PLEASE: Policy Language for Easy Administration of SELinux

A Thesis Presented
by

David Patrick Quigley

to

The Graduate School

in Partial Fulfillment of the

Requirements

for the Degree of

Master of Science

in

Computer Science

Stony Brook University

Technical Report FSL-07-02
May 2007

Stony Brook University

The Graduate School

David Patrick Quigley

We, the thesis committee for the above candidate for the

Master of Science degree, hereby recommend

acceptance of this thesis.

Dr. Erez Zadok, Thesis Advisor
Professor Computer Science

Dr. Scott D. Stoller, Chairperson of Defense
Professor Computer Science

Dr. Robert Johnson
Professor Computer Science

This thesis is accepted by the Graduate School

Lawrence Martin
Dean of the Graduate School

Abstract of the Thesis

PLEASE: Policy Language for Easy Administration of SELinux

by

David Patrick Quigley

Master of Science

in

Computer Science

Stony Brook University

May 2007

With the growing importance of security, alternate access control methods have become commonplace. The emergence of systems such as SELinux has provided a new means to restrict access beyond Linux's traditional capability-based system. Unfortunately, writing a policy for applications in SELinux is difficult, even for experienced policy developers. Previous attempts to simplify policy development, such as Tresys's Cross Domain Framework, rely on an existing policy and govern interactions between policies. Other attempts, such as Virgil, over-simplify policy development, restricting the developer too much.

To reduce the complexity of the policy development process, we developed PLEASE, a high-level language for writing SELinux policies. PLEASE is designed to integrate into the SELinux reference policy by making use of the interfaces and types already present, allowing for sections of the reference policy to be rewritten in it. We provide the developer with facilities to specify SELinux policy statements directly from PLEASE, to be analogous with the relationship between C and assembly. This preserves the power and flexibility of low-level policy statements while still allowing the developer to use our higher-level abstractions. We experimented with PLEASE and found that its abstractions yield considerable savings in policy size, ranging from a factor of 3 to 8 reduction in common cases to as much as 24 in certain special cases.

To my mom and dad.
Without you I would not be where I am today.

Contents

Acknowledgments

I would like to express my utmost gratitude to the following people...

My advisor, Dr. Erez Zadok, for his ideas, insight, and faith in me to allow me the freedom to explore my own research. Kim Albrecht for all the time, effort and dedication she put into our project to make it a success. All my lab mates in the FSL who made my time here enjoyable and "very educational," particularly Avishay who is giving me last minute corrections as I write this. Radu Sion for providing comments on this thesis early on. Dr. Scott Stoller for providing valuable feedback on this thesis, chairing my committee, and being a valuable advisor during my time as an SBCS officer. Dr. Rob Johnson for providing crucial advice concerning the structure of the paper and for asking the hard questions during my defense. The National Science Foundation whose Scholarship for Service program provided me with the opportunity to attend graduate school. Steve Smalley and Pete Loscocco at the National Security Agency for giving me the chance to learn first hand about SELinux. Finally, my friends Chris, Lisa, and Katie for being patient and understanding as I disappeared for two years to accomplish this.

This work was made possible partially thanks to NSF awards CCR-0310493 (CyberTrust) and DUE-0417103 (Scholarship for Service in Information Assurance).

Chapter 1

Introduction

Networked computer systems are now a part of everyday life. Because of this, standard access control methods have been shown to be increasingly ineffective. This has created an effort to provide stronger methods of access control to modern operating systems. SELinux is one such form of mandatory access control [7]. SELinux is an attempt to regulate access to important objects in both the kernel and user space. However, SELinux does have its shortcomings.

SELinux is a mandatory access control mechanism for the Linux kernel. There are several distinctions between the discretionary access control (DAC) models of traditional Unix systems and the mandatory access controls (MAC) in SELinux. In a DAC system, the owner of a file determines who has access to it. In a MAC system, the policy administrator specifies a policy that determines the access to resources, such as files and directories.

A second distinction between these systems is that in Unix's DAC implementation, the granularity of access control is limited. Only the access rights of the owner, the group of the file, and individual users can be specified. Also, there is no distinction between the actual types of files. According to the Unix paradigm, a file does not accurately reflect the true nature of a system. However, there are actually several types of files, such as character files, block files, regular files, and symlinks. Unix treats sockets as files, even though the semantics of sockets and files are different. However, in MAC systems, and particularly in SELinux, the distinctions among file types are important. A policy developer can make the distinction between different types of files, and thus provide different access rights to each accordingly. Because SELinux uses an object labeling mechanism, several different applications can each have separate access rights to a particular resource. Unix can provide different access rights to a file in only three different ways (user, group, other) without the use of Access Control Lists (ACLs).

Another major distinction between MAC and DAC systems is that in DAC, there are only two classes of users: unprivileged users and super users. In a MAC system, there is no user with complete control over the system. By giving system administrators the ability to create users with varying levels of access, instead of complete control, they can create separate administrative users for individual tasks in the system. Since SELinux's MAC implementation is a white list, meaning it only allows actions that are explicitly specified, an application cannot be tricked into performing actions on an object that it is not supposed to have access to. In traditional systems, however, if a process ran as root and was exploited, it could be used to take control and manipulate any file

Number of Types and Attributes:	
Types:	1,483
Attributes	147
Number of Type Enforcement Rules:	
allow	75,678
neverallow	0
auditallow	27
dontaudit	4,687
type_transition	1,348
RBAC Elements and Rules:	
Roles	6
Users	3
allow	5
Number of Other Context Statements:	
Portcons	260
NetifCons	0
Nodecons	8

Table 1.1: Relevant statistics derived from the apol policy summary of the targeted policy provided with Fedora Core 6 [2]

in the system. Since SELinux only allows what the policy specifies, even if an application is compromised, it cannot access a file for reading or writing if it is not given permission to do this. This is one of DAC's main flaws, since even an unprivileged user's application can be tricked into leaking information or destroying the files that it actually has access to.

1.1 Motivation

Even though SELinux provides strong access mechanisms and better system security, it is not as widely used as it could be. This is due to several drawbacks currently associated with SELinux. Writing an SELinux policy is difficult and time-consuming. Since SELinux is based on an object-labeling mechanism, a policy developer must first have a working knowledge of what types exist in the system. Tresys Systems maintains a set of system and application policies that is used in several Linux distributions, called the SELinux Reference Policy [16]. Although the reference policy is a good foundation to build upon, it is a very complex and long set of policies, as can be seen in Table 1.1.

Despite an extensive effort to expand the reference policy, it is far from being comprehensive. There are many applications from major distributions, such as RedHat and Debian, which make it impractical to run an SELinux system effectively under the `strict` policy. There is a `targeted` policy which covers network daemons and other sensitive services, but this only provides a subset of needed system security. Because of this, many people choose to disable SELinux rather than deal with the complications of not having a comprehensive policy. This is not the fault of the distributions, but rather a symptom of the difficulty of policy development. A distribution such as Debian contains 18,733 packages [3], making it impractical for distribution maintainers to write and maintain policy for all of them. This responsibility should instead fall to the application developers.

One reason for the lack of security policies is that most application developers do not understand all of the mechanisms used by SELinux. Application developers, however, are the best people to write these policies, since they know what their application should and should not be allowed to do. However, many application developers are not actually trained in the field of security. Even if they have some knowledge of security, they may have other priorities while developing their applications. This has made it difficult for administrators to easily specialize policy for their environment, and handle third party applications.

To address some of the issues present in existing SELinux technologies, we have developed PLEASE. PLEASE is an object-oriented, high-level language for writing SELinux policies that strives to reduce the complexity of the policy development process. This is accomplished by providing language-level features for resource and permission abstraction. In addition to these abstractions, PLEASE is designed in a manner such that a developer can ignore the existence of types. We also provide developers with a mechanism to use custom types, along with features that allow the developer to attach interfaces to an application in a structured manner. Finally, PLEASE provides the developer with facilities to specify SELinux policy statements directly, to be analogous with writing a block of assembly within C code. This allows the developer to use the higher-level abstractions of PLEASE while still exploiting the power and flexibility of the existing low-level policy statements. Brooks has speculated that the use of such high-level languages increases productivity by a factor of five, in addition to significantly improving reliability and simplicity [1].

Over the course of the development of PLEASE, we studied five use cases which demonstrate the potential for policy size reduction. Interfaces that are defined to aid in policy development throughout the reference policy cause some of the policy statements to be replicated. PLEASE has been able to reduce policy size by a factor of 3 to 8 in common cases, and reduced policy size by a factor of 24 in several specific cases. Since many of the defined interfaces are built by compounding other interfaces, features such as resource and permission inheritance in PLEASE yield substantial savings.

1.2 Thesis Organization

The rest of this thesis is organized as follows. In Chapter 2 we give a brief overview of the internals of SELinux and the base policy language. In Chapter 3 we outline the constructs and semantics of PLEASE and how they relate to base SELinux policy concepts. In Chapter 4 we briefly discuss the technologies and work put into the PLEASE compiler. In Chapter 5 we look at two examples of PLEASE policy. The first example explores an existing reference policy abstraction that has been rewritten using PLEASE. The second example is a sample of converting some of the M4 based

SELinux macros into a set of structured and efficient resource classes. In Chapter 6 we provide additional information about other works in the field of SELinux and how PLEASE addresses their deficiencies. In Chapter 7 we look at additional constructs and concepts that we hope to develop in the future. We conclude in Chapter 8. Appendix A contains the policy source mentioned in Chapter 5. Finally, Appendix B contains additional language information about PLEASE, including lexical issues and the grammar.

Chapter 2

SELinux Overview

In this section, we will discuss some basic concepts of SELinux, such as how the Linux kernel and the SELinux security server make access decisions and how the information for the security server is presented. We will then discuss how PLEASE can be integrated into the whole system. It is important to note that the actual implementation of SELinux is more complex than what is discussed in this section. Since the goal of this section is to provide the reader with only a working background of SELinux, some irrelevant concepts have been abstracted or omitted. We will be skipping over the specifics of Identity-Based Access Control (IBAC), Role-Based Access Control (RBAC), and Multi-Level Security (MLS), since they will be addressed in future work.

SELinux is an implementation of the FLASK architecture for Linux [14]. A main component of the original FLASK architecture is the separation of the policy from the actual enforcement of the policy. This is why there are three components in the Linux kernel for SELinux: the Security Server, the Access Vector Cache, and the Object Manager. The security server is responsible for making the actual access decisions for the system. The Access Vector Cache (AVC) is an optimization of the security server such that each access check does not require that the access rights be recalculated. The object manager is responsible for enforcing the decisions made by the security server. The Linux kernel provides a framework in which the object manager and other security mechanisms can be implemented. This is called the Linux Security Modules (LSM) Framework. The LSM Framework provides a global structure that contains a series of function pointers, similar to the Virtual File System (VFS) operations vectors. An LSM developer implements the functions defined by the framework in a manner consistent with the interfaces and the access model that they are trying to provide. A series of hooks are placed in the kernel code to make calls to the appropriate functions in the module. Normally, these calls are placed to either mimic the existing DAC checks or to provide further restrictions. In an increasingly larger number of places in the Linux kernel, these hooks are placed to ensure that security modules can provide finer-grained security access [7].

In SELinux's security module, the calls to the module first check the AVC to see if the access decisions have already been made. When the kernel checks this, it gathers security identifiers (SIDs). SIDs are internal representations of the policy language types for both the source and the target of the action. With these two SIDs and the object class it wishes to access, the kernel looks up the permissions in a table. The entry in the table is a bit mask with bits set for the allowed permissions. If an entry is found in the table, the desired access is checked against the bit mask. If such an entry exists and it matches the bit mask, the access is granted.

In the event of a cache miss in the AVC, the request is sent to the security server. Through a function called `security_compute_av`, the security server performs two checks before it allows access. First, it creates a mask representing the object permissions allowed, in accordance with the Type Enforcement (TE) allow rules in the policy. Second, it removes permissions that are disallowed by any constraints [8].

SIDs are the internal representations of types. Types are the foundation of the Type Enforcement (TE) mechanism that SELinux provides. However, the type is only one component of an object label. The SELinux model itself is a combination of several existing security models. The security context holds information to provide for Identity Based Access Control (IBAC), Role-Based Access Control (RBAC), Type Enforcement (TE), and Multi-Level Security (MLS). Although all of these are present, they do not necessarily have to be used in every system.

SELinux's model of access control inherently denies all actions by default, unless the actions are specified as allowed. Rules of this type are referred to as Access Vector (AV) rules, and are formatted as follows:

```
<AVRule> <source> <target> : <class> {<permissions>};
```

Internally, SELinux indexes access permissions by the source, target, and class tokens in the rule. The permissions specify the bit mask generated for this triplet. The AVRule token of the access rule can be one of four keywords. The `allow` keyword specifies that the permission is to be allowed. The `auditallow` keyword specifies that successful acquisition of the permission is to be audited. This is to provide auditing support for privileged operations, such as changing passwords. The `dontaudit` keyword allows SELinux to omit certain access failures from the auditing system. These two keywords are in place to modify SELinux's default auditing behavior of only auditing failed accesses. The `neverallow` keyword is used to mark assertions to be checked by the policy compiler, and does not map to the internal kernel mechanism.

The source and target tokens of the above AV rules can be one of several possibilities. When a type is defined in SELinux through the `type` keyword, aliases for the type may be specified, along with the attributes that it possesses. Aliases are mainly used for backward compatibility with existing policy. For example, when a type in the main policy changes, there may still be policy modules that refer to the old type. In order to give developers time to migrate their modules from the old type to the new one, they may alias the original type to this new type [8]. PLEASE addresses this differently: by removing the need to define actual types in PLEASE policy, the old type is automatically replaced when the compiler regenerates the module.

Aliases provide a means to access a single type through many identifiers, but SELinux also provides the inverse. When declaring a type, a series of attributes that the type possesses may also be declared. These attributes may be used in any of the type enforcement rules. Because of this, a single identifier can be used to grant access on all of the types that possess that attribute. PLEASE addresses this concept through its attribute construct. This is discussed in Section 3.5.

Aliases and attributes are declared when the type is declared. After a type has been declared, additional aliases may be added with the `typealias` keyword, and additional attributes may be added with the `typeattribute` keyword. The arguments for both are the type to be aliased or attributed to, and a list of the respective types. The purpose of these keywords is to allow a policy module to apply attributes and aliases to types that are defined in the base policy. This allows a developer to make minor changes to an application policy, without modifying and recompiling the entire reference policy. PLEASE is designed to define entire application policies. Because of this, we do not address policy modules and their associated keywords; they are easy to address using the base policy language once the policy is created.

The `runcon` utility in the SELinux core utilities allows applications to be executed in arbitrary domains as long as they have the appropriate permissions. If this were the only method for executing a program in its appropriate application domain , it would be tedious and not user-friendly. However, the `type_transition` keyword accounts for this. This keyword describes the default types for new subjects and objects if no explicit type was specified. They do not provide access themselves, so they must be coupled with appropriate access rules. There are multiple possible sets of access rules that may be appropriate which usually contains a minimal core set. The main use of this keyword in the reference policy is to specify the domain to transition to when a user to executes an application. In this case the minimal core set is `execute`, `transition`, and `entrypoint`. PLEASE addresses this functionality through the `application` block abstraction, which is described in Section 3.3. There are also alternate uses of this keyword to label transient objects, which we discuss in Section 7.

There is a concept in SELinux called object tranquility which states that once an object is created in the system, its label should not change. When the labels on a system are changed, the system needs to relabel itself based on the current policy. The reason for this is that once labels in a system are changed, the policy cannot be verified to be correct. There are two methods for doing this. The first method is to order the system to relabel on boot. This can be a time consuming process but is necessary when the system policy is changed. The second method involves a program called `restorecon`. This application will query the kernel for the necessary information to determine what it believes to be the label for the object.

Chapter 3

PLEASE Language

In this section, we discuss the key concepts of PLEASE and the language-level features that make policy development easier. Most elements in this section define both resource classes and application policy. The syntax and semantics for assigning values to members of resource classes is similar to that used in defining the resource classes themselves. Resource classes have additional modifiers to particular parts of the language primitives, which are not valid in the application section of the policy.

3.1 Overview

The main design decision behind PLEASE is the develop a language that is modular and conceptually easy to understand. An administrator understands a system in terms of components. To illustrate this consider a program named Crunch. Crunch is a simple program that takes files in a particular directory and then hashes them and emits the results to its own ad hoc log file. This is a practical example since this behavior is actually a subset of the functionality provided by integrity checking tools such as Tripwire [17]. Crunch relies on three items to perform its task. The first of these is a configuration file the second is the log file; and the third is the shared library containing the required cryptographic functionality. A possible application for a program such as this is to check for changes in key system files. Because of this we want to restrict Crunch to certain actions. We do not want Crunch to be able to write to its configuration file since an attacker could remove a particular file from the list of files to be checked and avoid detection. The second action we want to prevent is the ability to remove an old log file and write a new one in its place. To prevent this we want to grant append access to the log but not read or write since that will allow an attacker the ability to recreate the log with doctored information. We will use Crunch as much as possible in the explanation of the PLEASE language constructs. In all of the examples PLEASE language keywords are highlighted in bold text.

Since the underlying idea of PLEASE is its modular and hierarchical organization, we decided to use an Object-Oriented paradigm. The reason policy development is well suited to this model is that there already exist many components in the reference policy which attempt to represent objects. In addition to this reference policy interfaces provide in an ad hoc manner the concept of member functions for these objects. By creating a language that takes these two loosely implemented concepts and provides a well-structured mechanism to represent them, we can reduce the conceptual complexity of policy development.

3.2 Resource Classes

The written description of Crunch yields important insight into how an application policy can be developed. We clearly see three modular units that are needed by the application: the log, the configuration file, and shared libraries. The base SELinux language does not have the concept of an object, so there have been attempts to create these abstractions through attributes and interfaces. An example of this is the `device_node` attribute in the SELinux reference policy. In an attempt to encapsulate the idea of a device node, an attribute named `device_node` has a separately defined set of interfaces to this “object” [16].

PLEASE solves this problem by providing Resource Classes, a structured mechanism that allows developers to design abstracted resources for use in application policies. By an abstract resource we mean that the object describes a desired functionality. This functionality is then turned into concrete access statements when the object is used. These objects are plugged into a policy in a modular manner without concern for underlying types. Whereas resource classes are used to define higher-level abstractions, their control is still limited to the object classes provided by SELinux.

PLEASE is designed to allow the creation of resource objects without any language modifications. This is possible with several abstractions that are commonly used at the lowest levels of the SELinux policy. These primitives are used as building blocks to construct resource classes. Resource classes are then used as building blocks themselves, either to develop additional compound resource classes, or as blocks to define application policies.

Resource classes are perceived by the developer as one unit, with the primitives actually representing the three types of files present in the SELinux reference policy. As we outline the PLEASE language primitives, we relate them to the contents and constructs of the `*.te`, `*.if`, and `*.fc` files of the reference policy. Each of these types of files contain a specific set of information for the particular application policy. The `*.te` files contain the Access Vector (AV) rules for the policy, interface invocations, and macro usage. The `*.if` files contain interfaces which grant access to components of the application policy. Finally, the `*.fc` files contain path globs and file contexts that are used to relabel the file system. These paths can be used when the entire system is relabeled, or when `restorecon` is called on a particular file.

```
resource File {
}
```

Figure 3.1: Declaration of a resource class

The declaration of a resource class is specified using the `resource` keyword. In Figure 3.1, we see the declaration of an empty resource class named File. We use this example and build upon it for the remainder of the resource class examples, and as an instance in the application policy examples. The full definition of the File resource class can be seen in Appendix A.3.

3.3 Application Policies

The second part of PLEASE is the use of resource classes in application policies. The `Application` keyword declares the top-level block that defines an application policy. Application declarations contain instances of primitives, resource classes, and interfaces. When an application block is defined, the name of the application defines an entry point type and the main application type. Application resources inherit their type from the main application block, under the conditions outlined in Sections 3.4 through 3.9. An example of a policy definition can be seen in Figure 3.15 where we use the File resource class to define the behavior of Crunch.

3.4 Modifiers

Resource class definitions and object instantiations use three modifiers: `default`, `isolated`, and `optional`. The `default` and `isolated` modifiers are used for only certain purposes. However, `optional` is applicable in almost all cases.

`default:` The first of the special-purpose keywords is `default`. This modifier serves as a marker to denote the main label or attribute for a resource object. This keyword is primarily used when a developer defines a resource class with multiple `label` primitives. Since we do not use types explicitly in access rules, we infer the type for the rule based on the resource class instances it uses. Because of this, the developer must specify one of the labels in the resource class to be the default. Additionally, the `default` keyword specifies the default element of a class set in a class primitive declaration. The latter use of this keyword can be seen in the definition of the File object class. In Figure 3.6 a class named `file_type` is declared and its first element is declared as default. This means that in the event that `file_type` is left uninitialized, this will be its default value.

`isolated:` Currently, when an application policy contains more than one type, it is to provide different security domains. By creating multiple types, different accesses can be given to separate resources that the application is going to use. Since we are encapsulating resources in resource classes, we also need a mechanism to separate them into different domains. One way to do this is to declare a label explicitly in a resource class. However, to avoid the need for types, we mark the resource class with the `isolated` keyword. This means that instead of specifying a type, we implicitly create a type based on the identifier of the application and of the resource class instance. In Figure 3.2 we have created two isolated instances of File resources in the application Crunch. Since the name of the application is Crunch, and the identifier on the File instances are `config` and `log`, the resource has an implicit type of `crunch_config_t`, and `crunch_log_t`. As the purpose of `isolated` is to provide a domain separation abstraction while removing the need for type information, it overrides explicit definitions of types. This means that if the developer specifies the types explicitly in the instantiation of the object or as a default label in the object class definition, the types are ignored.

`optional:` Each of the primitives in PLEASE have a default value associated with them. This specifies that some of the object class components do not need to be specified in its instantiation. In Section 3.5, we state the default value for each primitive.

```
application crunch {
    isolated File config; //type for /etc/crunch.conf
    isolated File log; //type for /var/log/crunch
}
```

Figure 3.2: Instantiation of two isolated File resources in an application

3.5 Type Enforcement Keywords

In this section, we discuss all of the PLEASE keywords that provide control over the type enforcement mechanisms of SELinux. The basic SELinux concepts of attributes and labeling are expressible, as well as the more complex network labeling constructs. The network labeling portions of SELinux are currently being overhauled to comply with LSPP (Labeled Security Protection Profile) certification. This means that PLEASE will have to re-address these concepts at a later point in time. The keywords described in this section are based on a simple format, as seen in Figure 3.3.

```
<modifiers> <primitive> <identifier> <body>
```

Figure 3.3: Syntax for primitive declarations

Label: As explained at the beginning of Chapter 3, each of the keywords are used in two contexts. The first usage of the `Label` primitive is as a component in a resource class. Figure 3.4 shows the `label` primitive in the context of a resource declaration. This figure also displays the `default` and `optional` primitives mentioned in Section 3.4.

```
default label var_con {
    optional user = user_u;
    optional role = role_r;
    optional type = type_t;
};
optional label context2;
```

Figure 3.4: Declaration of two label primitives with modifier keywords

In Figure 3.4, `var_con` uses the primary label for the object. We also see the `optional` modifier used in two different ways. In the case of the label primitive, `optional` acts as a modifier to the `label` keyword, or to any of its components. The first three instances of `optional` are placed on each component of the `label` primitive, specifying any combination of user, role and type for this label. The last instance of `optional` specifies that the entire entry for `context2` need not be specified.

The use of `label` in an `application` block is almost identical to that of `context` in Figure 3.4. The only difference is that `optional` cannot be placed on the members of the `label` primitive, or on the primitive itself. In the `application` block, all components of the primitives are considered optional, allowing the developer to specify what they need in the label.

There are four rules that determine what label is associated with a resource class instance. The compiler checks these rules in the following order, until one succeeds:

1. Use the label that is explicitly defined when the resource class is instantiated.
2. Use the default value for the label, as seen in Figure 3.4.
3. If the resource is marked as `isolated`, then generate a new type based on the application identifier and the identifier of the resource class instance.
4. Use the application block's label or the label of the group it belongs to.

As both SELinux and PLEASE do not extensively use the `role` and `user` members of the label, we set the default to the values commonly used in the reference policy. In the case of `role` and `user` the default values are `staff_r` and `user_u`, respectively. We allow the developer to easily specify complete labels on the static resources, as described in Section 3.8. This also creates a base for future extensions to PLEASE that would handle RBAC.

`Group:` We allow the developer to create resource classes to encapsulate ideas, making policy development conceptually easier. However, even though the ideas the developer is trying to represent may be disjoint, the security domain may be the same. The `group` keyword allows the developer to group a set of resource objects together under a common identifier, assigning the group a shared label. Since the label is now associated with the group instead of a particular resource, a conflict is possible. Conflicts are resolved by allowing only one element of the group to define the group's label.

```
File contents {
    member = group1;
    file_type = Regular;
    file_list {
        /* Label our source directory and its files */
        /opt/crunch
        /opt/crunch/*
    }
};
Socket output {
    member = group1;
    context = {
        type = crunch_output_t;
    }
};
```

Figure 3.5: Declaration of two resources classes that share a group.

The example in Figure 3.5 shows two resource classes that share the `group` "group1." There can only be one `group` declaration in a resource class, and this variable can contain only one element. The `group` keyword has an implicit optional modifier applied to it since the default value of `empty` is acceptable in most cases. Because two elements share the same group, only one may define the type of the group. In this example the explicit type declared for Socket is the type value for the group.

Class: In certain circumstances, multiple types of the same resource are needed. However, the developer has to declare the same object several times just to change the underlying object class that the resource references. The `class` keyword provides a method for mapping an identifier to a string, which is needed to define abstractions for several object classes. The `class` keyword can also be used to declare class elements without associated strings. This allows the developer to conditionally include a policy, based on the class of the object. One use of this is when defining an encapsulation for a protocol's packet. The default label or actions within a permission can be specified, based on the class of the object.

```
class file_type
{
    default Regular = "file";
    Symlink = "lnk_file";
    FIFO = "fifo_file";
    BlockFile = "blk_file";
    CharFile = "chr_file";
    SockFile = "sock_file";
}
```

Figure 3.6: `file_type` class used to abstract file object classes

Using a class in an application policy is as simple as assigning one of the values to the member. Since a class is used as additional meta data to generate a policy, it always needs a valid default value. Because of this, the `optional` keyword is not allowed on a `class` definition. The developer can specify one of the keys in the class as `default`. In the example in Figure 3.6, `Regular` is marked as the default class for this resource object. The utility of this construct can be seen in Appendix A.3 in the File resource class example. In the permission for `getattr`, the `file_type` identifier is used in place of an SELinux object class. This means that the string corresponding to the current value of `file_type` will be substituted in for the object class, allowing the six object classes to be represented with one identifier.

Attribute: Although the `group` keyword allows a set of objects to be grouped together for type purposes, PLEASE still needs to be able to associate permissions with traditional SELinux attributes. The purpose of the attributes in the original language is to grant access rules on a large number of types at once. Types may have attributes associated with them, which can then be used as the target of access rules, instead of using a specific type.

When an object is defined in an application policy, any values assigned to the resource's attribute identifier translate into attributes assigned to the generated type for the resource object. At this point, the attributes used can be referenced wherever access rules are allowed. For the purpose of specifying multiple attributes, the set syntax used in SELinux is also applicable for attributes.

```
attribute attrs = { file_type exec_type };
```

Figure 3.7: Declaration of an attribute with default values

Normally, the attribute keyword has a default value of the empty set. This allows developers to define an attribute primitive as a member of a resource class, while not forcing them to assign the attribute primitive to anything. In Figure 3.7, the default value is overridden by the specified set. In Appendix A.2 we see the same example in the Bin resource class. In this case, since we have a default type of `bin_t` for the resource class, the attributes `file_type` and `exec_type` will be added to the type.

3.6 Network Resource Classes

PLEASE needs to be able to handle the existing labeling and access methods for the network object classes to be effective in writing SELinux policy. It is hard to generalize the existing labeling keywords (`netifcon`, `portcon`, and `nodecon`) into primitives to build resource classes upon. We provide three compound primitives to use as building blocks to accomplish this. These compound primitives are resource classes which the language defines for use in providing network functionality. When considering how to abstract these into primitives, we considered allowing the existing keywords to be used. However, this required more knowledge of the base policy language than we thought necessary for the developer. The compound primitives are instantiated and extended in the same way as any other resource class.

`netif:` The first network resource class addresses the functionality provided by the SELinux keyword `netifcon`. This keyword labels all traffic entering or exiting on an interface in a particular manner, and gives the developer a way to label the interface itself for access rules. The `netif` resource class contains the fields `name` used to specify the interface as it appears in ifconfig, the interface context `ifcon` for the interface object, and `packetcon`, the context for packets entering and exiting the interface. These fields translate directly into a `netifcon` statement, as seen in Figure 3.8. In addition, the `ifcon` and `packetcon` fields may be used in access rules. Other objects may inherit from the `netif` object, so a developer can associate permissions with the particular interface.

```
netif eth0 {
    name = eth0;
    ifcon = {
        user = user_u;
        role = staff_r;
        type = if_type_t;
    }
    packetcon = {
        user = user_u;
        role = staff_r;
        type = packet_type_t;
    }
}
```

Emitted Policy:

```
netifcon eth0 user_u:staff_r:if_type_t user_u:staff_r:packet_type_t
```

Figure 3.8: Example of a netif instantiation and its resulting translation

`node:` Figure 3.9 shows the `node` resource class that provides the functionality that is present in the `nodecon` SELinux keyword. The keyword `nodecon` specifies the context for data coming from a group of network nodes. The statement accepts both IPv4 and IPv6 addresses and subnets. Like the `netif` keyword, `node` contains fields for each of the sections of its corresponding base keyword. In this case, the three fields are an IPv4 or IPv6 `address`, `subnet`, and the node context `nodecon`. As with `netif`, the `node` resource class can also be inherited from, and its members may be accessed in a similar fashion.

```
node appnode {
    address = 192.168.0.1;
    subnet = 255.255.255.255;
    nodecon = {
        user = user_u;
        role = staff_r;
        type = internal_t;
    }
}
```

Emitted Policy:

```
nodecon 192.168.0.1 255.255.255.255 user_u:staff_r:internal_t
```

Figure 3.9: Example of a nodecon instantiation and its resulting translation

`port`: The final networking resource class, `port`, is provided to address the `portcon` keyword. A `portcon` statement specifies a port or ports in the form of either a single number, a comma-separated list of numbers, or a hyphenated range and the context associated to it. As seen in Figure 3.10, the PLEASE `port` object contains a port `number`, the protocol `protocol` (udp, tcp), and the context used to label the object `portcon`. Like `netif` and `node`, the `port` object is extensible to allow for permissions to be defined, and has accessible members.

```
port appport {
    port = 21, 70-75, 30060;
    protocol = tcp;
    portcon = {
        user = user_u;
        role = staff_r;
        type = app_data_t;
    }
}
```

Emitted Policy:

```
portcon tcp 21       user_u:staff_r:app_data_t
portcon tcp 70-75   user_u:staff_r:app_data_t
portcon tcp 30060    user_u:staff_r:app_data_t
```

Figure 3.10: Example of a portcon instantiation and its resulting translation

3.7 Resource Permissions

Many applications in the reference policy define several different target types with identical interfaces. PLEASE addresses this with a `permission` keyword that declares permission blocks. Since the permissions are associated with the resource class, the developer does not need to code the interfaces for each application they are using. To avoid the need to specify types for the permissions, PLEASE infers the types for the access rules. The source for the rule comes from using the permission in an application block. The target of the rule is either derived from the type of the resource object, or, in the case of an object having multiple labels, the identifier of a label.

```
permission Read extends SearchBase {
  allow context:file_type { getattr read };
  if(file_type != Symlink && file_type != SockFile) {
    allow context:file_type { lock ioctli };
  }
}
```

Figure 3.11: Example of a resource permission

In Figure 3.11, we see the declaration of a permission named `read`. The `allow` keyword is the same as the SELinux keyword with the same name. Normally, there would be two types after each of the keywords, as seen in Figure 3.12. However, since we eliminate the need to be concerned with types, these are not needed.

```
allow crunch_t bin_t:file { getattr read };
dontaudit crunch_t bin_t:file { getattr read };
```

Figure 3.12: Example of SELinux access rules

3.8 Static Resources

PLEASE specifies the default file context for files in the namespace. Normally, these statements are issued in the `*.fc` files of the reference policy. This requires the developer to specify a path and the context associated with the path. Static blocks remove the need to manage these contexts. These static resources are created by trusted sources and always exist in the namespace in specific locations. Static blocks can only be placed within a resource or application definition. All resources listed generate the appropriate statements in a corresponding `*.fc` file based on the label assigned to either that particular resource or the application type. An example of a static block is shown in Figure 3.13.

As a resource may have more than one label, it is important to allow the developer to specify which label is associated with this `static` block. The format of a `static` block specifies the identifier which refers to the block and the label associated with the block. In this case, we have two default values for the static resource block. If this area was blank, all desired paths would have to be specified when the object is instantiated. At this time we do not know the value of

```
resource File {
    label var_con;
    static file_list var_con {
        /opt/crunch
        /opt/crunch/*
    }
}
```

Figure 3.13: Example of a static resource block

var_con so that will be filled in later. When var_con is finally instantiated, the entries for the two paths will be created in the `crunch.fc` file with the label given to var_con.

3.9 Object/Permission Inheritance

Many of the interfaces defined in SELinux share many commonalities. For instance, the ability to write to a file is frequently defined as having both the read and write permissions on the file. Because of this, PLEASE defines object and permission inheritances for resource classes and their associated permissions. PLEASE allows multiple inheritance of object permissions and provides special syntax for specifying base permissions. Also, the default behavior for inheritance is to extend the base functionality instead of overriding it. We provide the modifier keyword `override` to signify that the compiler should not include the base definition.

Figure 3.14 shows a brief example of using a base object for further abstractions. The Directory and File resource classes share a common base containing a group, two labels, and a static block. The `extends` keyword for the File declaration means that all of PatternBase's members and permissions are accessible in our object. If PatternBase contained a Read permission, then the read permission in File would compose the rules in the base resource, with the ones we specified to form the final permission.

The second usage of `extends` is in inheriting rules from permissions. As seen in Figure 3.14, there are several uses of `extends` on permissions. Since all of the permissions require a common set of permissions on the container directory, we define searchbase as a parent permission. From here we implement `read` and `mmap` as normal. The final use of extends is to generate an `exec` permission by extending `mmap` to include `execute_no_trans`. This forms a chain of inheritance making the final `exec` permission a composition of `exec`, `mmap` and `searchbase`. When specifying the permission to extend, the compiler first uses the permissions from the current resource. In the event that the current resource does not posses the desired permission, it checks up the inheritance chain for the permission. Additionally the developer may specify a specific parent class to inherit from.

3.10 Interactions/Interfaces

In this section, we outline the methods for defining interactions between resources and an application and between an applications resources and another application's policy. Because of the

```
resource PatternBase {
  optional group member;
  optional label context;
  optional label container;
  optional static file_list;
}

resource File extends PatternBase {
  permission searchbase {
    allow container:dir { getattr search };
  }

  permission read extends searchbase {
    allow context:file_type { getattr read };
    if(file_type != Symlink && file_type != SockFile) {
      allow context:file_type { lock ioctli };
    }
  }

  permission mmap extends searchbase {
    if( file_type != Regular ) {
      #not allowed should throw refpolcy warning
    }
    allow context:Regular { getattr read execute };
  }
  permission exec extends mmap {
    allow context:Regular { execute_no_trans };
  }
}
```

Figure 3.14: Excerpts from the File resource class definition.

abstraction of types, when a particular object is specified to interact with another, it is not necessarily the only object with that type.

`action:` An application policy contains one `action` block that specifies all of the access rules. In Section 3.7 we defined permissions which are templates for use in an action block. In addition to this, `application` blocks may define a similar construct, called an `interface`. Permissions are abstract and are filled in based on the instance of the resource class they belong to. When defining an interface, all the relevant type information is available so that it can be defined in a more concrete manner. In Figure 3.15 we see a series of permissions being invoked in the `action` block. At this point we have the source type which comes from our `application` block and our target type which is present in our resource class instances. In the case of the `config.Read` statement, the target type is `crunch_config_t` since it is an isolated resource and our source type is `crunch_t`.

In addition to permissions and interfaces, there are two more groups of access keywords. The first of these mirrors the four original Access Vector (AV) rules that SELinux defines. The last keyword, `verbatim`, is used for backwards compatibility. This allows the developer to use statements from the original policy code, if they are not expressible through PLEASE.

`interface:` Although interfaces may be used in an action block, they are defined outside of the action block. Interfaces are defined in a way similar to how permissions are defined for a resource class. The difference is that when defining a permission in a resource class, we do not know the source for the target at that time. However, since we know the source and the target for an interface, we have an extended version of the access rules that allows the developer to specify both. In Figure 3.15 we define two interfaces. There, `config` has a solidified type which can be exposed to other applications. This is very similar to a permission except that we derive the target type from the calling application.

`allow`, **`auditallow`**, **`dontaudit`**, **`neverallow`**: The four access vector rules have the same meaning as their SELinux counterparts. The only difference in the syntax is that the PLEASE keywords are specified by the identifier of a resource object or by a particular label within that object. In addition to specifying the target and source via an identifier, one can also use a special identifier named `self` to refer to the object containing the rule. In the case of an `application` block, this is the base type of the application itself. In the case of a resource class instance, it is the type of that instance. In Figure 3.15 we see a `dontaudit` statement that is using a member variable from the File resource class. This means that when the rule is constructed in the output file, this context will be pulled from `config.context` instead of the resource class's type.

`verbatim:` The last of the action keywords is `verbatim` which tells the compiler to emit the text in this block as is. The placement of this text is determined by where in the code it appears. The compiler places the verbatim text immediately after the rule it follows.

The emitted policy code in Figure 3.15 is actually shorter than the PLEASE policy code needed to generate it, due to PLEASE's more structured but sometimes more verbose nature. This slight increase in lines of code is only seen when using the PLEASE abstractions. The emitted abstractions, however, still decrease the total size compared to the original reference policy. Essentially, PLEASE slightly increases the size of the `*.te` files, while substantially decreasing the size of

```
application crunch {
    isolated File config; //type for /etc/crunch.conf
    isolated File log; //type for /var/log/crunch
    //resource class for contents of a crunch directory
    File contents {
        file_type = Regular;
        file_list {
            /* Label our source directory and its files */
            /opt/crunch
            /opt/crunch/*
        }
    };
    //System's shared libraries.
    libso shared_libraries;

    action {
        config.Read;
        config.Mmap;
        log.Append;
        contents.Read;
        contents.Mmap;
        shared_libraries.Use;

        dontaudit config.context:file { getattr read execute };
    }
    interface ReadConfig {
        config.Read;
    }
    interface ReadLog {
        log.Read;
    }
}
```

Emitted Policy:

```
type crunch_t;
type crunch_exec_t;
allow crunch_exec_t crunch_t: \
file { entrypoint read getattr lock execute ioctl };

type crunch_config_t;
type crunch_log_t;

allow crunch_t crunch_config_t:file { getattr read mmap };
allow crunch_t crunch_log_t:file { getattr append lock ioctl };
allow crunch_t self:file { getattr read mmap };

libs_use_ld_so(crunch_t);

dontaudit crunch_t crunch_config_t:file { getattr read execute };

interface(`ReadConfig',`
    gen_require(`
        type crunch_config_t;
    ')
    allow $1 crunch_config_t:file { getattr read };
')
interface(`ReadLog',`
    gen_require(`
        type crunch_log_t;
    ')
    allow $1 crunch_log_t:file { getattr read };
')
```

Figure 3.15: Application policy for crunch and the emitted code

the `*.if` files and maintaining the size of the `*.fc` files, yielding an overall significant decrease in lines of policy. In addition to the reduction in policy size, PLEASE code is more readable; for example, the emitted policy contains M4 style macros and quoting which are harder to read and understand.

Chapter 4

Implementation

The PLEASE compiler is implemented in Java with two common technologies for generating parsers and abstract syntax trees: JJTree and JavaCC [15]. JJTree describes the nodes of the abstract syntax tree and then emits a grammar file for JavaCC. This grammar file contains additional Java code for creating the tree. JavaCC is similar to Yacc, in that it takes a grammar and emits a parser. These two technologies allow us to create a compiler that is cross-platform without any additional effort.

The PLEASE compiler is an ongoing effort, but has made reasonable progress so far. Our two-person team has put roughly 40 man-hours of time into the compiler, to date. The input file to JJTree contains 456 lines, describing the complete grammar for PLEASE. This yields an additional 3,272 lines of Java. This, coupled with additional code, such as the symbol table, brings the total up to 4,210 lines of code.

Chapter 5

Use Cases

Although it is possible to create a new policy for SELinux from the ground up, there are already worthwhile abstractions that exist in the reference policy. In this section, we discuss two areas where PLEASE has been used to reduce the size and complexity of Reference policy abstractions. The first example shows a way to recreate some of the high-level concepts represented in the `corecommands` series of files. The second example shows that the base abstractions of the reference policy can be recreated to greatly reduce the code size and complexity. In both of these examples, we reduce the code size by a factor of 2.8 overall. However, the two examples contain five individual use cases that yield reductions ranging from a factor of 3 to 8 for the more common cases, and as high as a factor of 24 for certain special cases.

5.1 Corecommands Abstractions

After a survey of the reference policy, we decided to choose the `corecommands` abstraction as a sample use case. The `corecommands` abstraction contains several types and interfaces that provide access to the common system binaries. From an inspection of the types in the file and determining commonalities, it is clear that the `bin_t` type is the key component of the abstraction since several of the other types are aliases of that type. There is also an `exec_type` attribute which is placed on all executable types. The purpose of this type and attribute is to grant rights on a large number of executable types.

The type for generic binary files is `bin_t`. PLEASE intends to abstract concepts such as this. The `corecommands.if` file yields a list of interfaces that could be called on this type, and many of these interfaces share common access statements. For example, all of the read interfaces require the same search permissions on directories. The permission to search directories of type `bin_t` is declared to be used as a base for other permissions. The `exec` permission is defined with multiple inheritance, as it is based on listing directory contents and reading `symlinks`, which are two previously defined permissions.

The ability to inherit permissions reduces the number of lines of code needed to represent the `corecommands` resource class. However, the same interfaces are also defined for the `sbin_t` type. These interfaces are defined as calls to the `bin` interface, but with different types. Even though their interfaces contain mostly the same code, they still need to be redefined. In PLEASE, the only thing needed for the `sbin_t` interface is a new resource that extends `bin` and overrides the default type, making it `sbin_t` instead of `bin_t`.

In addition to `sbin_t` and `bin_t`, the attribute `exec_type` is placed on all executable types. Some of the permissions in the interfaces for `exec_type` share commonalities with the `Bin` resource class permissions. Because of this, the code can be reused by inheriting from `Bin`. Many of the interfaces for `exec_type` require list permissions on directories of type `bin_t`. The class to inherit from is specified, to avoid ambiguities in resolving which `list` permission to use. In this case, we specify `Bin.list` as the base for the `getattr` and `exec` permissions.

After rewriting the abstractions in PLEASE we checked the line count of the original reference policy abstractions against our new implementation. The original reference policy implementation has 204 lines of code, while the PLEASE implementation of the abstractions has only 74 lines: a factor of 2.8 reduction in size. Although a large portion of this was due to the `sbin_t` interfaces being exact duplicates of the `bin_t` interfaces, we expect to see a reduction on this order for most of the abstractions.

5.2 Access Pattern Abstractions

There are several files in the reference policy that contain support macros and definitions. One of these contains a series of macros, referred to as patterns. The patterns are common access rights that can be issued on a type. These are broken up into file and IPC access patterns. We look at the patterns pertaining to files, including directories, regular files, sockets, named pipes, symlinks, character, and block files. This area is an example where the abstractions provided by PLEASE can drastically decrease code size.

The patterns in this file grant access either on objects of the `dir` object class or on objects of one of the `file` object classes. The opportunity to reuse code exists in both of these sets. However, the `file` set shows an example of how permission inheritance and use of the `class` keyword can drastically lower the amount of code needed.

As the benefits of creating a directory resource class were shown in the first use case, we will now focus on file resource classes. There are six types of files that have object classes in SELinux. The permissions needed to grant particular access rights on each file type are often identical. The ability to search the container directory of the files is needed for many of these patterns. We implemented this as a base permission called `searchbase`, which extends many of the permissions. In addition to this, permissions such as `create` and `delete` need the ability to add and remove entries from the container directory. These were also extracted into two base permissions. Finally, `rename` and `manage` need full read and write access to the container directory, yielding a fourth base permission. This took only twelve lines of code to implement, while the original file patterns contain 79 distinct statements to granting search permissions on the container directory. By having these permissions implicitly added through permission inheritance, we reduce the code size for those shared statements by a factor of 6.6.

Another large reduction in code comes from the ability to replace the specific object class with a `class` variable. In Figure 3.6 we show the use of the `class` keyword in abstracting a file resource class. This mapping allows us to condense six separate patterns into just one PLEASE permission. Permissions such as `setattr`, `getattr`, `create`, and `unlink` are identical except for the object class. By using a `class` variable named `file_type`, we are able to substitute that identifier into the object class portion of the access rule. This turned 24 lines of patterns into a three line permission that inherits from a common base and contains one access rule. This represents a factor of 8 code size reduction.

Some of the permissions defined for the file abstraction are actually more complex than those mentioned above, but they still exhibit a reduction in code size. Permissions such as `rename` and `manage` have several special cases to take into account. The conditional statements in PLEASE allow us to conditionally add policy, based on the value of the `file_type` variable. The `fifo` and `sock_file` object classes do not have defined permissions for `rename`. This conditionally throws a reference policy warning. This adds three lines to our policy, but removes eight lines of invalid patterns. The remaining object classes all share the same permissions for `rename`, which offsets the lines used by the conditional statement.

PLEASE yields even more substantial gains in many instances. For example, the pattern `rw` is just a combination of `read` and `write`, and the pattern `relabel` is just a combination of `relabelto` and `relabelfrom`. In the reference policy, there are six separate macros to represent each of these two patterns. In PLEASE, each is represented in only one line. By composing `rw` as a permission that inherits from both `read` and `write`, no additional access rules are required. Whereas this is a special case, it reduces 24 lines of reference policy code into just one line of PLEASE code.

Chapter 6

Related Work

There have been several other attempts to address all of the problems present in SELinux, as discussed in Section 1.1. These projects can be placed into two categories: tools that generate SELinux policy code, and alternate languages for SELinux policy. The second category includes the approach similar to our own of making a higher-level policy language on top of the existing SELinux policy language. Although each attempt has different benefits, they each also have some limitations.

6.1 Policy Generation Tools

MITRE's Polgen [13] is a tool that generates SELinux policy code automatically, with the help of some human-made annotations. Polgen uses the information flow patterns it detects from the `strace` of an application to generate the appropriate policy for it. The developer assists in making policy decisions when there is not enough information for the tool to guess itself. Although this can be a useful start, the process of using learning models as a form of generated policy has several flaws. Learning models do not necessarily exercise all possible paths, and the data used to train the learning model must not be malicious. This tool is meant for developers who are not necessarily familiar with the Linux capabilities system.

Virgil [4] is a tool that generates SELinux policy code according to the specifications that a user has selected through its graphical user interface. This tool only allows a policy developer to create isolated domains, with no ability to specify any interactions between domains. Though it makes it significantly faster to generate policy, Virgil still requires that the developer understands SELinux and its capabilities system. Virgil is rooted in the abstractions provided by the Reference Policy, and is limited in what it can express.

The SELinux Policy Editor (SEEdit) [9] by HitachiSoft and George Washington University is composed of a graphical user interface and a simplified version of the policy language. The developer only needs to understand the application that needs policy, without needing any other knowledge of SELinux. The simplified language is easier to understand because it does not include the `type` label. However, removing types takes away the ability to express some important aspects of the original policy.

6.2 Higher-Level Policy Languages

Tresys Technologies created the CDSFramework [12] policy language based on cross-domain solutions. CDSFramework is a higher-level policy language that focuses on information flow and domain separation. It provides a framework to develop guard applications for cross-domain solutions. This language allows the policy to be derived directly from the specified security architecture. Although this is important, it relies on having functioning policies for the applications that interact with each other. Tresys is also working on a CDSFramework IDE, a graphical user interface that an application developer can use to generate the desired SELinux policy code.

Purdue University's SENG [6] introduces well-defined abstractions on top of the existing policy, potentially eliminating the need for complex macros throughout the policy language. These abstractions can then be translated into the original policy language. SENG provides some useful extensions to the base SELinux policy language, such as permission sets, custom type transitions, and abstract resources and permission. SENG's abstract resources and permissions are structured in a procedural manner, and still requires that the policy developer has some knowledge of types. Removing the need for the M4 macros in the policy language makes it possible to automatically analyze the policy created for an application [19].

Lopol [5], by the University of Tulsa, uses a deductive database system to rewrite the existing policy into a higher-level set of logical rules, making it more readable. Lopol is able to represent what security rules a particular application policy must follow, making it possible to enforce these rules through rewriting the policy. However, these rules have not yet been formalized.

Chapter 7

Future Work

Additional Language Features. There are several additional features that we would like to see in the base PLEASE language. The first of these is the ability to embed resource class instances in the definition of other resource classes. This would allow for a more robust system where the developer can include functionality already implemented in another resource class in the new resource class.

The second feature we see being useful is the ability to pass in instances of primitives or resource classes into calls to permissions and interfaces. This would allow similar functionality to embedded resource instances. However, it would allow the developer to utilize this without having the embedded resource. This functionality should be type checked to provide facilities for proper error reporting. Without resource types being enforced, the system would act simply as a macro-expansion mechanism.

Another feature that we see being beneficial to developers is a clearer method for understanding and representing type transitions. As of now, there are few direct uses of the `type_transition` rule in the reference policy. Most of these references are found in the interface that allows a file of a certain label to transition into a security domain through the program's execution. However, this is not the sole use of this functionality. Type transitions are also used to specify the labeling on transient objects such as pipes, temporary files, message queues and their corresponding messages, etc. If we provide a better mechanism for this, the developer can properly create sub-domains for these transient files and limit access to them further.

Policy Analysis Plugin Architecture. One thing that policy developers want is to be able to verify the correctness of their policy. There have been several attempts at creating policy analysis tools for SELinux [2, 5, 10, 11]. One benefit of having used JJTree and JavaCC for the PLEASE compiler is that they already have a plug-able architecture for performing actions based on the abstract syntax tree. Without an additional framework to aid in policy analysis, a person who wished to analyze their policy would have to know about the internal compiler representation of PLEASE, and would also have to be familiar with Java and the internals of JJTree and JavaCC. A proposed solution for this is to first develop some use cases based on common policy analysis tasks. Then develop an intermediate representation that best expresses the needs of these tasks, and a visitor that outputs this representation. After this, a tool can be developed to visualize this data and verify constraints against it.

Graphical Interface. When designing PLEASE, we were careful to design the language in a way that would be easily expressed in a graphical form. One idea we have for such a tool is an application that allows the developer to build resource objects in a fashion similar to UML. In this model, the developer would have a catalog of all the PLEASE primitives and existing resource classes and would be able to drop them into boxes representing resource classes. Defining permissions for a particular object class could be done by allowing the developer to pick from either permissions from embedded resources or SELinux object classes and the associated permissions. An application like this would be able to house code snippets similar to the way other IDEs contain macros or templates for common actions that developers perform. The application would contain a library of such snippets for use in defining permissions and interfaces. Defining an application's policy and its interfaces would be the same as designing a resource class, except that access to other application's interfaces are allowed in addition to primitives and resource classes.

Multi-Level Security. At a later time PLEASE could be extended to take into account defining security levels and categories to support MLS policies. One way of doing this would be to extend the label statements to contain an MLS field. This field would be another primitive that allowed the developer to specify the sensitivities and categories for the system in a way that represents its hierarchical nature [18]. In this method, the developer could specify a group of MLS objects and in the definition of one object specify that it dominates another object. One consideration with addressing this method is that there may cycles in the hierarchy.

Role Based Access Controls. At the time of the writing of this thesis, little work has gone into the RBAC portion of SELinux. The current reference policy only specifies three to five roles, depending if the MLS policy is running. It would be possible to support the RBAC section of SELinux by creating a few more basic blocks to handle user definitions and role definitions. A developer could build a set of roles by assigning particular applications to a role and not be concerned with the underlying types for the application. This way the developer could easily define which applications are needed by a role.

Chapter 8

Conclusion

Although SELinux is able to provide better system security though its strong access mechanisms, its complexity has hindered its adoption. Even security experts have difficulty understanding and using the policy code itself. As a result of this, many administrators and users have problems adapting the policy to their specific environment and third party applications. Without these capabilities, users and administrators will be slow to adopt SELinux.

PLEASE is a higher-level policy language that conceals the complex nature of SELinux's type enforcement. As we have shown, the higher-level abstractions PLEASE provides can increase the "productivity, reliability, and simplicity" of policy development [1]. PLEASE reduces the amount of code needed for a policy by a factor of 2.8–8 for the five use cases we discussed in Chapter 5. In some of these cases, the reduction is as much as a factor of 24. PLEASE makes developing policy code for applications significantly easier. We believe that once policies are available for a larger number of applications, SELinux will be a sensible option to use for system security.

Bibliography

[1] F. Brooks. *The Mythical Man-Month, Anniversary Ed.*, pages 205–226. Addison-Wesley, 1995. Section: “No Silver Bullet” Refired.

[2] Kevin Carr. apol - selinux policy analysis tool, 2006. `http://www.die.net/doc/linux/man/man1/apol.1.html`.

[3] Debian Project. Debian gnu/linux, 2007. `http://www.debian.org/`.

[4] Daniel H. Jones. Virgil: Selinux policy generator. `http://sepolicy-virgil.sourceforge.net/`.

[5] A. Kissinger and J. C. Hale. Lopol: A deductive database approach to policy analysis. In *SELinux Symposium*, 2006. `http://selinux-symposium.org/2006/papers/10-lopol.pdf`.

[6] P. Kuliniewicz. Seng: An enhanced policy language for selinux. In *SELinux Symposium*, 2006. `http://selinux-symposium.org/2006/papers/09-SENG.pdf`.

[7] Peter Loscocco and Stephen Smalley. Integrating flexible support for security policies into the linux operating system. Technical report, National Security Agency, February 2001. `http://www.nsa.gov/selinux/papers/slinux.pdf`.

[8] Frank Mayer, Karl MacMillan, and David Caplan. *SELinux by Example*. Prentice Hall, August 2007.

[9] Yuichi Nakamura. Simplifying policy management with selinux policy editor, 2005. `http://selinux-symposium.org/2005/presentations/session4/4-2-nakamura.pdf`.

[10] Prasad Naldurg, Stefan Schwoon, Sriram Rajamani, and John Lambert. Netra: Seeing through access control. In *FMSE ’06: Proceedings of the fourth ACM workshop on Formal methods in security*, pages 55–66, New York, NY, USA, 2006. ACM Press.

[11] Beata Sarna-Starosta and Scott D. Stoller. Policy analysis for security-enhanced linux. In *Proceedings of the 2004 Workshop on Issues in the Theory of Security (WITS)*, pages 1–12, April 2004. Available at http://www.cs.sunysb.edu/˜stoller/WITS2004.html.

[12] C. Sellers, J. Athey, S. Shimko, F. Mayer A. Wilson, and K. MacMillan. Experiences implementing a higher-level policy language for selinux. In *SELinux Symposium*, 2006. `http://www.tresys.com/files/docs/HiLang-SELinux-Symp.pdf`.

[13] B. T. Sniffen, D. R. Harris, and J. D. Ramsdell. Guided policy generation for application authors. Technical report, MITRE, February 2006. `http://www.mitre.org/work/tech_papers/tech_papers_06/06_0046/06_0046.pdf`.

[14] Ray Spencer, Stephen Smalley, Peter Loscocco, Mike Hibler, David Andersen, and Jay Lepreau. The flask security architecture: System support for diverse security policies. Technical report, National Security Agency, August 1999. `http://www.nsa.gov/selinux/papers/flask.pdf`.

[15] Sun Microsystems. Java compiler compiler (javacc) - the java parser generator, 2007. `https://javacc.dev.java.net/`.

[16] Tresys Technology. Reference policy. `http://oss.tresys.com/projects/refpolicy`.

[17] Tripwire Inc. Tripwire Software. `www.tripwire.com`.

[18] US Department of Defense. Trusted computer system evaluation criteria (The Orange Book). Technical Report DoD 5200.28-STD, National Computer Security Center, Alexandria, VA, December 1985. `www.dynamoo.com/orange`.

[19] Gary V. Vaughan. Gnu m4. `http://www.gnu.org/software/m4/`, 2006.

Appendix A

PLEASE Policy Examples

A.1 Application Policy Example

This section defines a sample application policy for an application named “crunch.” Crunch is a simple program that takes files in a particular directory and then hashes them and emits the results to its own ad hoc log file. Crunch uses isolated types for its log file and configuration data. In addition to this it needs access to the system’s shared libraries in order to access cryptographic functions. We make the assumption that we have already defined a “libso” resource class that contains the permission use, which grants the ability to access and use the system’s libraries. Crunch also `mmaps` its configuration and contents files. The `mmap` operation on Crunch’s configuration files can fail often so we indicate not to audit these errors to avoid flooding our log.

```
application crunch {
    #type for /etc/crunch.conf
    isolated File config;
    #type for /var/log/crunch
    isolated File log;
    #resource class for contents of a crunch directory
    File contents {
        file_type = Regular;
        file_list {
            /* Label our source directory and its files */
            /opt/crunch
            /opt/crunch/*
        }
    };
    #System's shared libraries.
    libso shared_libraries;

    action {
        config.Read;
        config.Mmap;
        log.Append;
        contents.Read;
        contents.Mmap;
        shared_libraries.Use;

        dontaudit config.context:file { getattr read execute };
    }
    interface ReadConfig {
```

```
        config.Read;
    }
    interface ReadLog {
        log.Read;
    }
}
```

A.2 CoreCommands Policy Examples

Bin Resource Abstraction:

The reference policy defines an abstraction to handle all executable files that are located in the system's `bin` and `sbin` directories. The `Bin` resource class is the PLEASE representation of the same abstraction. The resource contains several components which allow it to function the same way as the reference policy implementation. The two primitives seen in the example specify that this object has a default type of `bin_t` and that it should be given the `file_type` attribute. The second section of the example are the permissions. These are derived straight from the interfaces provided by the reference policy for the `bin` and `sbin` abstractions.

```
resource Bin
{
  attribute attrs = { file_type exec_type };
  default label context {
    optional user;
    optional role;
    type = bin_t;
  };
  permission search {
    allow context:dir { getattr search };
  }
  permission list {
    allow context:dir { getattr search read lock ioctl };
  }
  permission getattr extends search {
    allow context:file { getattr };
  }
  permission read extends getattr {
    allow context:file { read lock ioctl };
  }
  permission readlinks extends search {
    allow context:lnk_file { getattr read };
  }
  permission readpipes extends search {
    allow context:fifo_file { getattr read lock ioctl };
  }
  permission readsockets extends search {
    allow context:sock_file { getattr read };
  }
  permission exec extends { list readlinks } {
    allow context:file { read getattr lock execute ioctl execute_no_trans};
  }
  permission manage {
    allow context:dir { read getattr lock search ioctl add_name remove_name write };
    allow context:file { create getattr setattr read write append rename link unlink ioctl lock };
  }
  permission relabel extends search {
    allow context:file { getattr relabelfrom relabelto };
  }
  permission mmap extends search {
    allow context:file { getattr read execute };
  }
}
```

Extension of the Bin resource to provide SBin:
The `SBin` resource class is identical so `Bin` in every way except for the underlying type. This allows for `SBin` to be a child class of the `Bin` resource with the implementation overriding the default type of the resource class.

```
resource SBin extends Bin
{
  default label context {
    optional user;
    optional role;
    type = sbin_t;
  }
}
```

Example of the Abstraction of an Attribute:
The AllExecutables resource class is an example of using an attribute instead of a type for permissions in a resource class. In this example we are specify a default type of `exec_type` for the attribute variable. If we had specified multiple attributes then each of the rules in the permission statements would generate multiple allow statements.

```
resource AllExecutables extends Bin
{
  attribute attr = exec_type;

  permission getattr extends Bin.list {
    allow attr:file { getattr };
  }
  permission exec extends Bin.list {
    allow attr:file { read getattr lock execute ioctl execute_no_trans};
    allow attr:lnk_file { getattr read };
  }
  permission manage {
    allow context:dir { read getattr lock search ioctl add_name remove_name write };
    allow attr:file { create getattr setattr read write append rename link unlink ioctl lock };
    allow attr:lnk_file { create read getattr setattr unlink rename };
  }
  permission relabel extends Bin.search {
    allow attr:file { getattr relabelfrom relabelto }
  }
  permission mmap extends Bin.search {
    allow attr:file { getattr read execute };
  }
}
```

A.3 Sample Policy for Pattern Abstraction

Base resource used to create Directory and File:
To eliminate the wasteful replication of policy we use the PatternBase resource class as a parent for the Directory and File resource classes. The context variable in this example is the label of the actual object being referred to while the container is the label of the parent directory.

```
resource PatternBase {
  optional group member;
  optional label context;
  optional label container;
  optional static file_list;
}
```

Directory resource class:
The Directory resource class is a conversion of the Directory pattern found in the reference policy. Since Directory extends PatternBase, it contains the three primitives belonging to that class. The remainder of the implementation is the permission statements which show the use of an identifier as the target or an access rule.

```
resource Directory extends PatternBase {

  permission searchbase {
    allow container:dir { getattr search };
  }
  permission getattr extends searchbase {
    allow context:dir { getatttr };
  }
  permission setattr extends searchbase {
    allow context:dir { setatttr };
  }
  permission search extends searchbase {
    allow context:dir { getattr search };
  }
  permission list extends searchbase {
    allow context:dir { getattr search read lock ioctl };
  }
  permission add_entry extends list {
    allow context:dir { add_name };
  }
  permission del_entry extends list {
    allow context:dir { remove_name };
  }
  permission create {
    allow container:dir { getattr search lock ioctl write add_name };
    allow context:dir { getattr create };
  }
  permission delete {
    allow container:dir { getattr search lock ioctl write remove_name };
    allow context:dir { getattr rmdir };
  }
  permission rename {
    #omitted since the refpolicy is missing rename_dir_perms
  }
  permission manage {
    allow container:dir { read getattr lock search ioctl add_name remove_name write };
```

```
    allow context:dir { create getattr setattr read write link unlink rename search
  }
  permission relabelfrom extends searchbase {
    allow context:dir { getattr relabelfrom };
  }
  permission relabelto extends searchbase {
    allow context:dir { getattr relabelto };
  }
  permission relabel extends { relabelfrom relabelto } {}
}
```

File Resource Class:
The File resource class is more complex than the Directory resource class. It shows the extension of PatternBase while including new primitives. In this case an instance of the class primitive named "file_type" is created. The class primitive serves an important purpose in this case. Since several of the permissions defined for File are identical except for a change in the underlying object class we can make use of this commonality with the class primitive. In this case the class primitive is defined to have six valid variables. Instead of using a specific object class we can substitute this variable in for the object class field of our allow rule. In the case of `getattr` and `setattr` this condenses the access statement for all six object classes to one statement.

```
resource File extends PatternBase {

  class file_type {
    default Regular = "file";
    Symlink = "lnk_file";
    FIFO = "fifo_file";
    BlockFile = "blk_file";
    CharFile = "chr_file";
    SockFile = "sock_file";
  }
  // Base permissions
  permission searchbase {
    allow container:dir { getattr search };
  }
  permission addentrybase {
    allow container:dir { getattr search lock ioctl write add_name };
  }
  permission delentrybase {
    allow container:dir { getattr search lock ioctl write remove_name };
  }
  permission rwdirbase {
    allow container:dir { read getattr lock search ioctl add_name remove_name write };
  }
  // Common Permissions
  permission getattr extends searchbase {
    allow context:file_type { getattr };
  }
  permission setattr extends searchbase {
    allow context:file_type { setattr };
  }
  permission read extends searchbase {
    allow context:file_type { getattr read };
    if(file_type != Symlink && file_type != SockFile) {
      allow context:file_type { lock ioctli };
    }
  }
  permission mmap extends searchbase {
    if( file_type != Regular ) {
      #throw refpolcy warning
    }
    allow context:Regular { getattr read execute };
  }
  permission exec extends mmap {
    allow context:Regular { execute_no_trans };
  }
  permission append extends searchbase {
    if( file_type == Symlink ) {
      #throw refpolcy warning
    } else {
```

```
      allow context:file_type { getattr append lock ioctl };
    }
  }
  permission write extends searchbase {
    if ( file_type == SockFile ) {
      allow context:SockFile { getattr write append };
    } else if ( file_type == Symlink ) {
      allow context:Symlink { getattr write lock ioctl };
    } else {
      allow context:file_type { getattr write append lock ioctl };
    }
  }
  permission rw extends { read write } {}
  permission create extends addentrybase {
    allow context:file_type { getattr create };
  }
  permission delete extends delentrybase {
    allow context:file_type { getattr unlink };
  }
  permission rename extends rwdirbase {
    if ( file_type == FIFO || file_type == SockFile ) {
      #throw refpolcy warning
    } else {
      allow context:file_type { getattr rename };
    }
  }
  permission manage extends rwdirbase {
    if ( file_type == SockFile ) {
      allow context:SockFile { create getattr setattr read write rename link
    } else if ( file_type == Symlink ) {
      allow context:Symlink {  create read getattr setattr unlink rename getattr
    } else {
      allow context:file_type { create getattr setattr read write append rename link
    }
    if ( file_type == BlkFile || file_type == ChrFile ) {
      allow self:capability mknod;
    }
  }
  permission relabelfrom extends searchbase {
    allow context:file_type { getattr relabelfrom };
  }
  permission relabelto extends searchbase {
    allow context:file_type { getattr relabelto };
  }
  permission relabel extends { relabelto relabelfrom } {}
}
```

Appendix B

PLEASE Language Description

B.1 Lexical Issues

PLEASE is *case-sensitive*; for example, “Class” and “class” are two distinct lexical entities.

White Space and Comments: Whitespace (spaces, newlines, and tabs) are used to separate tokens; otherwise they are ignored. Whitespace may not appear in any token, even a string constant. PLEASE supports three types of comments:

- Multi-line (C-Style) comments that start with “/*” and end with “*/”. As in C, these comments may not be nested. These comments are not placed in the output file.
- Single-line comments starting with “//” and ending at the end of the line. These comments are also not placed in the output file.
- SELinux comments start with “#” and end at the end of the line. These comments are placed in the output file based on where they appear in the input file.

Reserved Words:
The following are reserved words:

label	user	role	type	static	group	class
attribute	optional	default	isolated	override	resource	permission
extends	verbatim	application	action	interface	if	else
true	false	allow	auditallow	dontallow	neverallow	

Identifiers:
PLEASE identifiers may only contain letters [“a” – “z”] , digits [“0” – “9”] and underscores. An identifier must start with a letter and then may be followed by any number of letters, digits, and underscores.

Constants:
PLEASE only supports one type of constant. For the purpose of the class construct we allow strings conforming to the same rules as identifiers.

B.2 PLEASE Grammar:

```
policy ::= ( resource_decl | application_decl )*

resource_decl ::= resource identifier ( extends parent_var_list )? { resource_body }

parent_var_list ::= identifier | { ( identifier )+ }

resource_body ::= ( label_decl | group_decl | attribute_decl
                  | static_decl | class_decl | permission_decl )*

label_decl ::= ( optional | default )? label identifier ( { label_decl_body } )? ;

label_decl_body ::= ( ( optional )? ( user | role | type ) ( = identifier )? ; )*

group_decl ::= ( optional )? group identifier group_list ;

group_list ::= { ( identifier )+ } | identifier

attribute_decl ::= ( optional )? attribute identifier attribute_list ;

attribute_list ::= { ( identifier )+ } | identifier

static_decl ::= ( optional )? static identifier ( identifier )? { }

class_decl ::= class identifier { ( class_element  )? }

class_element ::= ( default )? ( identifier = "identifier" ; | identifier ; )

permission_decl ::= ( override )? permission identifier ( extends parent_list )?
                    { permission_body }

parent_list ::= parent_list_element | { ( parent_list_element )+ }

parent_list_element ::= identifier | identifier . identifier

permission_body ::= ( permission_stmt  | permission_conditional_stmt )*

permission_stmt ::= avrule ( identifier )? : identifier operation_list  ;
                    | identifier . identifier ; | verbatim_stmt

verbatim_stmt ::= verbatim { }

avrule ::= allow | auditallow | dontaudit | neverallow

operation_list ::= identifier | { ( identifier )+ }

permission_conditional_stmt ::= if_stmt ( elseif_stmt )* ( else_stmt )?

if_stmt ::= if (  boolean_expr ) { permission_body }

elseif_stmt ::= else if ( boolean_expr ) { permission_body }

else_stmt ::= else { permission_body }
```

```
boolean_expr ::= boolean_equals_expr ( ( && | || ) boolean_equals_expr )*

boolean_equals_expr ::= identifier . class == identifier

application_decl ::= application identifier { application_body }

application_body ::= ( primitive_stmt | resource_stmt | interface_stmt
                     | action_stmt )*

primitive_stmt ::= label_stmt | group_stmt | attribute_stmt  | static_stmt

label_stmt ::= label identifier = label_stmt_body

label_stmt_body ::= ( ( user | role | type ) = identifier ; )*

group_stmt ::= group identifier = group_list ;

attribute_stmt ::= attribute identifier = attribute_list ;

static_stmt ::= static identifier ( identifier )? { }

resource_stmt ::= identifier identifier { resource_stmt_body }

resource_stmt_body ::= ( ( identifier = ( label_stmt_body  | group_list
                       | "identifier" ) ) | ( identifier identifier { } ) )*

action_stmt ::= action { action_stmt_body }

action_stmt_body ::= ( avrule ( identifier )? ( identifier )? : identifier
                     operation_list ; | identifier . identifier ;
     | verbatim_stmt )*

interface_stmt ::= interface identifier { interface_stmt_body }

interface_stmt_body ::= ( avrule ( identifier )? ( identifier )? : identifier
                        operation_list ; | identifier . identifier ;
                        | verbatim_stmt )*
```