it's related information from the file, compile it, and create a module from that file.

This module will be loaded into SELinux once it is verified and contains the higher module version than the module already loaded into SELinux. SELinux does not require to reload itself or restart any other service to apply the new policy. The newly loaded policy will be save permanently on the secondary storage also it will be updated into primary memory directly.

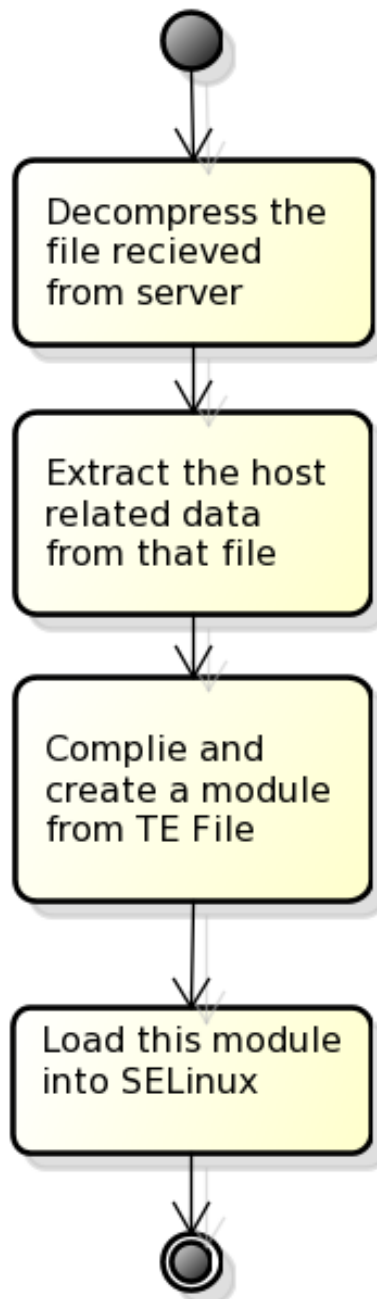


Figure 5.13: Update Policy on Client Nodes - Compiling and linking

5.9 Override Automatic Discovered Parent

Super User always have a facility to override the parent address, by changing the content of the file “serveraddress”. This file contains the address of the parent in the first and only line, super user can always edit this file and change the address for the client program to connect to the new desired server. If there is any error in defining the server address, for example the server address does not exists, then client program will not run at any point, as it would not be able to communicate with the server and also not able to find out the parent of parent in any case.

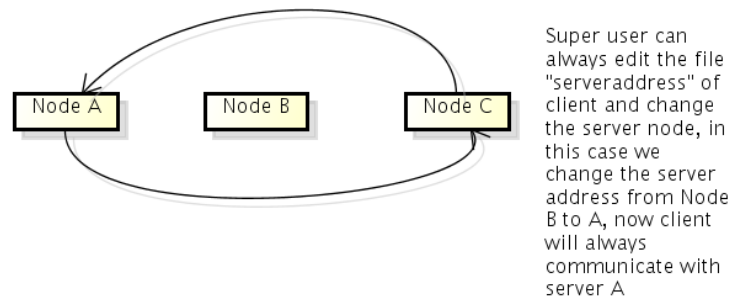


Figure 5.14: Overriding the parent address - serveraddress can be changed at any time and require client program restart to apply the new address

Chapter 6

Conclusion and Future work

6.1 Conclusion

- This dissertation created a simple distributed architecture which will ease our requirement to transfer the data from server to client with very less client configuration. Based on client node, the data will be applied to SELinux, which saves a lot of time to administer SELinux framework of all the nodes in a network, by setting only their hostname in the configuration file, other work will be automatically done by the python code.
- It uses libraries and packages of python as well as SELinux, which are already provided by basic package installation, so that makes it very easy to port this to any linux system. It does not require any change in SELinux or Kernel or any other part of linux system, which makes it very easy for new or experienced user to use.
- The same concept can be applied to any service which requires the data to be fetched from server and apply to itself. This effort to create a distributed architecture will clear path for many future applications, which requires configuration file or data file to function (almost every daemon in linux), for example

1. Web Server (e.g. httpd,tomcat)
2. FTP Server (e.g. nfsd,ftpd,smb)
3. DNS Server (e.g. BIND)
4. Database Servers (e.g. mysql,postgresql)
5. iptables - Firewall Service of Kernel
6. Authentication (e.g. passwd,shadow files)
7. Mail Server (e.g. postfix,sendmail)
8. Remote Login (e.g. sshd,telnet) etc.

6.2 Future work

This project can have the following enhancements :

- This architecture can be embedded into SELinux itself to allow distributed policy fetching
- Kernel changes can be made to allow SELinux to fetch the policy from distributed environment and apply the policy
- Create GUI for some tasks, like Create Multi-System Policy, change variables of server client program
- Multiple Parent Failure can be handled by client program, for example, node C is client of node B and node B is client of node A , if both node A and node B fails, client will contact parent of node A

Bibliography

- [1] JOSHUA BRINDLE, KAREN VANCE AND CHAD SELLERS. **Enforcing Flexible Access Control in a Networked Policy Domain.** *SELinux Symposium*, 2007. 9, 10, 11, 12, 13
- [2] MAYER, FRANK, KARL MACMILLAN, AND DAVID CAPLAN. **SELinux by Example.** *Prentice Hall*, 2006. 1, 2, 4, 6, 8, 42, 44, 48, 49, 51
- [3] WESLEY J. CHUN. **Core Python Applications Programming.** *Pearson*, 2013.
- [4] MACMILLAN, KARL, JOSHUA BRINDLE, FRANK MAYER, DAVID CAPLAN, AND JASON TANG. **Design and Implementation of the SELinux Policy Management Server.** *SELinux Symposium*, 2006. 21
- [5] ASHWORTH, CHRISTOPHER, AND JAMES ATHEY. **Developing SELinux Management Tools.** *SELinux Symposium*, 2007.
- [6] PEENITO, CHRISTOPHER, FRANK MAYER, AND KARL MACMILLAN. **Reference Policy for Security Enhanced Linux.** *SELinux Symposium*, 2006. 45
- [7] SPENCER, RAY, STEPHEN SMALLEY, MIKE HIBLER, DAVID ANDERSON, AND JAY LEPREAU. **The Flask Security Architecture: System Support for Diverse Security Policies.** *SELinux Symposium*, 1999. 20, 22
- [8] SECURE COMPUTING CORP. **Assurance in the Fluke Microkernel: Formal Security Policy Model.** *CDRL A003, 2675 Long Lake Rd, Roseville, MN 55113*, Feb. 1999. 24

- [9] B. FORD, M. HIBLER, J. LEPREAU, R. MCGRATH, AND P. TULLMANN. **Interface and Execution Models in the Fluke Kernel.** *In Proceedings of the 3rd USENIX Symposium on Operating Systems Design and Implementation*, pages 101116, Feb. 1999.
- [10] R. GRIMM AND B. N. BERSHAD. **Providing Policy-Neutral and Transparent Access Control in Extensible Systems.** *In J. Vitek and C. Jensen, editors, Secure Internet Programming: Security Issues for Distributed and Mobile Objects*, volume 1603 of Lecture Notes in Computer Science. Springer-Verlag, June 1999.
- [11] T. JAEGER, J. LIEDTKE, AND N. ISLAM. **Operating System Protection for Fine-Grained Programs.** *In Proceedings of the Seventh USENIX Security Symposium*, pages 143157, Jan. 1998. 17
- [12] S. KENT AND R. ATKINSON. **Security Architecture for the Internet Protocol.** *RFC 2401, Internet Engineering Task Force*, Nov. 1998. 7, 18
- [13] P. A. LOSCOCO, S. D. SMALLEY, P. A. MUCKELBAUER, R. C. TAYLOR, S. J. TURNER, AND J. F. FARRELL. **The Inevitability of Failure: The Flawed Assumption of Security in Modern Computing Environments.** *In Proceedings of the 21st National Information Systems Security Conference*, pages 303314, Oct. 1998. 3
- [14] D. MAUGHAN, M. SCHERTLER, M. SCHNEIDER, AND J. TURNER. **Internet Security Association and Key Management Protocol (ISAKMP).** *RFC 2408, Internet Engineering Task Force*, Nov. 1998. 4
- [15] M. CARNEY AND B. LOE. **A Comparison of Methods for Implementing Adaptive Security Policies.** *In Proceedings of the Seventh USENIX Security Symposium*, pages 114, Jan. 1998. 5, 6
- [16] A. CHITTURI. **Implementing Mandatory Network Security in a Policy-flexible System.** *Masters thesis, University of Utah, 1998*, pp. 70. 19
- [17] T. FRASER AND L. BADGER. **Ensuring Continuity During Dynamic Security Policy Reconfiguration in DTE.** *In Proceedings of the 1998 IEEE Symposium on Security and Privacy*, pages 1526, May 1998.

- [18] T. MITCHEM, R. LU, AND R. OBRIEN. **Using Kernel Hypervisors to Secure Applications.** *In Proceedings of the Annual Computer Security Applications Conference*, Dec. 1997.
- [19] S. G. RAVI SANDHU, VENKATA BHAMIDIPATI AND C. YOUMAN. **The ARBAC97 Model for Role-Based Administration of Roles: Preliminary Description and Outline.** *In Proceedings of the Second ACM Workshop on Role-Based Access Control*, pages 4150, Nov. 1997. 17, 36
- [20] B. FORD, G. BACK, G. BENSON, J. LEPREAU, A. LIN, AND O. SHIVERS. **The Flux OSKit: A Substrate for OS and Language Research.** *In Proc. of the 16th ACM Symp. on Operating Systems Principles*, pages 3851, St. Malo, France, Oct. 1997.
- [21] J. S. SHAPIRO. **EROS: A Capability System.** *Technical Report Technical Report MS-CIS-97-04, University of Pennsylvania, Department of Computer and Information Science*, 1997.
- [22] D. S. WALLACH, D. BALFANZ, D. DEAN, AND E. W. FELTEN. **Extensible Security Architectures for Java.** *In Proc. of the 16th ACM Symp. on Operating Systems Principles*, pages 116128, Oct. 1997.
- [23] D. OLAWSKY, T. FINE, E. SCHNEIDER, AND R. SPENCER. **Developing and Using a Policy Neutral Access Control Policy.** *In Proceedings of the New Security Paradigms Workshop. ACM*, Sept. 1996.
- [24] B. FORD, M. HIBLER, J. LEPREAU, P. TULLMANN, G. BACK, AND S. CLAWSON. **Microkernels Meet Recursive Virtual Machines.** *In Proceedings of the Symposium on Operating Systems Design and Implementations*, pages 137151, Oct. 1996. 9
- [25] I. GOLDBERG, D. WAGNER, R. THOMAS, AND E. A. BREWER. **A Secure Environment for Untrusted Helper Applications.** *In Proceedings of the 6th Usenix Security Symposium*, July 1996.
- [26] M. I. SELTZER, Y. ENDO, C. SMALL, AND K. A. SMITH. **Dealing With Disaster: Surviving Misbehaved Kernel Extensions.** *In Proc. of the Second Symp. on Operating Systems Design and Implementation*, pages 213227, Seattle, WA, Oct. 1996. USENIX Assoc.

- [27] M. E. ZURKO AND R. SIMON. **User-Centered Security.** *In Proceedings of the New Security Paradigms Workshop*, Sept. 1996.
- [28] S. E. MINEAR. **Providing Policy Control Over Object Operations in a Mach Based System.** *In Proceedings of the Fifth USENIX UNIX Security Symposium*, pages 141156, June 1995.
- [29] T. C. V. BENZEL, E. J. SEBES, AND H. TAJALLI. **Identification of Subjects and Objects in a Trusted Extensible Client Server Architecture.** *In Proceedings of the 18th National Information Systems Security Conference*, pages 8399, 1995.
- [30] B. N. BERSHAD, S. SAVAGE, P. PARDYAK, E. G. SIRER, M. E. FIUCZYNSKI, D. BECKER, C. CHAMBERS, AND S. EGGERS. **Extensibility, Safety, and Performance in the SPIN Operating System.** *In Proc. of the 15th ACM Symp. on Operating Systems Principles*, pages 267284, Copper Mountain, CO, Dec. 1995.
- [31] D. F. FERRAILOLO, J. A. CUGINI, AND D. R. KUHN. **Role-Based Access Control (RBAC): Features and Motivations.** *In Proceedings of the Eleventh Annual Computer Security Applications Conference*, Dec. 1995.
- [32] J. G. MITCHELL, J. J. GIBBONS, G. HAMILTON, P. B. KESSLER, Y. A. KHALIDI, P. KOUGIOURIS, P. W. MADANY, M. N. NELSON, M. L. POWELL, AND S. R. RADIA. **An Overview of the Spring System.** *In A Spring Collection. Sun Microsystems, Inc.*, 1994.
- [33] M. I. BUSHNELL. **Towards a New Strategy of OS Design.** *GNUs Bulletin*, 1(16), Jan. 1994.
- [34] M. D. ABRAMS. **Renewed Understanding of Access Control Policies.** *In Proceedings of the 16th National Computer Security Conference*, pages 8796, Oct. 1993. 56
- [35] T. FINE AND S. E. MINEAR. **Assuring Distributed Trusted Mach.** *In Proceedings IEEE Computer Society Symposium on Research in Security and Privacy*, pages 206218, May 1993.

Appendix A

SELinux Policy Objects

A.1 Object Classes and Permissions

The object classes have matching declarations in the kernel, meaning that it is not trivial to add or change object class details. The same thing is true for permissions. Development work is ongoing to make it possible to register and unregister classes and permissions dynamically. Permissions are the actions that a subject can take on an object, if the policy allows it. These permissions are the access requests that SELinux actively allows or denies. There are several common sets of permissions defined in the targeted policy, in `$SELINUX_SRC/flask/access_vectors`. These allow the actual classes to inherit the sets, instead of rewriting the same permissions across multiple classes:

```
# Define a common prefix for file access vectors.
#
common file
{
    ioctl
    read
    write
    create
```

```
getattr
setattr
lock
relabelfrom
relabelto
append
unlink
link
rename
execute
swapon
quotaon
mounton
}

# Define a common prefix for socket access vectors.
#
common socket
{ # inherited from file
ioctl
read
write
create
getattr
setattr
lock
relabelfrom
relabelto
append
# socket-specific
bind
connect
```

```
listen
accept
getopt
setopt
shutdown
recvfrom
sendto
recv_msg
send_msg
name_bind }

# Define a common prefix for ipc access vectors.
#
common ipc
{
create
destroy
getattr
setattr
read
write
associate
unix_read
unix_write
}
```

Following the common sets are all the access vector definitions. The definition is structured this way: `class class_name [inherits common_name] { permission_name ... }`. A good example is the `dir` class, which inherits the permissions from the `file` class, and has additional permissions on top:

```
class dir
```

```
inherits file
{
  add_name
  remove_name
  reparent
  search
  rmdir
}
```

Another example is the class for `tcp_socket`, which inherits the `socket` set plus having its own set of additional permissions:

```
class tcp_socket
inherits socket
{
  connectto
  newconn
  acceptfrom
  node_bind
}
```

A.2 TE Rules - Attributes

Policy attributes identify as groups sets of security types that have a similar property. These groups can be controlled by fewer, overarching rules. The relationship is many-to-many: a type can have any amount of attributes, and an attribute can be associated with any number of types.

```
# The domain attribute identifies every type that can be
# assigned to a process. This attribute is used in TE rules
```

```
# that should be applied to all domains, e.g. permitting
# init to kill all processes.
attribute domain;
# Identifies all default types assigned to packets received
# on network interfaces.
attribute netmsg_type;
```

Here are a few noteworthy attributes.

httpdcontent The purpose of this attribute is to group together the various types associated with the policy for Apache HTTP. Because of the complexity of the httpd configuration, the targeted policy includes Boolean values that allow you to grant blanket permissions for httpd content types. This helps Web applications and built-in scripting, such as PHP for Apache HTTP, to work with the content. The types in this attribute are:

```
{# This is an aliasing relationship
httpd_sys_content_t: httpd_sysadm_content_t, httpd_user_content_t
# These types handle different permissions sets for scripts
httpd_sys_script_ro_t
httpd_sys_script_rw_t
httpd_sys_script_ra_t
}
```

The first line in the attribute group specifies that `httpd_sys_content_t` is an alias for `httpd_sysadm_content_t` and `httpd_user_content_t`.

file_type This attribute is for all the types that are assigned to files, allowing for easier association of all file types to various kinds of file system needs. This attribute makes it more convenient to allow specific domains access to all file types. The list of types associated with the `file_type` attribute is greater than 170 types:

...

device_t
xconsole_device_t
file_t
default_t
root_t
mnt_t
home_root_t
lost_found_t
boot_t
system_map_t
boot_runtime_t
tmp_t
etc_t: hotplug_etc_t
shadow_t
ld_so_cache_t
etc_runtime_t
fonts_t
etc_aliases_t
net_conf_t: resolv_conf_t
lib_t
shlib_t

netif_type, port_type, and node_type These attributes relate to network activity by domains. The `netif_type` identifies the types associated with network interfaces, allowing policy to control sending, receiving, and various operations on the interface:

netif_t
netif_eth0_t
netif_eth1_t
netif_eth2_t

netif_lo.t
netif_ippp0.t
netif_ipsec0.t
netif_ipsec1.t
netif_ipsec2.t

The `port_type` attribute is associated with all types that are assigned to port numbers. This allows SELinux to control port binding, meaning daemons are restricted in using a port depending on the type assigned to the port:

dns_port.t
dhcpd_port.t
http_cache_port.t
port.t
reserved_port.t
http_port.t
pxe_port.t
smtp_port.t
mysqld_port.t
rndc_port.t
ntp_port.t
portmap_port.t
postgresql_port.t
snmp_port.t
syslogd_port.t

The `node_type` is for types assigned to network nodes or hosts, allowing SELinux to control traffic to and from the node:

node.t
node_lo.t
node_internal.t

node_inaddr_any_t
node_unspec_t
node_link_local_t
node_site_local_t
node_multicast_t
node_mapped_ipv4_t
node_compat_ipv4_t

fs_type This attribute identifies all types assigned to file systems, including non-persistent file systems. The fs_type attribute is used in TE rules to allow most domains to obtain overall file system statistics, and for some specific domains to mount any file system. Here are the SELinux file types that are part of fs_type:

devpts_t: sysadm_devpts_t, staff_devpts_t, user_devpts_t
fs_t
eventpollfs_t
futexfs_t
bdev_t
usbfs_t
nfsd_fs_t
rpc_pipefs_t
binfmt_misc_fs_t
tmpfs_t
autofs_t
usbdevfs_t
sysfs_t
iso9660_t
romfs_t
ramfs_t
dosfs_t

cifs.t: sambafs.t

nfs.t

proc.t

security.t

exec.type This attribute groups together all types that are assigned to entry point executables. Any TE rules and assertions that should be applied to all entry point executables use this attribute. Here are the domains in this attribute:

ls_exec.t

shell_exec.t

httpd_exec.t

httpd_suexec_exec.t

httpd_php_exec.t

httpd_helper_exec.t

dhcpd_exec.t

hotplug_exec.t

initrc_exec.t

run_init_exec.t

init_exec.t

ldconfig_exec.t

mailman_queue_exec.t

mailman_mail_exec.t

mailman_cgi_exec.t

depmod_exec.t

insmod_exec.t

update_modules_exec.t

sendmail_exec.t

mysqld_exec.t

named_exec.t

ndc_exec_t
nscd_exec_t
ntpd_exec_t
ntpdate_exec_t
portmap_exec_t
postgresql_exec_t
rpm_exec_t
snmpd_exec_t
squid_exec_t
syslogd_exec_t
udev_exec_t
udev_helper_exec_t
winbind_exec_t
ypbind_exec_t

domain This attribute is for all types that can be assigned to a process. This is the method for identifying what is a domain in SELinux. In other Type Enforcement systems, domains may be implemented separately from types. In SELinux, domains are essentially types with the domain attribute. This attribute allows you to have rules that can be applied to all domains, such as allowing init to send signals to all processes. Another example is the following rule that allows all processes to perform a search on directory objects that have a type of `var_t` or `var_run_t`, that is, the directories `/var` and `/var/run`:

```
allow domain var_run_t var_t : dir search ;
```

Here are the domains covered by this attribute:

unconfined_t, kernel_t, init_t, initrc_t, sysadm_t, rpm_t,
rpm_script_t, logrotate_t
mount_t
httpd_t
httpd_sys_script_t
httpd_suexec_t

httpd_php_t
 httpd_helper_t
 dhcpd_t
 ldconfig_t
 mailman_queue_t
 mailman_mail_t
 mailman_cgi_t
 system_mail_t
 mysqld_t
 named_t
 ndc_t
 nscd_t
 ntpd_t
 portmap_t
 postgresql_t
 snmpd_t
 squid_t
 syslogd_t
 winbind_t
 ypbind_t

This attribute identifies all the types that are assigned to any of the reserved network ports, that is, ports numbered lower than 1024. The attribute is used to control binding. An example binding rule is followed here by the types that are part of this attribute:

```

reserved_port_type      #The allow rule permits the domain portmap_t to bind to
a
#port with a type of portmap_port_t, which is one of the #types identified
by the reserved_port_type attribute. The #dontaudit rule tells SELinux to
never audit the access of
#portmap_t to a reserved_port_type.
  
```

```
allow portmap_t portmap_port_t: udp_socket tcp_socket
name_bind;

dontaudit portmap_t reserved_port_type:tcp_socket name_bind;

# Types associated with the reserved_port_type attribute

http_port_t

smtp_port_t

rndc_port_t

ntp_port_t

portmap_port_t

snmp_port_t

syslogd_port_t
```

A.3 TE Rules - Types

SELinux uses types in various ways. After they are declared, they can be used to make rules for the transition decision process, type changing process, and access vector decisions and assertions.

Domains are types applied to processes, identified by the type having the domain attribute. The same type is used for the process itself and the associated /proc file. Typically, you see the domain used as the source context for system operations, that is, the domain is the doer. A domain can be a target context, such as when init is sending process signals to a daemon. With every SELinux transaction involving at least one domain, the number and kind of domains is central to the complexity of the security policy. More domains means finer security control, with a matching increase in configuration and maintenance difficulties.

Type Declaration

This syntax defines how types are declared. A type must be declared before rules can be written about it. The targeted daemons have their top-level domain declared through the macro `daemon_domain()`

```
## Syntax of a type declaration
type <typename> [aliases] [attributes];

## Examples
type httpd_config_t, file_type, sysadmfile;
# httpd_config_t is a system administration file
type http_port_t, port_type, reserved_port_type;
# http_port_t is a reserved port, number less than 1024
type httpd_php_exec_t, file_type, sysadmfile, exec_type;
# httpd_php_exec_t is a sysadmin file that is an entry point
# executable
```

Type Transitions

A type transition results in a new process running in a new domain different from the executing process, or a new object being labeled with a type different from the source doing the labeling. The rules define what domain and file type transitions occur by default. The domain transition default can be overridden if the process explicitly requests a particular context. File transition default is actually inherit-from-parent, that is, the new file receives its context from the parent directory unless an explicit transition rule makes it inherit-from-creator. For example, the directory `/` has a type of `user_home_dir_t`, and policy specifies that files created in a directory with that type are labeled with `user_home_t`. Transitions are defined through macros that combine the `type_transition` rule with a set of allow rules. The allow rules are macros with variables that support common transitioning needs.

```
## General syntax of a transition
type_transition <source_type(s)> <target_type(s)> : <class(es)> <new_type>

# Note that all excepting the new_type can be
# multiple types and classes, surrounded by brackets

## Domain transition syntax
```

```

type_transition <current_domain> <type_of_program> : process <new_domain>
# note that the object class is fixed to the process attribute
## Domain transition examples
type_transition httpd_t httpd_sys_script_exec_t:process httpd_sys_script_t;
# When the httpd daemon running in the domain httpd_t executes
# a program of the type httpd_sys_script_exec_t, such as a CGI
# script, the new process is given the domain of
# httpd_sys_script_t
type_transition initrc_t squid_exec_t:process squid_t;
# When init exec()s a program of the type squid_exec_t, the new
# process is transitioned to the squid_t domain
## New object labeling syntax
type_transition <creating_domain> <parent_object_type> <class(es)>
<new_type>
# Note that multiple classes are allowed using the # brackets ##
New object labeling example
type_transition named_t var_run_t:sock_file named_var_run_t;
# When a process in the domain named_t creates a socket file
# in a directory of the type var_run_t, the socket file is
# given the type named_var_run_t. The directory with the
# type var_run_t is defined in the policy as /var/run/.
file_type_auto_trans(named_t, var_run_t, named_var_run_t, sock_file)
# This rule evokes the file_type_auto_trans macro from # $SELINUX_SRC/
macros/core_macros.te, ultimately feeding the 4 # variables in to the
macro file_type_trans($1,$2,$3,$4)

```

A.4 TE Rules - Access Vectors

Access vectors (AVs) are the rules that allow domains to access various system objects. An AV is a set of permissions. A basic AV rule is a subject and object pair of types, a class

definition for the object, and a permission for the subject. There is a general rule syntax that covers all the kinds of AV rules:

```
<av_kind> <source_type(s)> <target_type(s)> : <class(es)> <permission(s)>
```

All AV rules are considered by the policy enforcement engine as two types, one class, and one access permission. However, rules are written using attributes, sets, and macros to be more efficient. AV rules are simplified during policy compilation. This section describes the kinds of access vectors used in the AV rule at `av_kind`. `av_kind` is one of three rule types:

allow permit a subject to act in a specific way with an object. The rule here allows named (in the domain of `named_t`) to perform a search of a directory with the type `sbin_t` (for example, `/sbin`, `/usr/sbin`, `/opt/sbin`, etc.):

```
allow named_t sbin_t:dir search;
```

If the ruling results in a denial, the denial is audited (that is, logged). Granted permission events are not logged.

auditallow when the permission is granted, log the access decision. In the targeted policy, there is only one `auditallow` rule. This rule logs usage of certain SELinux applications, for example logging `avc: granted setenforce` when allowing `setenforce`:

```
auditallow unconfined_t security_t : security load_policy setenforce setbool ;
```

dontaudit never audit a specific access denial. This is used when a program is attempting an action that is not allowed by policy, and the resulting denials are filling the log, but the denial is not affecting the application doing its tasks. This AV lets you silently deny and ignore the access violation. For example, this `dontaudit` rule says to ignore when the `named_t` domain attempts to read or get attributes on a file with the `root_t` type. Denial of this access attempt does not effect named doing its job, so the denial is ignored to keep the logs clean:

```
dontaudit named_t root_t:file getattr read ;
```

There is one additional AV rule, `neverallow`. This AV assertion, defined in `$SELINUX_SRC/assert.te`, is not part of the regular permission checking. The purpose of this rule is to

declare access vectors that must never be allowed. These are used to protect against policy writing mistakes, especially where macros can provide unexpected rights. These assertions are checked by the policy compiler, `checkpolicy`, when the policy is built, but after the entire policy has been evaluated, and are not part of the runtime access vector cache. Here is the syntax and an example. In practice, a wildcard character `*` is often used to cover all instances possible in a rule field. The syntax is different in that it is possible to use `ifdef()` statements as sources or targets:

```
# Syntax for AV assertion
```

```
neverallow <source_name(s)> <target_name(s)> : <class(es)> <permission(s)>
```

In this example from `assert.te`, the `neverallow` rule verifies that every type that a domain can enter into has the attribute `domain`. This prevents a rule from elsewhere in the policy allowing a domain to transition to a type that is not a process type. The tilde in front, `~domain`, means “anything that is not a domain”:

```
# Verify that every type that can be entered by
```

```
# a domain is also tagged as a domain.
```

```
#
```

```
neverallow domain ~domain:process transition;
```

A.4.1 Understanding `avc` Message

When SELinux disallows an operation, a denial message is generated for the audit logs. In Red Hat Enterprise Linux, `$AUDIT_LOG` is `/var/log/messages`. This section explains the format of these log messages. For suggestions on using an `avc: denied` message for troubleshooting,

following shows a denial generated when a users Web content residing in `/public.html` does not have the correct label.

```
Jan 14 19:10:04 hostname kernel: audit(1105758604.519:420): avc: denied  
getattr for pid=5962 exe=/usr/sbin/httpd path=/home/ausser/public.html
```

```
dev=hdb2 ino=921135 scontext=root:system_r:httpd_t
tcontext=user_u:object_r:user_home_t tclass=dir
```

This shows the message parts and an explanation of what the part means:

avc: denied Message Explained

Jan 14 19:10:04

Timestamp on the audit message.

hostname

The hostname of the system.

kernel: audit(1105758604.519:420):

This is the kernel audit log message pointer. The timestamp consists of a long number, which is the unformatted current time, and a short number, which is the milliseconds, that is, `<current_time> . <milliseconds_past_current_time>`. The third number is the serial number, which helps in stitching together the full audit trail from multiple messages. Multiple messages for the same event occur when full audit logging is enabled using an audit daemon, which logs various kernel events.

avc: denied

The operation was denied. A few operations have `auditallow` set so they generate granted messages instead.

getattr

What was denied or granted. The brackets contain the actual permission that was attempted.

for pid=5962

The process ID of the application that is the source of the operation.

exe=/usr/sbin/httpd

The application being denied.

path=/home/auser/public.html

The path to the target file or directory the operation was attempted on.

dev=hdb2

The device node that holds the file system. The object of the denied operation lives in this file system.

ino=921135

The inode number of the target file or directory.

scontext=root:system_r:httpd_t

The security context of the source, that is, the process being denied access.

tcontext=user_u:object_r:user_home_t

The security context of the target, that is, the file or directory that is denied.

tclass=dir

The object class of the target, indicating that it was the directory /home/auser/public.html/ that was being blocked.

A.5 Policy Macros

Macros are used throughout programming, as they provide reusable pieces of code that you can call one time and have explode into many meaningful lines. SELinux uses the m4 macro language for writing reusable policy rules. This makes policy writing and management easier. In using macros, policy writers gain flexibility, modularity, shared quality control, and central management for complex pieces of policy. Macros do not exist in the policy.conf file, as that file represents the exploded macro policy code. It is possible to work backward in finding where a particular policy.conf entry exists. If a daemon has a rule that you cannot find in the associated TE file at `$SELINUX_SRC/domains/program/<foo>.te`, then it is likely to be found in the macros. This section first explains the syntax and usage of a macro, then discusses the analysis method in more detail. This usage example shows the first few lines from the Apache HTTP macro file, `$SELINUX_SRC/macros/program/apache_macros.te`:

```
define(apache_domain,
#This type is for webpages
#
type httpd_$1_content_t, file_type, homedirfile, httpdcontent, sysadmfile;
ifelse($1, sys, typealias httpd_sys_content_t alias httpd_sysadm_content_t;)
# This type is used for .htaccess files
#
type httpd_$1_htaccess_t, file_type, sysadmfile;
...
```

The `define(apache_domain,` is the beginning of the macro definition. Inside the definition, the `$1` represents the parameter passed to the macro. Look in `$SELINUX_SRC/domains/program/apache.te`, which has the following invocation: `apache_domain(sys)` This single line then generates a large set of types and rules, substituting the passed parameter `sys` for every `$1`:

```
type httpd_$1_htaccess_t, file_type, sysadmfile; -> type httpd_sys_htaccess_t, file_type,  
sysadmfile; type httpd_$1_script_exec_t, file_type, sysadmfile -> type httpd_sys_script_exec_t,  
file_type, sysadmfile role system_r types httpd_$1_script_t; -> role system_r types  
httpd_sys_script_t;
```

Appendix B

Installation steps

B.1 SELinux

SELinux is installed in almost all flavors of Linux. If not already installed, installation source of linux flavor provides a file named `selinux-policy-3.7.19-195.rpm`. This is a rpm file which can be installed by the following command:

```
rpm -ivh selinux-policy-3.7.19-195.rpm
```

if SELinux is installed but updation is required then following command can be used,

```
rpm -Uvh selinux-policy-3.7.19-195.rpm
```

If yum repository is configured, we can use the following command to install/update SELinux,

```
yum install selinux-policy
```

B.2 Python

Python is also installed as default application in almost all linux flavors. If not already installed, rpm can be installed from installation source using following command:

```
rpm -ivh python-2.6.6.rpm
```

if python is installed but updation is required then following command can be used,

```
rpm -Uvh python-2.6.6.rpm
```

If yum repository is configured, we can use the following command to install/update Python,

```
yum install python
```

Appendix C

Contents of files

C.1 Client Program

```
#!/bin/env python
import xmlrpclib
import subprocess
import socket
import time
import os
import pickle
import random

#Set the following Parameters, for client side
nodename='node1'
HASH='0d476526af2c9bbb100841cb5ef8dbd58119bb78b8f9f8af18eab4459be2cd9d'
#####

parentfileexists=0

while 1: #To get the list of hosts repeatedly if no host is able to provide service
#If host.dictionary exists and entry of client exists, then contact that server only
if(os.path.isfile('/etc/diss/serveraddress')):
a=open('/etc/diss/serveraddress')
parentfileexists=1
else:
#Fetch the list of hosts which provides the service
print 'Fetching the host list'
runsh1 = subprocess.Popen( ["-c", "avahi-browse -arpt | grep \"=\"\" |grep \"distributed\" |
```

```

grep -v '/sbin/ifconfig | grep "inet addr" | grep -v "127.0.0.1" |
cut -d\:\' -f2 | cut -d\' \' -f1\' | cut -d \' ; \' -f8 | sort | uniq > /etc/diss/serveraddresses.temp" ],
    stdin= subprocess.PIPE, shell=True )

time.sleep(5)
a=random.choice(open('/etc/diss/serveraddresses.temp').readlines())
# For Each address in avahi-browse command output
for line in a:
    value=0 #set this value to check if overwritten by exception
    print 'trying http://'+line.rstrip()+':1986'
    #Connect to Server
    s = xmlrpclib.ServerProxy('http://'+line.rstrip()+':1986')
    try: #check if error is raised
        print 'Trying to register client'
        s.register_client(line.rstrip(),HASH)
        #s.checkhash(HASH,line.rstrip())
    except socket.error, (value,message):
        print value
    if(parentfileexists==1): #If server address exists then contact parent of parent
        hostdict = open("/etc/diss/host.dictionary","r")
        x=pickle.load(hostdict) #assign the dictionary to x
        hostdict.close()      #close the file

    for host in x:
        if(line.rstrip() in x[host]): #find the parent of parent
            print host
            parentofparent=host #assign the parent of parent to variable
            value=0 #set this value to check if overwritten by exception
            print 'trying http://'+parentofparent.rstrip()+':1986'
            #Connect to Parentofparent
            s = xmlrpclib.ServerProxy('http://'+parentofparent.rstrip()+':1986')
        else:
            print 'parent file does not exists'
            #Check the exception variable and Register Client
            if(value==0 and s.register_client(line.rstrip(),HASH)):

                print 'communication established'

    while 1: #stay in communication with the given host

        a=subprocess.call(["-c","echo "+line.rstrip()+" > /etc/diss/serveraddress "],
            stdin=subprocess.PIPE, shell=True)

        try: #if exception occurs in given host
            print 'fetching hash file'
            #get the hash file from server to check for local files are old or not
            with open("/var/diss/selinux/SELINUX.HASH","wb") as handle:

```

```
handle.write(s.get_hash_files(HASH).data)
except socket.error:
break

with open("/etc/diss/host.dictionary","wb") as handle:
    handle.write(s.get_host_dict(HASH).data)

print 'checking hash of local files'

#Check the shasum of with the existing files
runsh2 = subprocess.Popen( ["-c","sha256sum --quiet -c /var/diss/selinux/SELINUX.HASH |
cut -f1 -d\" :\ " > /var/diss/selinux/files.to.fetch"],
stdin= subprocess.PIPE, shell=True )
time.sleep(5)
f=open('/var/diss/selinux/files.to.fetch')
a=subprocess.call(['touch','/var/diss/selinux/.lock'])
print 'lock file created'

#If the shasum does not match of the existing file
for file in f:
print file.rstrip()
#Get the new file from server, whose shasum is not matching
with open(file.rstrip(),"wb") as handle:
    handle.write(s.get_file(file.rstrip(),HASH).data)
modules=subprocess.Popen(["-c","cut -d\" /\ " -f6 /var/diss/selinux/files.to.fetch |
cut -d\" .\" -f1 > /var/diss/selinux/modules.to.update"],
stdin=subprocess.PIPE,shell=True)
time.sleep(5)
f=open('/var/diss/selinux/modules.to.update')
for file in f:
print 'Applying file'+file.rstrip()
a=subprocess.call(["-c","/etc/diss/selinux.sh @"+"nodename+" "+"file.rstrip()],
stdin=subprocess.PIPE, shell=True)
print 'files downloaded: now releasing lock'
a=subprocess.call(['rm','/var/diss/selinux/.lock'])
print 'now going to sleep'
time.sleep(60) #sleep for given seconds for normal operation

#If Shared Hash does not match, break and exit
else:
print 'Registration of client Failed'
#Sleep for 10 seconds if address file is now empty
time.sleep(10)
```

C.2 Server Program

```
from SimpleXMLRPCServer import SimpleXMLRPCServer
from SimpleXMLRPCServer import SimpleXMLRPCRequestHandler
import xmlrpclib
import time
import os #to check file exist or not
import pickle #to save dictionary
import subprocess

class MyXMLRPCServer(SimpleXMLRPCServer):
    def process_request(self,request,client_address):
        self.client_address=client_address
        return SimpleXMLRPCServer.process_request(self,request,client_address)

server=MyXMLRPCServer(('0.0.0.0',1986))

server.register_introspection_functions()

register_service = subprocess.Popen( ["-c","cp -f /etc/diss/disslinux.service /etc/avahi/services/"],
stdin= subprocess.PIPE, shell=True )
HASH='0d476526af2c9bbb100841cb5ef8dbd58119bb78b8f9f8af18eab4459be2cd9d'

#Register New Clients
def register_client(serveraddress,CHASH):

    if(CHASH==HASH): #check hash

        try: #try to open the hast.dictionary file
            hostdict = open("/etc/diss/host.dictionary","r")
            x=pickle.load(hostdict) #assign the dictionary to x
            hostdict.close() #close the file

        if(server.client_address[0] in x[serveraddress]):
            #if x has variables set return true
                print 'Server Address :'+serveraddress
            print 'client Address :'+server.client_address[0]
            print 'Client Already Registered OK'
            return 1

        elif(len(x[serveraddress])<3): #count no of clients in server address
            print 'No of clients in server = '+len(x[serveraddress])
            x[serveraddress].append(server.client_address)
            return 1
```

```
else: # return error if no of clients exceeded
print "No of clients exceeded in server array"
return 0

except KeyError: #in case of key error add new entry
hostdict.close()
print 'Client not registered adding new entry'
hostdict = open("/etc/diss/host.dictionary","w")#open file in write mode
x[serveraddress]=[server.client_address[0]] #get the variables set
    print 'Adding Server Address :'+serveraddress
print 'Adding client Address :'+server.client_address[0]
pickle.dump(x,hostdict) #dump the value to file
hostdict.close() #close the file
print "Host array entry added to file"
return 1

except IOError: #in case of IOError return false
print "error opening file"
return 0

else: #if wrong hash return false
print "Wrong HASH"
return 0

server.register_function(register_client,'register_client')

#hash file transfer class
def get_hash_files(CHASH):
if CHASH==HASH:
while(os.path.isfile('.lock')): #Check for lock file
time.sleep(10)
#Check the shasum of with the existing files
print 'Creating HASH FILE'
    runsh2 = subprocess.Popen( ["-c","sha256sum /var/diss/selinux/data/* >
/var/diss/selinux/SELINUX.HASH"],
stdin= subprocess.PIPE, shell=True )
time.sleep(5)
print 'Transferring HASH FILE'
with open("/var/diss/selinux/SELINUX.HASH","rb") as handle:
return xmlrpclib.Binary(handle.read())

server.register_function(get_hash_files,'get_hash_files')

def get_host_dict(CHASH):
if CHASH==HASH:
```

```
print 'Transferring DICTIONARY FILE'
with open("/etc/diss/host.dictionary","rb") as handle:
    return xmlrpclib.Binary(handle.read())

server.register_function(get_host_dict,'get_host_dict')

#file transfer class
def get_file(filename,CHASH):
    if CHASH==HASH:
        while(os.path.isfile('.lock')): #check for lock file
            time.sleep(10)
        print 'Transferring file :'+filename
            with open(filename,"rb") as handle:
                return xmlrpclib.Binary(handle.read())

server.register_function(get_file,'get_file')

# Run the server's main loop
server.serve_forever()
```

C.3 Avahi Service Entry File

```
<?xml version="1.0" standalone='no'?><!---nxml-!-->
<!DOCTYPE service-group SYSTEM "avahi-service.dtd">
<service-group>
  <name replace-wildcards="yes">%h</name>
  <service>
    <type>_distributed_selenium._tcp</type>
    <port>1986</port>
  </service>
</service-group>
```

C.4 SELinux Policy Updation Script

```
#!/bin/sh
# $1=nodename
# $2=modulename
cp -f /var/diss/selinux/data/$2.te.gz /var/diss/selinux/incoming/
zcat /var/diss/selinux/incoming/$2.te.gz | sed 's/'$1'//g' | grep -v "@" | grep -v "initrc_domain" >
/var/diss/selinux/incoming/$2.te
cp -f /var/diss/selinux/data/$2.mod.fc.gz /var/diss/selinux/incoming/
```

```
gunzip -f /var/diss/selinux/incoming/$2.mod.fc.gz
checkmodule -M -m -o /var/diss/selinux/incoming/$2.mod /var/diss/selinux/incoming/$2.te
semodule_package -o /var/diss/selinux/incoming/$2.pp -m /var/diss/selinux/incoming/$2.mod -f
/var/diss/selinux/incoming/$2.mod.fc
rm -f /etc/selinux/targeted/modules/semanage.trans.LOCK
semodule -u /var/diss/selinux/incoming/$2.pp
mv -f /var/diss/selinux/incoming/$2.pp /var/diss/selinux/patched
```

C.5 Host Dictionary file

Initial Host Dictionary :

```
(dp0
S'0.0.0.0'
p1
(lp2
g1
as.
```

Host dictionary after addition on node :

```
(dp0
S'172.16.10.91'
p1
(lp2
S'172.16.10.71'
p3
asS'0.0.0.0'
p4
(lp5
g4
as.
```

where 172.16.10.91 is the server and 172.16.10.71 is the client for the same

C.6 README

- host.dictionary.init file should be copied as host.dictionary
- There should not be any file named serveraddress, if exists it should not be empty, there must be an address of server
- Run the client program first to get data from parent , then the server program to serve other clients

- /var/diss contains the runtime information of the service
- /etc/diss contains the all configurations and the programs

C.7 HASH file

```

812c6e29acf56bfc4f9afb736d65d417d61283a5baf4849b4e3493b349625c67 /var/diss/selinux/data/abrt.mod.fc.gz
49a5c7b95eaae19839601aea0e3dee849f4f02c5c670acd9e4e314ccc2502a1f /var/diss/selinux/data/abrt.te.gz
7f8c9efb5917e4ef79f0cd45511b4f42482ee82b588daa962cec2422e154baee /var/diss/selinux/data/accountsd.mod.fc.gz
5ab3ef5b45b398f819e74cb32db0443baec7bce48df9cc4c63e334905a5d7781 /var/diss/selinux/data/accountsd.te.gz
bd8ff632ad599f7899c40f0b616fae68b4153656121abe9e9a93d42cdbfcb929 /var/diss/selinux/data/ada.mod.fc.gz
c6b970434a855829451d1513df0651325166caead796bd2c40abcc1553efbd9 /var/diss/selinux/data/ada.te.gz
69f1ba7f5dfaaea8e3378b37534794ccced3b6b6dddaaa1e9210f68a4fcec2f /var/diss/selinux/data/afs.mod.fc.gz
c22422fa072b237833fdd9530dc1b2433f778729434b7a046c9d34340899ccda /var/diss/selinux/data/afs.te.gz
86117265d2cf2800ff1f0c66e15e2ce98d6fa18c4859d0cd751a69f5ff2a5f2c /var/diss/selinux/data/aiccu.mod.fc.gz
f63264f5d35df6bb6c2422292453b38103657370087c19bc8474d9ff7acc53 /var/diss/selinux/data/aiccu.te.gz
9201714343dead31475a1c12905f780604a8731313e876f4e4a7b1c376258a0f /var/diss/selinux/data/aide.mod.fc.gz
4964ccb9287dbf87a0d9443b88ecff5f892e9d835fa178f4ba0177eb9ff9b2fc /var/diss/selinux/data/aide.te.gz
2d5869f6e33f117b5cea9410116bcd42f05464e964dd7a58e9dd0b7f99c6f1e8 /var/diss/selinux/data/aisexec.mod.fc.gz
3cd28780036741f26b2ca6a3d169194641e25bd785fbf023aeaa864d38d9dc47 /var/diss/selinux/data/aisexec.te.gz
4568a783ca3b89eeb399967687daa5f869a2142e2bbfe7b0d6d358b3e6973f44 /var/diss/selinux/data/all_interfaces.conf.te.gz
b9c9405733351b38599069794979849543bb483ad800c7411fcde25919ed3308 /var/diss/selinux/data/amanda.mod.fc.gz
b168b8ba2af73ee3847a008effeff6a00913d8a30465a9c9ad021d532aabc559 /var/diss/selinux/data/amanda.te.gz
80501650e42c26ab146ee27f754682db7e98b232a03b50023a6f066b37d9891e /var/diss/selinux/data/amavis.mod.fc.gz
b14445b08ce6c47741b6a8d9811aa0efa582503c4c73956f7f3feeb57e0f69f1 /var/diss/selinux/data/amavis.te.gz
0016adea3b2ac1de3843993f2176fa46d4cd6f6ca96165d56e6f6c23553fc13d /var/diss/selinux/data/amtu.mod.fc.gz
6c8d8942bbc81cb3d4c79e4fa4d1182c716be7c6f10a585c531b978f47331e25 /var/diss/selinux/data/amtu.te.gz
3a19dc81655ed81b2965c8bc14390f4300a1c037b646099ce693809d07d5bf2a /var/diss/selinux/data/antivirus.mod.fc.gz
f26f71a4a9fb663f615963a774c70899102bba1d68d123b27029750b0a00ae9b /var/diss/selinux/data/antivirus.te.gz
04435af43213a4cdec97ed64bee148750f7decbf228a1e8a31a7d945e8d84aa1 /var/diss/selinux/data/apache.mod.fc.gz
24558d5eb9260e1dfdb121f2654f8bface6bd3a64d7486ca6f347776641bdec3 /var/diss/selinux/data/apache.te.gz
6859f3d4f9a5d72a668b961b863610a79b02a51a9a3a5edbfc4d59a6594463d1 /var/diss/selinux/data/apcupsd.mod.fc.gz
6304b172e761bbcdc41f9f5a81ac51ec26fecfdd5871288ea074b21ad93662b /var/diss/selinux/data/apcupsd.te.gz
67555640343bc57e34ae3e27ad2d6fb6ca65a86f36258d6e94a6cca7b4b41997 /var/diss/selinux/data/arpwatch.mod.fc.gz
4c27be38209b4e032fa1a17ac3880c44b27aa27a3fc7b8fc8ee3fd9c865854a3 /var/diss/selinux/data/arpwatch.te.gz
ff6d9616d04f97eceb2c94475dbab9e886b61bd541b64892a9d330f074befaff /var/diss/selinux/data/asterisk.mod.fc.gz
189938f0a36f00f35009f6876b86c07d374120581cda96d20d013cea939de546 /var/diss/selinux/data/asterisk.te.gz
71f113f4c6499d1c05a057482620896b77e82cfa905527c8e40bc06da419df98 /var/diss/selinux/data/audioentropy.mod.fc.gz
546e65a0e4c8c73645b45dd77ff55e70d16a3a57e940bf0bba5e80c03f06e49 /var/diss/selinux/data/audioentropy.te.gz
65588721b5ef3c7b04d21fc17b4f89b4530b5babd8d26eb3d0fc2a1c247cd720 /var/diss/selinux/data/automount.mod.fc.gz
6ab712bf961ba33d2bc468340840d7ae277da798e8699ddaa2366442bbce28fe /var/diss/selinux/data/automount.te.gz
6c1930642ef4140f459780c9f87089aa83f63b5833b0080989e66b5c375605fa /var/diss/selinux/data/avahi.mod.fc.gz
57a552a7efd33564187edb6144f6b045ffec8294f72eeb71d0c9fd3768a1a /var/diss/selinux/data/avahi.te.gz
6fd498c41cdaa2797d07c1e3d653e87c2633c86de02b1731a8a0a0cf56dc5758 /var/diss/selinux/data/awstats.mod.fc.gz

```

C.7 HASH file

```
f7167a09d0cef161fad69cd8c0c2404be0642ee0834490079b7be9dcc390cf57 /var/diss/selinux/data/awstats.te.gz
58f90ad4a109ab7ae6d9e9f100eb7abb713513c0dc8c687fd067243c19f4295e /var/diss/selinux/data/base.fc.te.gz
321db191dd951edf9f910514ec1e22f8789863e9f30b9a15002b3d264b8a3c25 /var/diss/selinux/data/bcfg2.mod.fc.gz
1338a9ae478d2be968671fd0e231dc31f133d8874b2b5d1399cfc4e40fada36e /var/diss/selinux/data/bcfg2.te.gz
3639d231206fadf281a68abf674b212b9033b7fca0e12b5853de73e428d70776 /var/diss/selinux/data/bind.mod.fc.gz
5f4cad2d3fcda569016a0d018323338cf157a4c7accd1401dbf8179612ea5533 /var/diss/selinux/data/bind.te.gz
789a0aaffb68deda344b72322daa87bb37c01acb402b128ec85b64ef9bfbecff /var/diss/selinux/data/bitlbee.mod.fc.gz
75c5049aa942eba5bccff34b4200ab5291677c23311493f4471243009ad6bb20 /var/diss/selinux/data/bitlbee.te.gz
65d6671db513afb6808f7909d258d246362ab6c7eae487beab28772e8d2231bc /var/diss/selinux/data/bluetooth.mod.fc.gz
9baa12d6434f96ced469e767749d9ee4df75fe1cfb31e30ed038905082755409 /var/diss/selinux/data/bluetooth.te.gz
11cb3f3826ad89bc17f3a998cc35f672598bcd9e033125f0ea02413559905c2 /var/diss/selinux/data/boinc.mod.fc.gz
fabaa673f15436e14943fb162aac283782c78775dcafcd5f114931b88af90645 /var/diss/selinux/data/boinc.te.gz
.....
```