High-level Formal Modeling and Verification of Mandatory Access Control Policies Across Multiple Security-Enhanced Linux Devices

A Thesis

Presented in Partial Fulfillment of the Requirements for the

Degree of Master of Science

with a

Major in Computer Science

in the

College of Graduate Studies

University of Idaho

by

Jared Thomas Zook

Major Professor: Daniel Conte de Leon, Ph.D.

Committee Members: Michael Haney, Ph.D.; Axel Krings, Ph.D.

Department Administrator: Frederick Sheldon, Ph. D.

December 2016

Authorization to Submit Thesis

This thesis of Jared Thomas Zook, submitted for the degree of Master of Science with a Major in Computer Science and titled "**High-level Formal Modeling and Verification of Mandatory Access Control Policies Across Multiple Security-Enhanced Linux Devices**," has been reviewed in final form. Permission, as indicated by the signatures and dates given below, is now granted to submit final copies to the College of Graduate Studies for approval.

Major Professor: _____Date_____
Daniel Conte de Leon, Ph.D.

Committee Members: _____Date_____
Michael Haney, Ph.D.

_____Date_____
Axel Krings, Ph.D.

Department Administrator: _____Date_____
Frederick Sheldon, Ph.D.

Abstract

The evolution of technological progress continually presents new information security challenges for large enterprises. Organizations must actively implement security policies to mitigate modern threats. Access control policies which define the way in which an organization's principals can interact with a system are particularly vital to enforce and verify. Accordingly, this thesis demonstrates that policies from an enterprise Linux implementation of a mandatory access control scheme can be used to populate a useful and efficient policy model. This model can be queried from a high-level to verify proper implementation of policies across one or more devices in a networked environment. It provides the user with both 1) a graphical representation of one or more policy implementations and 2) a means for an analyst to ensure whether specified actions between subjects and objects are permitted or not, aiding in providing them with an intuitive understanding of the higher-level organizational security policy.

Acknowledgements

I extend my gratitude to my major professor, Dr. Daniel Conte de Leon. His guidance on this thesis project has proven to be invaluable to me time and time again.

In addition, I would like to thank my committee members, Dr. Michael Haney and Dr. Axel Krings for their encouragement and input in the process of completing this thesis.

I would also like to thank Dr. Anna L. Buczak and the Johns Hopkins University Applied Physics Laboratory for providing me with an internship experience that gave me operational knowledge that was directly applicable to my thesis work.

Work on this thesis was made possible as a result of the CyberCorps(R) Scholarship for Service, a generous scholarship financed by a National Science Foundation grant and administered by the U.S. Office of Personnel Management and the University of Idaho Center for Secure and Dependable Systems.

Table of Contents

Authorization to Submit Thesis ..... **ii**

Abstract ..... **iii**

Acknowledgements ..... **iv**

Table of Contents ..... **v**

List of Figures ..... **viii**

List of Listings ..... **ix**

1 Introduction, Research Problem, and Overview **1**

1.1 Introduction 1

1.2 Research Problem 3

1.3 Overview 3

2 Background **5**

2.1 Access Control 5

2.2 Security-Enhanced Linux (SELinux) 9

2.3 SELinux Policies and Type Enforcement 10

2.4 SELinux "Access Vector" Rules 11

2.5 SELinux Domain Transitions 13

3 The Hierarchical Policy Model (HPol) and the High-level, Easily Reconfigurable Machine Environment Specification (HERMES) High-Level Policy Description Language 15

3.1 The Hierarchical Policy Model 15

3.2 High-level, Easily Reconfigurable Machine Environment Specification (HERMES) Output of HPol Models 17

3.3 HERMES Queries with XSB Prolog 19

4 Related Work 22

4.1 Other Policy Modeling and SELinux-Related Projects 22

5 Research Question and Contributions 26

5.1 Research Question 26

5.2 Thesis Contributions 26

6 Contribution 1: Formalizing Enterprise Linux Mandatory Access Control Policies Using HPol 28

6.1 Case Study 1: Parsing of an SELinux Policy with Domain Transitions 29

6.2 Case Study 1: Findings 32

6.3 Case Study 2: Parsing the Default Red Hat Targeted SELinux Policy 35

6.4 Case Study 2: Findings 38

6.5 Case Study 2: Visualization 42

7 Contribution 2: Interactively Answering Queries About SELinux Device Policies 46

8 Contribution 3: Policy Merging for Multiple Device Policy Violation Detection 49

9 Future Work **59**

9.1 Visualization 59

9.2 Selective Parsing 59

9.3 Incorporation of Linux Filesystem Policies 60

9.4 Creating SELinux Configurations Using a High-level Policy Decription Language 60

9.5 Expansion of the Interactive HERMES Querying Script 60

9.6 Merging Policies for Additional Types of Devices 61

10 Summary and Conclusion **62**

10.1 Summary 62

10.2 Conclusion 63

Bibliography **64**

List of Figures

3.1 HPol Model: Alice Read/Write Example 18

6.1 Abbreviated HPol Model Demonstrating the *passwd* Domain Transition 31
6.2 SELinux-to-HPol Parser Code Demonstrating Path Finding for Domain Transitions 40
6.3 SELinux Targeted Policy: Gephi View of Directory Permissions 43
6.4 SELinux Targeted Policy: Initial View in Gephi 44
6.5 SELinux Targeted Policy: ForceAtlas2 Layout View in Gephi for 10,000 Allow Rules 45

8.1 Policy Merging Code from the mergePolicies() Method of the HPol Library . . . 50
8.2 Code to Connect the Merged Start and End Node to the Named Policies from the mergePolicies() Method of the HPol Library 51
8.3 Abbreviated HPol Model Representing Interactions on the Low Privilege Machine 55
8.4 Abbreviated HPol Model Representing Interactions on the High Privilege Machine 56
8.5 SELinux Model Resulting from Merging High and Low Privilege Machine Policies 57
8.6 SELinux Policy Merging: Merging Three Policies 58

List of Listings

3.1 HERMES Policy Output for the Example HPol Model 21
3.2 HERMES Node Output for the Example HPol Model 21
3.3 HERMES Query of the Example HPol Model 21
4.1 Sample SELinux Policy Statements Written in Lobster [1] 24
6.1 HERMES Policy Output for the *passwd* Domain Transition 35
6.2 Attribute and Type Output from the *seinfo* Utility 37
6.3 Allow Rule Output from the *sesearch* Utility 38
6.4 HPol Results from Parsing the SELinux Targeted Policy 39
6.5 An Extraction of HERMES Policy Output from Parsing the SELinux Targeted Policy 41
7.1 Operative HERMES Policy Output for Querying the *passwd* Domain Transition . 47
7.2 Output from Querying HERMES Policy Output from the *passwd* Domain Transition 48
7.3 A Sample Policy from Parsing the SELinux Targeted Policy 48
7.4 Query Result for the Sample Policy from the SELinux Targeted Policy 48
8.1 Query Results for the Low Privilege HPol Model 53
8.2 Query Results for the High Privilege HPol Model 53
8.3 Query Results for the Merged HPol Models 54

CHAPTER 1

Introduction, Research Problem, and Overview

This chapter introduces some of the key challenges that large enterprises face in the ongoing process of implementing and verifying information security policies in a highly-dynamic environment. It then lays out the research problem that is particular to the work in this thesis, namely, how to construct a high-level formal model out of low-level Security-Enhanced Linux mandatory access control policy configurations. It concludes with an overview of the thesis content detailing the approach and solutions used to construct this thesis work, ideas for future work, and conclusions implied by the work.

1.1 Introduction

The evolution of information technology poses a continual stream of security-related challenges for large enterprises. Enterprises need to balance these challenges while still remaining productive and operational. At present, there is a clear need for a paradigm shift in the way in which these organizations manage information security. Traditionally, information security concerns were thought to lie only within the realm of the technical means that practitioners routinely implemented to mitigate vulnerabilities within an organization's network. Now, the burgeoning amount and capability of connected devices necessitates a higher-level, more meaningful engagement with the process on the part of an organization's managerial and administrative staff. This will inevitably involve many employees who lack domain-specific knowledge of the technologies put into effect to secure the organization's network [2].

The need for a higher-level of engagement exists largely in part due to today's distributed network protocols and the technological environment within which sensitive resources are more accessible than ever. To exert control over these resources, analysts must design and implement policies that ensure that an enterprise's resources are not subject to any unauthorized accesses [3]. Policies are high-level requirements that specify the nature of how an

access is managed in terms of who may access a resource and in what manner they may do so [4]. Specifying and implementing polices is not a one-time event. Once implemented, policies must be actively maintained, which is a complicated process unto itself–as organizations grow, the effects of altering individual policies become more subtle, making their overall effect difficult to discern. Therefore, there is a need for security researchers to develop solutions that readily demonstrate the effect of altering policies in order to make the consequences of the changes evident to those who have to administer and manage them. Accordingly, there is a need for administrators to have a means to understand which policies are implemented within their system and what their overall effect is from a high-level vantage point [3].

One key factor that makes efforts to develop holistic policy mapping solutions in large-scaled networked systems such a daunting task lies in the fact that the inner workings of disparate system components such as routers, firewalls, and host-based operating systems are diverse in scope and, accordingly, at the device level, they must be configured in different ways from each other. The configurations and the associated data garnered from each class of component is outputted into a format specific to that class, hampering efforts to efficiently analyze unified policy enforcement across the components that comprise a system [5]. An effective solution to this problem is to create a *model*, a formal representation of the theoretical boundaries of a system given the set of security policies it employs [4]. Models cut down on the complexity within a large-scale networked environment by creating an abstraction of the system's various device configurations and a set of relevant potential interactions that may occur between the devices [5].

Mandatory Access Control (MAC) is a widely-implemented access control scheme that helps exhibit the complexities of policy implementation and analysis. When implemented properly, MAC is effective at prohibiting unauthorized accesses of system resources. However, it is difficult to properly specify and maintain MAC policies. This is due to the fact that, although MAC policies can span multiple applications and system components, no standard that exists to specify them at this time. As such, MAC policies are prone to faults that

are a result of misconfiguration, creating major vulnerabilities for the systems that they are implemented within. These issues are then rapidly compounded as the system expands [4].

1.2 Research Problem

Accounting for the issues outlined above, the work in this thesis tackles the problem of formally modeling low-level system configurations into a high-level formal policy model. The focus lies on the problem of populating a useful model for an access control policy scheme that is actually applied to the technologies that comprise a large enterprise. In particular, it explores how this task can be carried out across an access control scheme enforced by the Security-Enhanced Linux mandatory access control mechanism commonly included and used in conjunction with enterprise Linux distributions such as Fedora Red Hat Linux and the open source CentOS Linux distribution.

Work in this thesis also seeks to address how to go about using the populated high-level formal model in a useful and intuitive way, aiming to aid an organization's managerial staff in analyzing implemented policies effectively and thus to narrow the information security risks associated with employing many devices and policies within a modern, large-scale, enterprise networked environment.

1.3 Overview

The remainder of this thesis is organized as follows: Chapter 2, *Background* provides background information for various access control schemes and the Security-Enhanced Linux mechanism used to enforce or augment them. Chapter 3, *The Hierarchical Policy Model (HPol) and the High-level, Easily Reconfigurable Machine Environment Specification (HERMES) High-Level Policy Description Language* is an exposition of the work researchers have already accomplished with the Hierarchical Policy Model and HERMES, its associated high-level policy description language. Chapter 4, *Related Work* outlines other projects that

cover the same scope as the HPol library and the parser and query script demonstrated within this thesis document. Chapter 5, *Research Questions* expands on the research problem focused on by this thesis and breaks the problem down into three components, each of which represents a contribution of this thesis. Chapter 6, *Formalizing Enterprise Linux Mandatory Access Control Policies Using HPol* provides details and results derived from translating SELinux configurations into HPol models. Chapter 7, *Contribution 2: Interactively Answering Queries About SELinux Device Policies* details the process of interactively querying SELinux HPol models once they have been constructed. Chapter 8, *Contribution 3: Policy Merging for Multiple Device Policy Violation Detection* details this thesis project's addition to the HPol library that allows for merging and verification of multiple HPol policies. Chapter 9, *Future Work*, provides multiple avenues for future research on the efforts detailed herein. Finally, Chapter 10, *Summary and Conclusions* summarizes the content of the thesis, details possibilities for the future work it gives rise to, and provides a brief conclusion. A bibliography follows.

CHAPTER 2

Background

This chapter describes discretionary, mandatory, role-based access control, and multilevel security: common access control schemes that enterprises use to shape their security policies. It also covers Security-Enhanced Linux, an operating system-layer mechanism which allows administrators to implement and enforce access control schemes. This will provide the necessary background information to understand the systematic context that the contributions of this thesis are modeled and verified within.

2.1 Access Control

According to the Internet Security Glossary, the term **information security** covers "[m]easures that implement and assure security services in information systems, including computer systems". Along the same lines, and viewed from a slightly lower level, **computer security** requires "measurements to implement and assure security services in a computer system, particularly those that assure access control service" [6]. Likewise, Stallings and Brown [7] state that access control is the central focus of computer security. Access control systems are comprised of mechanisms that sit between users and computing resources with the aim to determine whether certain interactions are permitted between the two. Thus, the access control policies that are specified for these systems specify who can use what computing resources in what manner. The goal of access control measures is to allow for designated users to access organizational resources in accordance with their authorized actions while simultaneously denying unauthorized uses of resources by both malicious and legitimate users. These systems work in tandem to allow smooth operation for business functions while helping to assure that organizational information security goals are met [7].

There are a variety of access control schemes available to define an organization's high-level security policy. Common access control schemes include discretionary, mandatory, role-

based access control and multilevel security. Though they vary in their focus and scope, they share many of the same basic characteristics. In particular, these schemes are all comprised of *subjects*, *objects*, and *actions* (also referred to as *access rights*). Subjects in a policy include a system's users or user processes (e.g. "Alice", `/bin/passwd`, or an administrator role). Objects are the resources that may be accessed by subjects (e.g. directories, databases, and network appliances). Actions are the access right attributes associated with subjects that are used to determine which interactions are permitted between subjects and objects (e.g. read, write, or execute). Thus, an example of a single access control policy would be the specification that a subject, an employee named "Bob", has the access right to execute files located in the directory `/bin/`, the object being acted upon [7].

Under discretionary access control (DAC), the ability to alter permissions to act upon an object are granted to specific users, groups, and user processes by resource "owners" [8]. For example, in a traditional Linux environment, an administrator can grant one subject "read" access on a particular file, but not the ability to write to the file. Thus, under this scheme, owners use their discretion to provide access rights to other subjects at will. This type of scheme can work well for smaller systems with a relatively small amount of subjects, but the task becomes more daunting as organizations grow, especially if the organization's operations require more finely-grained access rights (e.g. cordoning off data in accordance to a certain trust level) [7]. Despite its limitations, DAC serves alongside other schemes to provide another layer of overall security [8].

The shortcomings of DAC are particularly evident when viewed in light of the process of implementing a high-level, administrator-defined policy. Since many permissions are at the discretion of individual users, exerting control in an over-arching, system-wide fashion is nearly impossible. DAC relies only on user identity and ownership, to the exclusion of other vital information including the user's role in the system, whether or not an application a user is going to run is trusted, and how sensitive the data is that a user is trying to access. Further, each process a user initiates inherits all of his or her same permissions, creating a

major vulnerability if user processes get compromised by unauthorized users. If the process is malicious, it has the same discretionary ability as the user, and thus may interfere with and take control of other processes (e.g. in the case of a privilege escalation attack). This level of uncertainty and lack of control runs counter to the goal of establishing a high-level, hierarchical organizational security policy. To better protect a system, an access control scheme needs to account for as much context-specific information as is feasible [9].

Unlike DAC, mandatory access control (MAC) is not centered around ownership and prohibits any subject's ability to have or grant arbitrary access rights to files or processes. If the intended policy is such that a particular subject may alter access rights, it is a mandatory requirement that he or she must be explicitly granted that privilege [7]. In the strictest application of MAC, MAC policies require that each potential action that a subject can perform on an object must be explicitly specified if that action is to be permitted, otherwise it will be denied by default. However, this is not practicable in a large-scale operational environment. Typically, a more flexible MAC or DAC policy is enacted so that innocuous, routine tasks can be carried out with minimal interference from the mechanism [10].

Role-based access control (RBAC) schemes define a set of rules that lay out a series of potential actions between subjects and objects and associate them with a "role". Attempted accesses for privileged functions are then only granted by an RBAC mechanism then when they match up with the user's role within the organization and the system [7]. For example, if an RBAC system were employed in a bank, a user with the role of "teller" could be granted the right to process deposits, pay out withdrawals, and furnish account balances, but not the right to view loan applications, an access belonging to an employee associated with the "loan officer" role. Thus, privileged accesses are segregated to trusted roles.

Another major access control scheme, multilevel security (MLS), is especially useful for organizations that are responsible for managing data of varying degrees of sensitivity, such as governmental agencies that manage data that are labeled with security clearances (e.g. Secret, Top Secret, etc.). With MLS, subjects may only act on objects with the same or

lower level of classification. Thus, each subject, action, and object are associated with a particular classification. Then, the access control mechanism must only grant the access to data of a particular classification if the different entities involved required by the interaction possess the same (or higher) level of classification. For example, if a government agency that is responsible for classified and unclassified data and has an MLS-enforcing mechanism in place, the ability to read "Secret" data will only be granted to a subject whose clearance level is "Secret" or higher [8].

A classic multilevel access control scheme is the Bell-LaPadula model outlined in [11]. In order to ensure that certain resource objects are kept confidential in a Bell-LaPadula implementation, they are assigned a privilege level. Subjects in this model then can access resource objects either in accordance with their own privilege level or one of those below it. They cannot access any resource objects assigned with higher privilege levels, but they may write to them. Likewise, they also may not write down to a lower privilege level. In short, under Bell-LaPadula, in addition to accesses associated with their specified privilege level, the subject may also "write up" or "read down" on resources with other privilege level designations [11].

Correctly specifying access control policies is a complex task because of the dynamic nature of organizations and systems. Individuals, components, and uses of a system change throughout an organization's life cycle making it a daunting process to keep track of these changes. Further, system and enterprise-specific information needs to be incorporated into the policy design process. Policy designers must understand what information is available to them and decide on how much of it they need to incorporate. They also need to understand typical usage of their systems so they can best craft a policy that will not unduly inhibit user interactions with the system that do not pose a high probability of a vulnerability.

The research problem that this thesis work considers is focused on high-level, device-independent modeling of low-level mandatory access control policies. However, RBAC is simulated in the model to display the associations between user roles and subjects in the

system. Also, a Bell-LaPadula MLS simulation is also provided to illustrate the ability to merge policies from multiple MAC devices. Discretionary access control provides the basis for understanding access control, especially as it pertains to file system permissions, and is also an important policy factor consideration for enterprise Linux distributions. Chapter 9 covers how future iterations of this work can incorporate a DAC dimension when modeling enterprise Linux policies.

2.2 Security-Enhanced Linux (SELinux)

After specifying a written, high-level security policy and choosing one or more access control schemes, organizations still need to find the best means of actually enforcing the policy in an operational setting. One robust solution for enterprise Linux environments is Security-Enhanced Linux (SELinux), a flexible, open source mandatory access control mechanism residing in the Linux kernel [8]. SELinux is enabled by default on Red Hat Enterprise Linux and the open-source CentOS Linux distribution [9] [12].

SELinux has three basic states: *enforcing*, *permissive*, and *disabled*. When SELinux is enforcing, all the policy modules it has loaded are put into effect. In a permissive state, SELinux allows all accesses (as long as they are allowed by the operating system and any other access control mechanisms), but keeps a log of all interactions that are denied in accordance with the SELinux policy. When SELinux is disabled, all access rights on the operating system rely on traditional Linux DAC [9]. When SElinux is enforced, all interactions with files or processes are intercepted by SELinux Linux Security Module in the kernel [8]. There, access decisions are made based on the SELinux policy placed into effect. Accordingly, SELinux does not rely only on user identity and ownership, its policy can be applied on resources down to the independent object level within the Linux kernel itself [9].

2.3 SELinux Policies and Type Enforcement

By operating under the notion that the best way to secure an operating system is to secure as many components as possible, SELinux augments application-level security mechanisms that are already in place. At its core, it is a mandatory access control mechanism, but also has the ability to allow administrators to implement role-based and multilevel security access control schemes. Additionally, it enforces policies alongside, and independent of, a traditional Linux discretionary access control configuration (e.g. one with file owners, permissions, etc) [8]. Whenever an interaction with the operating system is initiated by a subject, it must first be permitted, if at all, by Linux-based DAC. If the action is permitted, then it can and must be vetted by SELinux [9]. Since SELinux policies and Linux DAC are employed in tandem, it is possible for an access to be allowed under one mechanism, but not the other [8].

SELinux policies may be *strict* or *targeted*. Strict policies are a complete application of mandatory access control. As such, every possible interaction between subjects and objects must be explicitly specified in order to be allowed. For most organizations, this level of specification is excessive. Thus, employing the targeted policy is often a more balanced solution. The targeted policy protects key processes, while keeping interference with the user's experience to a minimum [12]. It allows most subjects and objects to run under traditional Linux discretionary policies, but targets the behavior of specific daemon processes that are known to be particularly vulnerable to attack. The goal of this policy is to confine the behavior of an exploited daemon to a minimal domain to prevent contagion to the entire operating system. This is the standard Fedora SELinux policy and is supported by Red Hat [10]. Non-targeted processes may run in the "unconfined" domain which lightens the restrictions placed upon them [12].

While strict, multilevel security capabilities are useful for assuring that data is kept confidential, they can be too simplistic for common use cases. For example, in many organizations, subjects will generally be attempting innocuous accesses (e.g. reading their email), regard-

less of privilege level. For this reason, SELinux provides a more flexible and finely-grained mandatory access control mechanism to better fit different organizational needs while incorporating as much system context information as is deemed necessary by the organization's principals themselves. To implement this capability, SELinux relies on *type enforcement*. Under type enforcement, all of the operating system's resources and capabilities are abstracted into SELinux *types*. For example, *bin_t* would be the type representing the `/bin/` directory and *ls_exec_t* would be the type representing the ability to execute the `ls` program. Types are independent of standard kernel operation, so organizations can specify an *SELinux policy* to specify how subjects, action, and objects may interact in accordance with their types or attributes (associations of types) without having to consider Linux kernel architecture [8].

When user processes in the operating system are labeled with a type, the type also defines a domain for that process to run within. Under SELinux, each process runs in a separate domain to allow the mechanism to have finer control over the capabilities for each process. When one process is running in its domain, no other processes may have access to the files it is accessing without a specific provision in the policy that allows them to do so. In addition, by default, SELinux denies processes from accessing other processes, unless there is a rule that specifically permits them to. Separating processes in accordance with their domain is key to avoiding privilege escalation attacks. It also limits the scope of the damage to the overall operating system if one process does happen to be compromised maliciously. If a malicious process tries to access another process, but no SELinux policy rules allow for the interaction, the malicious process cannot assume that attack vector [9].

2.4 SELinux "Access Vector" Rules

Most allowable interactions between different SELinux types are enshrouded within its policy configuration in the form of SELinux *access vector* (AV) rules. AV rules include: `allow`, which specifies how two types may interact, `auditallow` and `dontaudit`, which specify parameters about how events are logged, and `neverallow`, which specifies what interactions

are not permitted between types. The most common AV rule is the `allow` rule. The `neverallow` rule is seldom used. Allow rules are likely the most common because `auditallow` and `dontaudit` focus on aspects of logging. Few `neverallow` rules are included in the targeted policy. Actions that are not laid out in an `allow` rule are denied by default [8].

Constraints, which are defined by boolean expressions, can also be applied within an SELinux policy to nullify specified allowed interactions under different circumstances. Multilayer security constraints may also be added to bolster system security [8]. For the purpose of a general modeling of SELinux mandatory access control policies, this thesis assumes that accesses that are not specifically laid out in an allow rule are not allowed. Constraints and MLS are likewise not accounted for in this work, but are simulated in the forthcoming Bell-LaPadua example in Chapter 8.

SELinux AV rules take the form:

`<AV Rule> <Source> <Target> : <Object Class> <Permission(s)>;` [8]

where the sources and targets are SELinux types, the object class is the type of object being acted on (e.g. a file) and the permissions are the interactions allowed between the source, target, and object.

Thus, for example, the allow rule:

`allow user_t bin_t : dir {read execute getattr};` [8]

allows Linux user processes associated with the SELinux `user` type to read, execute, and get system attributes for files associated with the SELinux `bin` type [8].

A typical SELinux policy may include thousands of types so that it can represent all the resources found within the operating system. Since accesses are denied by default in SELinux, every possible action between two types intended by the high-level policy would each require its own allow rule. This creates the possibility of having to specify hundreds of allow rules just to allow a simple interaction to occur. This would make for a verbose and unwieldy security policy that would be difficult to manage and analyze. To simplify this process, SELinux allows types to be assumed into *attributes*, collections of types that may

be incorporated into allow rules. Types can be included in more than one attribute, as is required by the intended policy [8].

Mayer [8] provides an example of a situation where attributes can be useful when used within allow rules. In this example, there is an application that can back up the entire file system. In SELinux, this application is labeled with the type `backup_t`. Since an operating system has many files, `backup_t` would need an allow rule to have read access for each type associated with a file. Instead, by using attributes, all files are associated with an SELinux attribute, `file_type`. Then, the following allow rule may be written and enforced:

`allow backup_t file_type : file read;` [8]

This rule provides the application with read access to all files associated with the `file_type` attribute. This cuts down on the need to specify many allow rules, and makes it easier for an analyst to manage the policy. If an analyst needs to drill down further in order to understand what types are associated with an attribute, SELinux provides the command line tool `seinfo` which makes the association clearer to discern [8].

The specification of types, attributes, and access vector rules provides for implementing more flexible access control policies than using standard Linux discretionary access control policies alone. By assigning types to files and processes, rules can be written to specifically delineate what interactions are permitted between system users and objects. This takes away the ability for users to arbitrarily assign permissions to resources and makes it easier to implement system-wide, administrator-defined high-level policies.

2.5 SELinux Domain Transitions

In some cases, untrusted types need access to types associated with privileged processes. For example, if a user wishes to change his or her Linux user password, he or she will have to execute the Linux `passwd` program which requires access to files owned by the root user. To allow the user's type purpose-specific, limited interaction with the privileged process, SELinux **domain transitions** can be placed within `allow` rules to enable this access. The

following steps demonstrate this domain transition and are illustrated in Mayer [8]:

The first rule is a straightforward `allow` rule allowing a `user` type, `user_t`, the ability to execute using the `passwd` execute type, `passwd_exec_t` (note that a the `passwd_exec_t` type has been specifically defined for this type of interaction):

```
allow user_t passwd_exec_t : file {getattr execute};
```
[8]

Then, the following rule is used to create an "entrypoint" between the `passwd` executable and its associated executable type which allows the executable type enter `passwd`'s domain:

```
allow passwd_t passwd_exec_t : file entrypoint;
```
[8]

This final rule grants the user's type the permission to enter `passwd`'s domain:

```
allow user_t passwd_t : process transition;
```
[8]

When considered together, the three rules then allow the user to enter the privileged domain, execute *passwd*, and change their Linux user password.

Role-based access control is also supported by SELinux. The heart of SELinux is still type enforcement, but roles may be simply laid on top of type enforcement by associating roles with types. For example, the SELinux association:

```
role user_r types user_t;
```
[8]

will allow a subject belonging to the `user` role all accesses provided to the `user` type, `user_t` [8]. Thus, for example, an administrator role could be created and associated with sensitive tasks that an enterprise's non-administrative employees would not need to carry out.

CHAPTER 3

The Hierarchical Policy Model (HPol) and the High-level, Easily Reconfigurable Machine Environment Specification (HERMES) High-Level Policy Description Language

This chapter introduces the HPol project and outlines the progress University of Idaho researchers have already achieved up until this point with the modeling of various exemplar systems. Also included is the introduction of HERMES, the high-level policy description language that expresses HPol output and queries [13] [14].

3.1 The Hierarchical Policy Model

The goal for access control policies and holistic organizational security policies are the same: to determine what users may access which resources in what manner. Policies implemented using device-level mechanisms require technical resources to ensure that the rules laid out in the enterprise's high-level policy are properly enforced. This process requires time, resources, and device-specific technical expertise. However, these requirements can be impossible to meet and difficult to comprehend for managers who need useful, accurate information on a timely basis. Beyond enforcement, the dynamic nature of the cybersecurity problem space also makes it vital to speed up this process. The Hierarchical Policy Model (HPol) offers a higher-level of abstraction that serves to provide management with the ability to verify different system interactions and receive sensible, intuitive output, for high-level organizational policies that are intended to be enforced across multiple devices within an enterprise's network [15].

HPol is a framework and associated tool set that allows for formal modeling, verification, and visualization of system security policies. It serves to verify that an organization's high-level security policy (i.e. the *intended policy*) is implemented as intended in lower-level

system components where policies are typically and actually enforced (i.e. the *implemented policy*). HPol was developed at the University of Idaho Center for Secure and Dependable Systems and is part of an ongoing research effort [15].

System configuration-to-HPol model translations have been carried out across three "exemplar" systems in order to demonstrate its usefulness and extensibility. These include the OpenStack Cinder distributed storage system, a Cisco router-based virtual private network tunneling interaction [16], and the SELinux mandatory access control mechanism detailed herein [15]. The HPol project includes parsers to translate all of these exemplars' configurations into high-level models and visual graphs.

Formal policy models need to represent a system's behavior as accurately as possible. However, there is a trade-off between the simplicity of the model and the ability to represent specific security goals for the system. Therefore, security goals and models need to be carefully crafted such that the model may be analyzed to ensure that the goals are met while still faithfully maintaining the system's semantic characteristics within the model [5].

To best map out a system component's characteristics, HPol policy models are built from the ground up. The library uses elements from low-level configurations (e.g. those that control a router or those specified in an SELinux policy) to populate a model of hierarchical system security policies with little loss of integrity during the translation. Once the formal model is constructed, its output may be saved to a PyGraphViz [17] `.dot` file for visualization in various purpose-specific graph viewers and/or in a high level policy description language as detailed in Section 3.2 below. Formal operations and querying may also be applied to the model. For example, this thesis demonstrates an effective use of *policy merging* to fuse disparate policies from disparate SELinux machines. Once the model is established, an analyst can verify if a policy is implemented correctly by constructing a query to determine whether a certain action would be permitted between subjects and objects found across the merged policy models [15].

The group of researchers initially responsible for developing HPol determined that the

best way to display policy data groupings was to model them with directed acyclic graphs (DAGs) that grow vertically and policy links that populate them horizontally. Accordingly, a component-to-HPol parser begins by first constructing an overall HPol DAG comprised of three smaller DAGs: an individual DAG each for subjects, actions, and objects. Subjects, actions, and objects of a particular component are then populated in accordance with the researcher's interpretation of how the peculiar features and use cases for that system fit within the HPol paradigm. Then, in a similar manner, policy links are drawn across the model to indicate complete, distinct allowable interactions. All policies begin with a master HPol start node and end with a master HPol end node. Wildcard links (i.e. "all" links) not associated with specific, named policies are allowed as well [15].

Figure 3.1 provides a sample graph of a basic HPol model. In this model, the Subjects DAG is populated with a Linux user, Alice. The Objects and Actions DAGs are populated with file system objects and interactions. Policies begin with the `HPolStart` node and terminate with `HPolEnd` node that are both independent of the DAGs. The policies indicated by links `1001` and `1002` demonstrate that Alice may read and write on objects associated with the `/home/alice` directory. Thus, each potential interaction between subjects and objects is mapped within a single policy. However, in an implementation it would be possible, and sometimes useful, for the researcher to make a single node serve as an abstraction of many nodes, but must keep in mind that they are portraying the information in a way that is most suitable to the particular context in which they are modeling.

3.2 High-level, Easily Reconfigurable Machine Environment Specification (HERMES) Output of HPol Models

The HPol library includes a method that will output policy information in the High-level, Easily Reconfigurable Machine Environment Specification (HERMES) format. HERMES is a high-level policy description language developed at the Center for Secure and Dependable

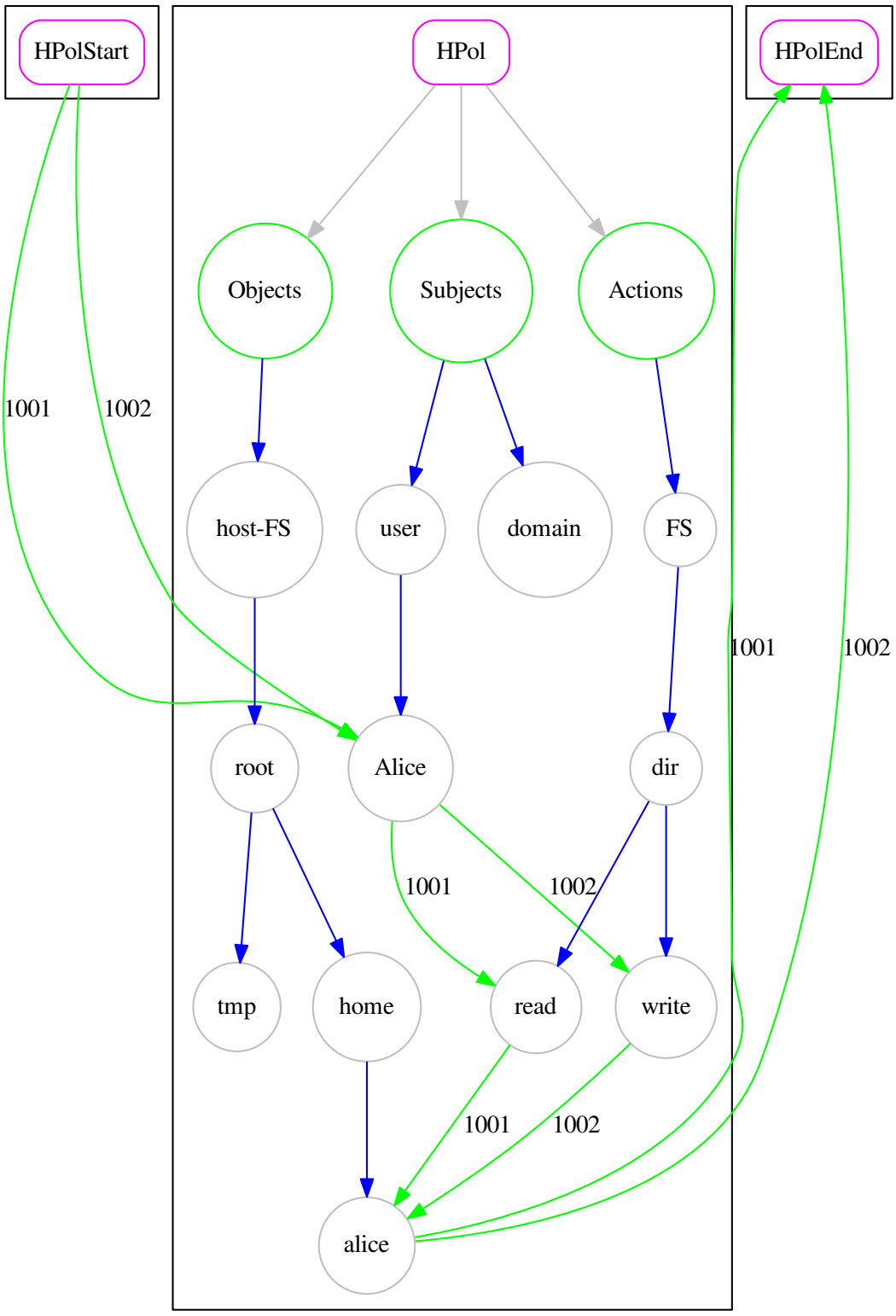

Figure 3.1: HPol Model: Alice Read/Write Example

Systems at the University of Idaho, for the initial purpose of describing high-level web browser policy specifications [13] [14]. HERMES provides a clear format that an analyst may use to see what components and policies make up different systems and policies, including HPol policies. Once an HPol model's policies are encoded in HERMES output, XSB Prolog modules developed at the Center for Secure and Dependable Systems may be used to query the output to aid in policy verification. The abbreviated HERMES output found in Listing 3.1 details policies found within the example HPol model generated in Figure 3.1. Listing 3.2 displays a HERMES representation of a portion of the structure of the same model. Once generated, this HERMES output can be read directly by special parsers using XSB Prolog.

3.3 HERMES Queries with XSB Prolog

In order to query an HPol model using HERMES output and XSB Prolog, the following workflow is necessary:

1. Generate an HPol model and use the library function `convert2hermes()`.

   This method writes node and policy information to a file called `hermes.out`.

2. Open an interactive window for XSB Prolog.

3. Type `[main].` and press return.

   This loads the HERMES output parser.

4. Type `parse_and_convert(hermes.out).` and press return.

   This parses the HPol HERMES output into a format that can be used by the XSB Prolog evaluator associated with this project.

5. Open a text editor and generate the desired query in HPol format.

   Create a file called `query.herm` and enter the query in the form found in Listing 3.3.

6. Return to the XSB Prolog interactive window, type `parse_and_convert(query.herm)` and press return.

7. Type `[evaluate].` and press return.

   This loads the module XSB Prolog uses to evaluate the parsed output from `hermes.out` and `query.herm`.

8. Type `eval_query(A, B).` and press return.

   This method evaluates the query generated in HERMES. It can run one query at a time. Pressing return will result in either `yes` for a successful query or a message reading `Query failed.`

9. Type `halt.` and press enter.

   This terminates the XSB Prolog interactive session.

While none of these nine steps is difficult to apply, it would be cumbersome for an analyst to walk-through all of them for each query. They would also need to possess HERMES and XSB Prolog technical knowledge to complete the query. To simplify this process, an interactive script to automatically generate a query and execute the XSB Prolog portion of the query makes up the contribution detailed in Chapter 7.

Listing 3.1: HERMES Policy Output for the Example HPol Model

```
1
2       Policy: p1001
3       {
4       Description: ``HPol Policy'';
5       Status: Enabled;
6       Path: [HPolStart, Alice, read, alice, HPolEnd];
7       }
8
9       Policy: p1002
10      {
11      Description: ``HPol Policy'';
12      Status: Enabled;
13      Path: [HPolStart, Alice, write, alice, HPolEnd];
14      }
```

Listing 3.2: HERMES Node Output for the Example HPol Model

```
1
2       Node: dir
3       {
4       Description: ``dir";
5       Path: ``HPol/Actions/FS/dir";
6       ID: ``3028190526064582664088867455944794154599
7            907038349986933090158624410117085 4817";
8       Type: Actions;
9       Children: [read, write];
10      }
```

Listing 3.3: HERMES Query of the Example HPol Model

```
1
2       Query: 1
3       {
4       Allow: yes;
5       Contains:[Alice, read, alice];
6       }
```

CHAPTER 4

Related Work

This chapter details work by other researchers within a similar scope as this thesis. Other policy models and other work involving SELinux policies, as well as brief comparisons between them and HPol, are included.

4.1 Other Policy Modeling and SELinux-Related Projects

Guttman and Herzog [5] applied "rigorous automated network security management" to model a small network in order to verify policies relating to distributed packet distribution and filtering and, additionally, to IP Security protocol enforcement. Their model incorporated an undirected and bipartite graph to represent the flow of packets back-and-forth across its nodes. Its nodes were comprised of network devices and locations and its edges were comprised of packet filtering interfaces. Security goals within this model included restricting the flow of specific packets through the network and ensuring the authenticity and confidentiality of other packets that passed filtering. They represented behavior as a function of various system configurations and the output of the model consisted of algorithms that could predict possible consequences of different system configurations. Once these algorithms were obtained, administrators could write a program to verify that security goals were met, to provide examples of which ones were not met, or even to generate a set of optimal configurations for the network.

Since network devices made up the operative core of Guttman and Herzog's model, their approach varies materially from HPol modeling. For instance, they employed an undirected graph to model packets flowing back-and-forth between nodes. They found that this did not scale to networks including more than a trivial amount of nodes [5]. HPol's directed acyclic graphs and policy links may be comprised of thousands of nodes and edges and thus can already scale to larger representations of systems. In addition, the end results of their model

were predictive algorithms, leaving administrators to then craft a program to assume and interpret them on their own. The HPol model provides HERMES output which represents policies that can be directly queried. Unlike Guttman and Herzog's model, the interactive policy querying contribution of this thesis demonstrates a program which may be used to query results and free administrators from this requirement. Both approaches construct formal models that may be used against desired predicates prior to the problem-solving stage, making them highly-efficient.

Guttman, Herzog, et al. [18] also formalized a subset of SELinux to determine information flow security goals. In this work, they modeled covert security channels on an SELinux web server. As in [5], this approach can verify security goals by checking them against the formal model. This model's approach differs from the HPol approach because this work is using SELinux to model mandatory access control overall. Conversely, a general purpose model for mandatory access control has been constructed that does not explicitly cover SELinux, but it provides an abstraction of system components, rather than directly mapping to the components themselves [4].

Another noteworthy SELinux policy effort is Lobster, a domain-specific language for describing SELinux policies [1]. The researchers who created Lobster note that SELinux policies can be understood as information flows between domains. The goal of Lobster is to aid policy designers in confirming whether the information flows laid out in their intended policy match up with how they are implemented within SELinux policies. Ultimately, Lobster is supposed to compile SELinux policy statements *from* Lobster thus generating the SELinux configuration itself. An example Lobster policy appears in Listing 4.1.

Listing 4.1: Sample SELinux Policy Statements Written in Lobster [1]

```
domain p = Process();
domain f = File(``/tmp/file'');
p.active -- f.read;
```

The rule written in Lobster specifies the SELinux types `p_t` and `f_t` and allows `p_t` to read a file of type `f_t`. The Lobster compiler breaks this down into the following SELinux allow rule:

`allow p_t f_t : file read;` [1]

No current HPol examplars actually create the configurations from the high-level policy in the format of their own paradigm.

One more policy model to consider is Margrave, a "general-purpose policy analyzer" that has been applied to different exemplars including role-based access control [3] and firewall configuration [19]. In terms of role-based access control, Margrave processes system configurations and outputs them into XACML, an XML-derivative, to express access control policies. Margrave consists of two major components: (1) a verification system that accepts a policy and a property to determine whether they match up (i.e. a query) and (2) a system for "change-impact analysis" that can take multiple policies as input and summarize the differences between them. It also includes support for policy and query combinations. For efficiency, Margrave does not explicitly represent a system's data, but uses uninterpreted symbols to simulate reasoning about its exemplar's data [3].

Margrave and HPol vary in terms of how they ingest access control data because HPol explicitly incorporates data directly from its system exemplars whereas Margrave uses higher-level symbol abstraction. The HPol approach avoids as much information loss as possible when translating configuration information into a model. It also allows HPol to scale to a larger system than exhibited in [3]. HPol does not yet include capabilities for compound policy querying. Both models output policies into a markup representation that may be

queried, but the query mechanism detailed in this thesis allows administrators to skip the step of having to analyze the markup as it applies to the SELinux-to-HPol variant.

The efforts outlined in this chapter, as well as the work completed with HPol thus far, demonstrate that researchers have been successful in many cases at constructing and analyzing formal models using configurations from actual systems. However, the need still exists for a more developed holistic model to formalize system security policies *as implemented* such that they can be represented at a high-level abstraction that can be formally verified against the intended security policies. This need remains due to the countless permutations of possible network configurations and the additional system components these methods must be applied to.

CHAPTER 5

Research Question and Contributions

This chapter outlines the major question this research seeks to answer. It then breaks the main question down into three components which map directly to the three contributions of this thesis to the HPol project.

5.1 Research Question

The particular issues encountered when abstracting low-level policies into higher level models, along with the particularities of implementing HPol and its related tools, give rise to many interesting questions. Since this thesis is especially concerned with applying HPol to model and query SELinux policies, the central question is, necessarily:

Can a high-level formal model for policy verification be constructed from low-level SELinux mandatory access control policy configurations?

To answer this question in the affirmative, it is important to demonstrate that the HPol model can be applied to SELinux policies in a useful and effective manner, especially as it would be used by managers in a large enterprise.

5.2 Thesis Contributions

The crux of this thesis maintains that, for HPol to be successfully applied to an SELinux MAC policy, three requirements must be met: (1) the ability to translate SELinux MAC policies into an HPol model, (2) the ability to query that model, and (3) the ability to merge and query multiple SELinux-to-HPol models. The contributions of this thesis that address these requirements are outlined as follows:

1. **Formalizing Enterprise Linux Mandatory Access Control Policies Using HPol**

   This requirement is met in terms of an SELinux-to-HPol parser that constructs an HPol model congruent with other HPol models already described. First, an HPol model that incorporates SELinux subjects, actions, and objects into HPol directed acyclic graphs. Policy links must also be populated.

   Although technically, mandatory access control schemes need to have every possible class of access specified, most enterprises will lack the time and resources necessary to specify every possible access between subjects and objects. For this reason, the standard "targeted" SELinux policy is enabled by default, and then configured from there, if at all. Therefore, an SELinux-to-HPol parser needs to successfully parse the targeted policy with respect to mandatory access control.

2. **Interactively Answering Queries About SELinux Device Policies**

   Since one goal of HPol is to make it easier to make sense of disparate system component policies, it follows that querying the model should be a straight-forward process that does not require technical expertise for specific system components (e.g. SELinux access vector rule details) or the nine-step process outlined in Section 3.3. A front-end, interactive mechanism to query policies using only subject, actions, and object names makes up the second contribution, providing an intuitive way to verify the policies.

3. **Merging Policies for Multiple Device Policy Violation Detection**

   Another goal of HPol is to abstract policies derived from a variety system components (e.g. a router, a workstation, and a storage system). The first step to accomplish this is to merge policies that originate from the same system components. In terms of the SELinux-to-HPol translation, the merge operation combines two SELinux policies such that each model retains its particular attributes and such that the merged model may be visualized and queried in the same manner as a single model.

CHAPTER 6

Contribution 1: Formalizing Enterprise Linux Mandatory Access Control Policies Using HPol

The SELinux-to-HPol policy parser outlined in this chapter demonstrates that an HPol hierarchical policy model for an enterprise Linux mandatory access control policies can be constructed. This parser was written in Python 2.7.5 using the HPol library. It was deployed on CentOS Linux 7 (64-bit) with the Linux 3.10 kernel in a virtual machine with a quad-core CPU and 5GB of RAM.

In the first portion of this chapter, the base SELinux-to-HPol parser created for this thesis work will be introduced. Then, in first cast study, the normal parsing workflow will be walked through to demonstrate that it works in accordance with the sample model and queries found in Chapter 3. Once this basis is established, the second case study demonstrates a full parsing of the standard SELinux targeted policy.

The portion of nodes that make up the foundation of any SELinux-to-HPol policy (e.g. ObjectTypeAttr, SELinuxClassPerms, and RoleType) are manually-coded into all models. The DAG nodes comprising the example *passwd* policies and policy merging examples are hard-coded into the application. Above the base SELinux-to-HPol policy foundation, the DAGs for the targeted policy model outlined in Section 2.2 are populated programmatically using flat files. Actual policies for all models are parsed using code in the script. Role-based access control is a major component of SELinux, but is simulated herein to make querying more coherent. To simulate basic RBAC interactions within the model, each named policy begins with user "alice". Similarly, since most users in the targeted policy are associated with the `unconfined_t` type, it serves as the starting point for Subjects. The parser also includes the functionality to output mappings between policy link numbers and device names. This is useful for policy merging for multiple devices.

Allow rules are stored in `av_rules.txt` and used to populate the HPol model's policy

links. Listing 6.3 demonstrates how allow rules typically contain multiple permissions. To aid in providing the model with finer granularity, the SELinux-to-HPol parser populates separate policies for each permission in the rule. For example, the HPol model will have three separate policy links for a single AV rule with read, write, and execute permissions. Thus, the HPol representation of an SELinux implementation will contain more full policies than allow rules.

6.1 Case Study 1: Parsing of an SELinux Policy with Domain Transitions

To demonstrate the basic operation of the SELinux-to-HPol parser, this section revisits the SELinux `passwd` domain transition policy outlined in Chapter 3. Figure 6.1 provides the graph for this policy. This case study assumes that the targeted policy is active. The DAGs in this model were hard-coded into the script based on the SELinux-to-HPol semantics and the information contained within the relevant allow rules. This selection of allow rules was actually parsed in order to demonstrate the feasibility of translating this entire transaction.

Policy link creation for the policy's allow rules will be broken down piece-by-piece to provide the following policy path walk-through:

1. `allow unconfined_t passwd_exec_t : file {getattr execute};`

   This allow rule provides the user type the ability to perform the actions necessary to execute `passwd`. This ability is associated with the `passwd_exec_type`. The association between the two is modeled with named policies `1001` and `1002`. Label `(A)` demonstrates where the policies begin.

2. `allow passwd_t passwd_exec_t : file entrypoint;`

   Then, `passwd_exec_type` is associated to the `passwd` domain. Note that there is no direct link between the user's type and the privileged process itself. The association of

the *passwd* executive type with *passwd*'s domain is demonstrated in Figure 6.1 in the red "all" link located between `passwd_exec_t` and `passwd_t`. Label `(B)` shows where this file entrypoint occurs.

3. `allow unconfined_t passwd_t : process transition;`

   To complete the domain transition, named policies `1003` and `1004` show how the user type `unconfined_t` is allowed to enter the `passwd` domain in accordance with the actions permitted in the first allow rule in the set. Label `(C)` shows where these policies end.

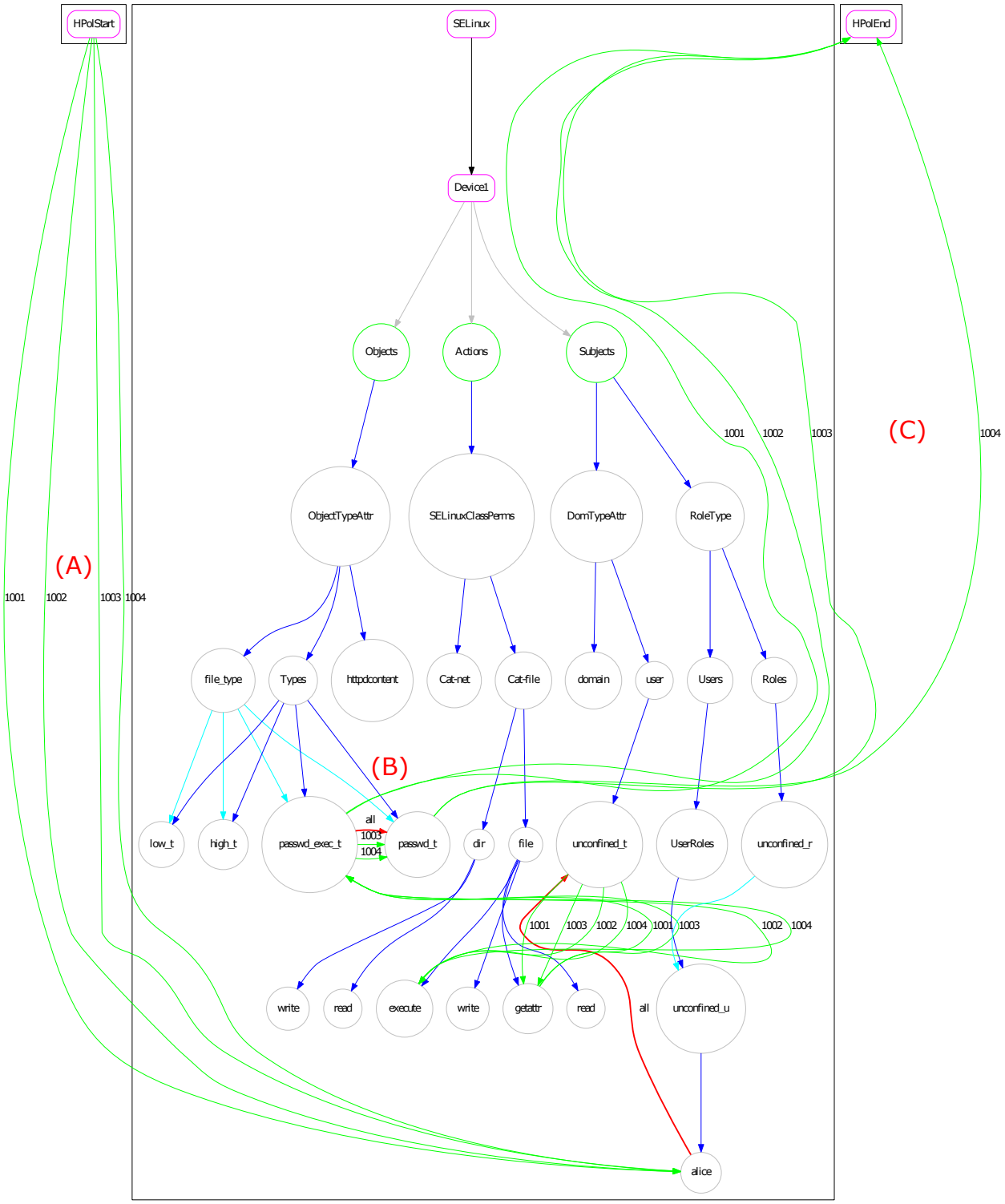

Figure 6.1: Abbreviated HPol Model Demonstrating the *passwd* Domain Transition

6.2 Case Study 1: Findings

Figure 6.1 shows an abbreviated, but fundamentally complete HPol model of an SELinux policy. Note on this figure that all types emanate from the `Types` node in the `Objects DAG` and later connect to their associated attribute, `file_type`. The SELinux DAGs were constructed in this manner for two reasons: 1) so all types have an absolute path to identify them and 2) because a type may be associated with multiple attributes. Thus, the secondary hierarchical links are added after the types are already populated. Roles and users are populated in a similar manner, but it is not readily evident here since this parser assumes that "alice" is the only user.

Once the HPol model's DAGs are properly populated, it is trivial to populate the bulk of the SELinux allow rules. New policies are created, provided with a policy number, and links are drawn through the relevant nodes. Thus, the first two policies that are a result of the first allow rule in the `passwd` domain transition are straight-forward to parse. For the accesses permitted between `unconfined_t` and `passwd_exec_t`, two links are added in addition to the policy's starting and ending links: one between the subject and the action and another between the action and the object acted on. This represents a full named policy because there is a complete subject-action-object interaction.

The file entrypoint and domain transition allow rules represent special cases for policy link creation and require additional logic to be incorporated into the model accurately. In both cases, they appear as regular allow rules representing a subject-action-object interaction since they do not appear differently than other allow rules. However, the process of the domain transition does not work in that way.

The file entrypoint rule does not, in and of itself, result in a named policy. Thus, when the parser encounters a file entrypoint in an allow rule, it does not create a new policy and does not connect the associated types with an endpoint. Instead, a single wildcard link is created between the two types listed. This link runs in the opposite direction as the policy

links already demonstrated. This is because the allow rule:

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

means, in this case: create a file entry point between the `passwd_exec_t` type and the `passwd_t` type. The link must be a wildcard link because domain transitions that possibly represent multiple interactions will need to create links for full policies that need to use the wildcard link multiple times.

The domain transition rule provides additional challenges because it contains little information about all the policies it implies:

```
allow unconfined_t passwd_t : process transition;
```

This process transition serves to allow all permitted policy interactions between `unconfined_t` and `passwd_t` to be allowed. The HPol model lends itself well to populating these policies: using internal HPol library methods, the parser can find all the links between the two types and create new policies for those. This is the reason that creating the file entrypoint between `passwd_exec_t` and `passwd_t` was important. Since the first two policies linking `unconfined_t` and `passwd_exec_t` were already populated, the file entrypoint provided a complete path between `unconfined_t` and `passwd_t`. This path is vital to represent the ability for the user to change their password using `passwd_t`.

Once the parser can find all the links between two types, it can create complete policies, even with the limited information provided by the process transition allow rules. To find the links, some basic graph querying functionality needed to be added to the HPol library itself. This is because HPol graphs are constructed internally using the `networkx` library [20] to construct directed graphs with multiple edges. The graph object for the HPol model may only be accessed via private methods within the library source itself in order to prevent graph manipulation and unwanted side-effects. Accordingly, the `isPathBetweenNodes()` and `getPathsBetweenNodes()` external methods were created for this thesis work within the library itself in order to accomplish the tasks necessary for this challenge. These methods were relatively straight-forward to implement because they take advantage of the application

programmer interface of the `networkx` library itself.

With the implementation of the node querying methods, it is possible to find out whether there are any paths connecting the two types delineated in the process transition. For each process transition, the parser includes logic to find out whether there is a path between the nodes. If so, it finds all the paths. All these paths should include a subject, action, and object. Then, for each path, a new policy is created, and all policy links can be populated.

In the case of the `passwd` domain transition illustrated here, the process would now be complete. This is because the whole transaction was parsed in the ideal order: the initial links were created first, then a file entry point was added, and then all possible paths were found. In the actual compilation of allow rules, however, the rules are not necessarily placed in this proper order. If, for example, the process transition were parsed before the rules allowing the desired accesses, no paths would be found, and no associated new policies would be created–even if they were specified elsewhere in the policy. To avoid this problem, this parser populates SELinux process transitions only after all other accesses and entry points have already been populated. To do this, whenever a process transition is being parsed, its associated information is stored in a dictionary to be used later. This dictionary uses the source of the rule as its key and the destination of the rule as its value. Then, once all other allow rules have been parsed, the dictionary is used to find all the policies allowed by the process transitions (e.g. all the paths between `HPolStart` and `HPolEnd` nodes between them using `networkx` methods) and processes them accordingly. In this case, the two paths were found were the ones used to populate policies `1003` and `1004`.

Once all the allow rules are properly parsed accounting for domain transitions, it is possible to derive the model's HERMES output. Listing 6.1 demonstrates the policy portion of the HERMES output for the `passwd` domain transition.

Policies `p1003` and `1004` demonstrate that the user type, `unconfined_t` may perform both actions specified in the allow rules indirectly with `passwd_t`. Thus, the domain transition steps taken within the parser and described in this section were a success.

Listing 6.1: HERMES Policy Output for the *passwd* Domain Transition

```
1
2          Policy: p1001
3          {
4          Description: ``HPol Policy";
5          Status: Enabled;
6          Path: [HPolStart, alice, unconfined_t, getattr,
              passwd_exec_t, HPolEnd];
7          }
8
9          Policy: p1002
10         {
11         Description: ``HPol Policy";
12         Status: Enabled;
13         Path: [HPolStart, alice, unconfined_t, execute,
              passwd_exec_t, HPolEnd];
14         }
15
16         Policy: p1003
17         {
18         Description: ``HPol Policy";
19         Status: Enabled;
20         Path: [HPolStart, alice, unconfined_t, getattr,
              passwd_exec_t, passwd_t, HPolEnd];
21         }
22
23         Policy: p1004
24         {
25         Description: ``HPol Policy";
26         Status: Enabled;
27         Path: [HPolStart, alice, unconfined_t, execute,
              passwd_exec_t, passwd_t, HPolEnd];
28         }
```

6.3 Case Study 2: Parsing the Default Red Hat Targeted SELinux Policy

Up to this point, only example HPol models for demonstrative purposes have been provided and analyzed. To prove the robustness of the SELinux-to-HPol parser, this case study shows

that the parser also works at scale when provided with a real-world policy covering an instance of an entire operating system's functions. The full policy that was parsed was the Red Hat targeted SELinux policy, which is the default policy for enterprise Linux distributions. Thus, the results from the parsing of the targeted policy provide a rough idea of what populating HPol models using large-scale SELinux policies might look like in an operational setting.

The full parsing does not differ materially from the abbreviated parsing of the smaller examples. The only difference is that it incorporates all nodes and allow rules from the SELinux `seinfo` and `sesearch` utilities. The DAGs were populated with the results from the subjects, roles, action, attributes, and types dictionaries emanating from files created within a policy directory. Specifically, the parser populates the HPol model using SELinux policies and components stored in flat (i.e. ".txt") files. This method necessarily means that contextual information for the device that hosts SELinux is lost along the way (e.g. the Linux DAC permissions). However, it does allow the parser to be used on different machines that are independent of the target machine. This is a reasonable trade-off because a goal of HPol is to free policy information from independent components to allow for systematic implementation and verification of security policies. An example of the flexibility that this approach allows is exhibited by the the ability to merge HPol models representing different devices as described in Chapter 8 which details the HPol policy merging contribution of this thesis.

SELinux policy components are compiled by running the `seinfo` command line argument. Running the command for specific components along with the `-x` option provides a tiered mapping between the component and the most relevant component it is associated with. For example, the following command and resulting output demonstrate the relationship between the `confined_admindomain` which includes types for sensitive processes entrusted to an admin role and the types themselves (see Listing 6.2).

Roles are provided by `seinfo -r`, users and their associated roles by `seinfo -u -x`,

Listing 6.2: Attribute and Type Output from the *seinfo* Utility

```
$ seinfo --attribute=confined_admindomain -x

confined_admindomain
logadm_t
secadm_t
sysadm_t
dbadm_t
auditadm_t
```

object classes and their associated permissions by `seinfo -c -x`, object attributes and their associated types by `seinfo -a -x`, types and their associated attributes by `seinfo -t -x`, and types by themselves with the command `seinfo -t`. Command output for these is stored in `roles.txt`, `user.txt`, `action.txt`, `object.txt`, `attr.txt` and `type.txt`, respectively.

Together, the files populated from `seinfo` represent a complete representation of all the resources within the operating system as well as all *potential* interactions for the purpose of constructing the hierarchical portion of the high-level HPol model from an SELinux configuration. These files can be placed in a directory which can then be easily loaded into the parser. From these files, the parser compiles these various resources and interaction representations into dictionary data structures where the key is the primary component and the associated component is the value. This makes it simple to populate the Subject, Action, and Object DAGs. The associations also aid in populating the policies, especially in the case of domain transitions which were described above.

To build a model using a complete SELinux policy, it is also necessary to incorporate all of the allowed accesses, in the form of allow rules, between the nodes in the DAGs. To compile all the SELinux policies for HPol, the `sesearch` command line tool provides all the SELinux access vector rules that have been specified in the system's policy modules. Since this work focuses only on SELinux allow rules, `sesearch -all` is ran to provide all allow rules for the SELinux implementation. For example, Listing 6.3 shows the following

Listing 6.3: Allow Rule Output from the *sesearch* Utility

```
$ sesearch -A -s confined_admindomain | head -5
Found 1627 semantic av rules:
allow sysadm_t security_t : security { compute_av
   compute_create check_context compute_relabel
   compute_user setsecparam read_policy } ;
allow sysadm_t security_t : filesystem getattr ;
allow sysadm_t security_t : file { ioctl read write
   getattr lock append open } ;
allow sysadm_t semanage_t : process transition ;
```

command and output providing the first five results of an *sesearch* execution for allow rules in which types belonging to the `confined_admindomain` attribute are the source. Note that, even though none of the source types are labeled `confined_admindomain`, these types are associated with it as shown in Listing 6.2. Although `confined_admindomain` may be used in place of the type itself, it then would represent many types having the specified accesses.

6.4 Case Study 2: Findings

This parser successfully incorporated all information included within the policy flat files. Listing 6.4 displays the statistics generated by the full parsing. In all, over 6,000 nodes were populated in the model's DAGs and over 2,000,000 links were created. The policy consisted of 95,604 allow rules broken down into 537,553 individual HPol policies. It took less than two hours to parse on a quad-core CentOS virtual machine with 5GB of RAM.

Up until this case study, the solution was complete for all file and process types. However, the need to parse SELinux *attributes* yielded additional considerations when parsing the targeted policy. SELinux attributes are able to represent many types as the `backup_t` example illustrated earlier:

```
allow backup_t file_type : file read;
```

The attribute `file_type` can be comprised of many types. In accordance with the reason

Listing 6.4: HPol Results from Parsing the SELinux Targeted Policy

```
populating subjects at:  21:01:06.978904
populating actions at:  21:01:07.060726
populating objects at:  21:01:08.789825
adding policy links at:  21:02:37.226885
adding transitions at: 22:40:25.704490

No paths found between sepgsql_ranged_proc_t and
   sepgsql_client_type or their associated attributes.

394 transitions attempted  393 were successful
537553 allow rules and components parsed
graphing model at  22:43:07.357749

SHOWING HPOLICY STATISTICS
Name: TargetedPolicy
Type: SELinux
NUMBER of NODES:
6192
NUMBER of EDGES:
2221768
finished at  22:43:18.239398
```

that SELinux has attributes to begin with, namely, simplicity, this parser also does not draw links through every possible interaction with a type, but through the attribute itself. This choice greatly simplifies the model while letting it remain congruent with the operation of SELinux. However, it complicates the search for paths between types. This is because one allow rule can specify that an attribute can act on other types, but another one may include a process transition that only names the type, but not its attribute. This means the process transition only applies to the type, but that, if a process transition between it and a privileged type is granted, it is able to carry out all actions that are associated with its attribute on the priveleged type.

The process transition code found in Figure 6.2 makes the process needed to account for attributes evident. These conditions apply to all key-value pairs of types and attributes that

```
foundPaths = []
if hpol.isPathBetweenNodes(fromTypePath, toTypePath):
        # e.g. between from_t and to_t
        foundPaths = hpol.getPathsBetweenNodes(fromTypePath,
           toTypePath)
elif hpol.isPathBetweenNodes(fromTypePath, toAttrPath):
        # e.g. between from_t and to_type
        foundPaths = hpol.getPathsBetweenNodes(fromTypePath,
           toAttrPath)
elif hpol.isPathBetweenNodes(fromAttrPath, toTypePath):
        # e.g. between from_type and to_t
        foundPaths = hpol.getPathsBetweenNodes(fromAttrPath,
           toTypePath)
else:
        failures+=1
        print 'No paths found between', key,'and',proc_trans[key
           ], 'or their associated attributes.'
```

Figure 6.2: SELinux-to-HPol Parser Code Demonstrating Path Finding for Domain Transitions

were stored in the process transition dictionary during the first round of AV rule parsing. The first conditional takes care of the case where there is a process transition, but no attributes are involved. The HPol method `getPathBetweenNodes()` uses `networkx` to find all paths between the SELinux types. The result, stored in `foundPaths` is then later used to populate additional full policies into the model. The second conditional covers the case in which the source of the allow rule is a type, but the target is an attribute. This works the same as the first conditional, except now all links between the single type and its target attribute

Listing 6.5: An Extraction of HERMES Policy Output from Parsing the SELinux Targeted Policy

```
1
2           Policy: p1021
3           {
4           Description: ``HPol Policy";
5           Status: Enabled;
6           Path: [HPolStart, l2tpd_t, read, l2tpd_var_run_t, HPolEnd
                ];
7           }
8
9           Policy: p1022
10          {
11          Description: ``HPol Policy";
12          Status: Enabled;
13          Path: [HPolStart, l2tpd_t, write, l2tpd_var_run_t, HPolEnd
                ];
14          }
15
16          Policy: p1023
17          {
18          Description: ``HPol Policy";
19          Status: Enabled;
20          Path: [HPolStart, l2tpd_t, create, l2tpd_var_run_t,
                HPolEnd];
21          }
```

are accounted for. In the third conditional, the converse of the same process takes place. The only difference is that, in this case, the source of the allow rule is an attribute, and the target is a single type. Notably, there is no path search between "from" attributes and "to" attributes. A process transition this general should not be permitted unless it is explicitly specified. This makes sense since process transitions aim to confine unprivileged access to privileged processes. Also, if it were the case that both the source and destination of a domain transition were attributes, the first "if" clause would cover that case.

An extraction of HERMES output from parsing the targeted policy is presented in Listing 6.5.

6.5 Case Study 2: Visualization

This thesis work did not focus on the problems associated with graph visualizations of SELinux HPol models. However, this section includes some sample graphs from parsing selections of allow rules from the targeted policy. This demonstrates an idea of the size and scope of a full SELinux policy and associated HPol model. Figure 6.3 shows a zoomed-in screenshot of the `dir` object class and its associated permissions once populated into a Gephi viewer graph. Figure 6.4 shows the initial view of a graph populated with all its nodes. Since this is impractical to view, Figure 6.5 shows the graph with the ForceAtlas2 algorithmic layout and 10,000 allow rules.

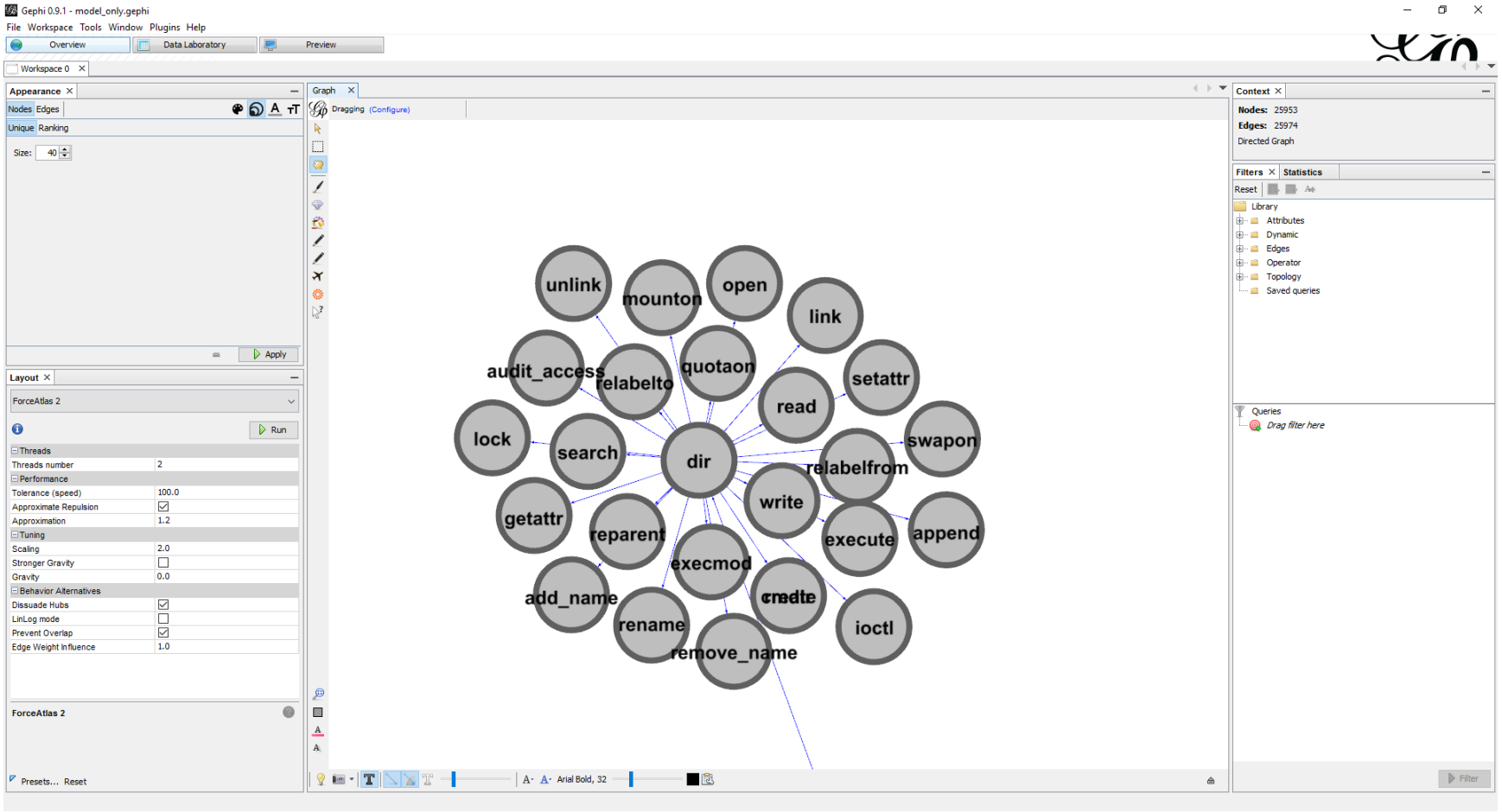

Figure 6.3: SELinux Targeted Policy: Gephi View of Directory Permissions

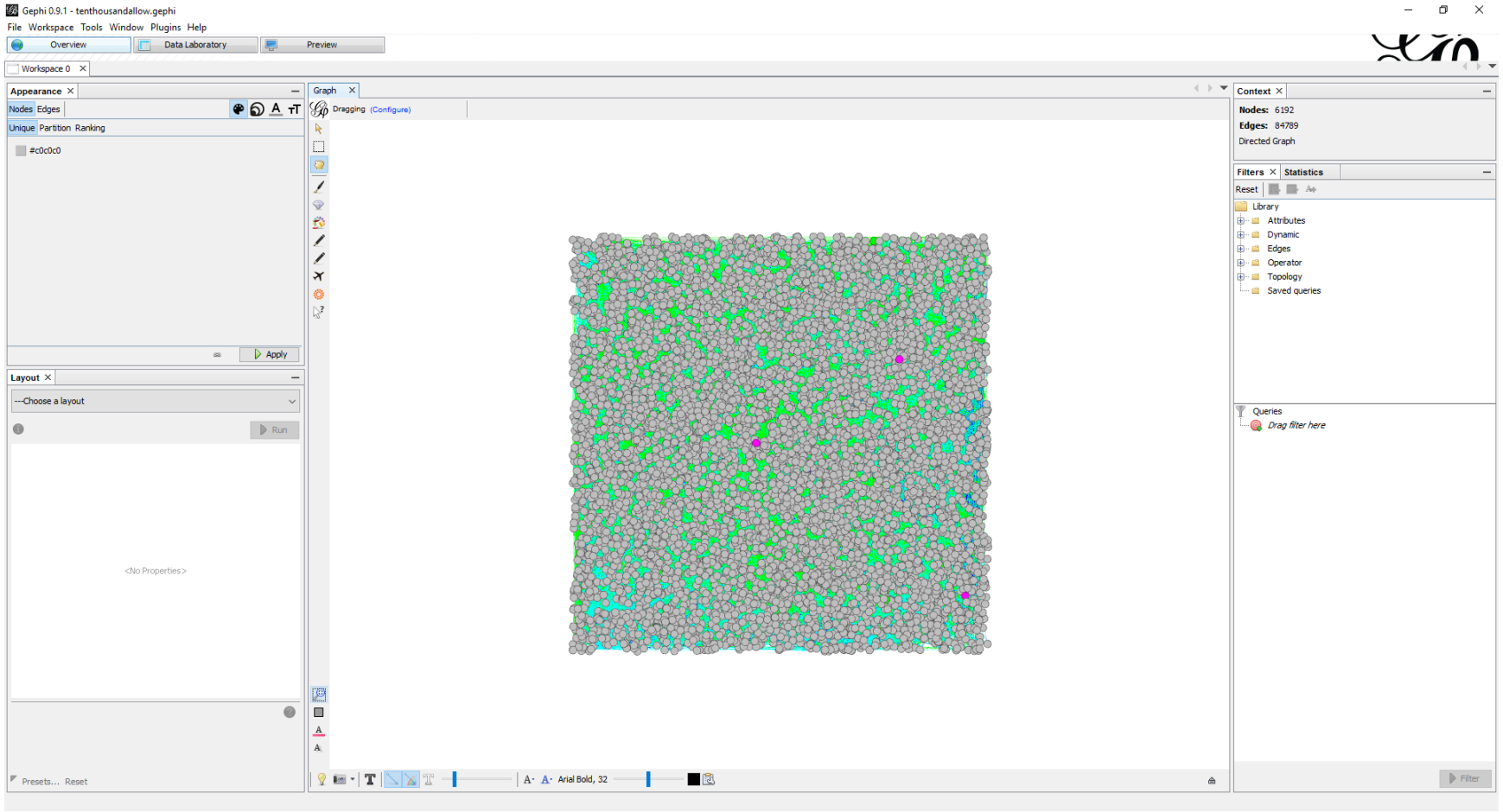

Figure 6.4: SELinux Targeted Policy: Initial View in Gephi

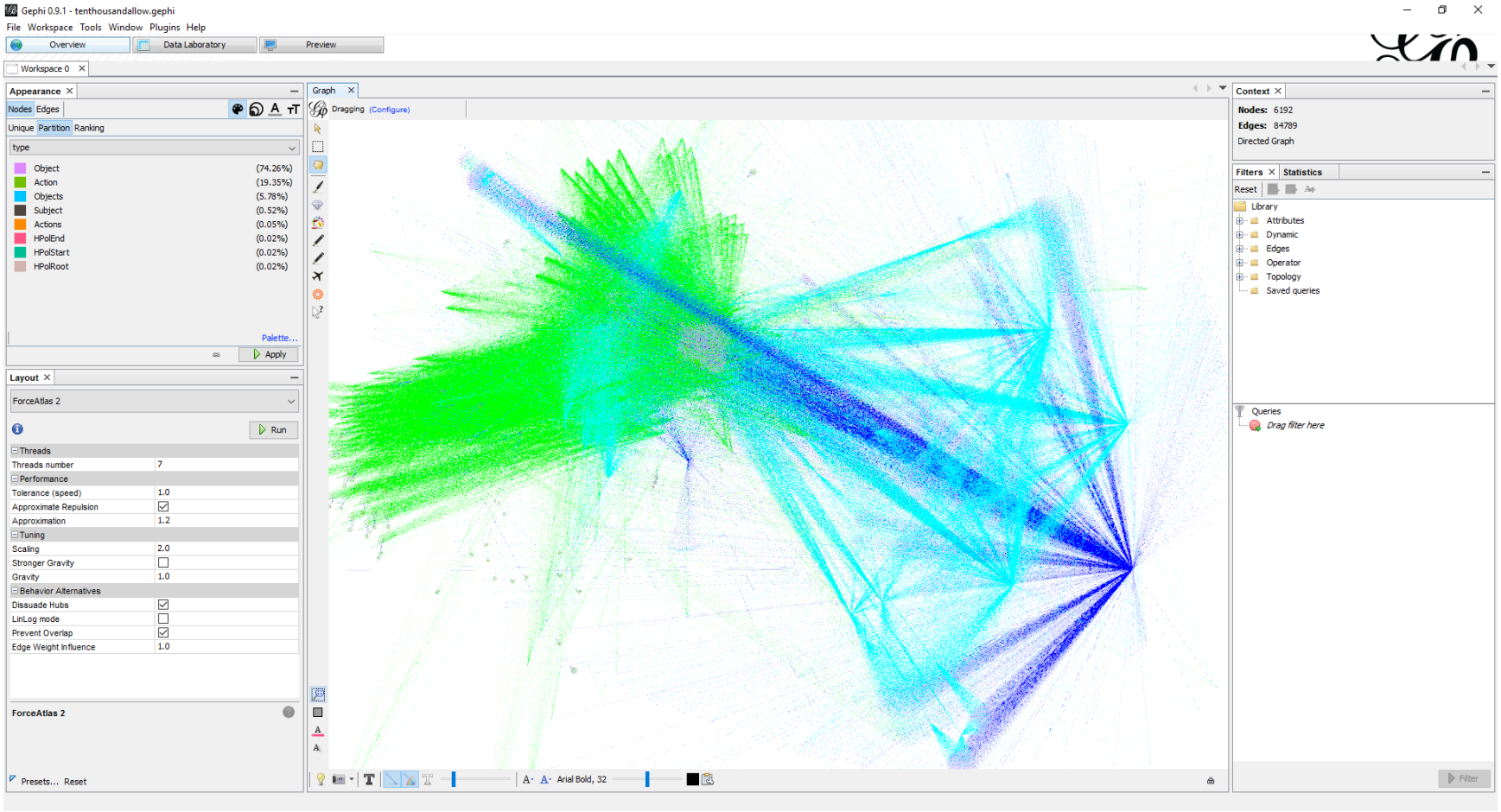

Figure 6.5: SELinux Targeted Policy: ForceAtlas2 Layout View in Gephi for 10,000 Allow Rules

CHAPTER 7

Contribution 2: Interactively Answering Queries About SELinux Device Policies

HERMES output may be queried using the form demonstrated in Chapter 3. However, this requires domain-specific knowledge of HERMES and XSB Prolog and the consequent nine-step process detailed in Section 3.3. Since HPol aims to remove an analyst from domain-specific knowledge as much as possible, the need to learn these paradigms is counter-productive. Ideally, an analyst would need only to know the policy's subject, action, and object in order to attempt to query the HPol model output. This would turn the nine-step process into a two-step process: (1) build the model and run the `convert2hermes()` method and (2) run `hermes_query.py` using the `hermes.out` file generated in step (1).

To simplify the querying workflow, the second contribution of this thesis is a HERMES/XSB Prolog front-end that 1) interactively elicits query information from the analyst, 2) constructs the query in proper HERMES format, and 3) executes the query. Additionally, if the query script is provided with a mapping between policy link numbers and device names, it outputs which device and policy link triggered the query result if the query is answered in the affirmative. This script eases the process involved in verifying policy enforcement.

The operative portion of HERMES output from the `passwd` domain transition documented in Chapter 6 is provided in Listing 7.1. By running the HPol interactive querying script using the HERMES output file and policy path mappings from the SELinux-to-HPol parser, the query output presented in Listing 7.2 is generated.

Policies `p1003` and `p1004` were the operative policies in this example because they include the entire interaction that was intended to be achieved by the domain transition to begin with. Since the user is "alice" and *passwd* is associated with `passwd_t`, the model should answer in the affirmative if queried about whether "alice" may execute the *passwd* program. This result ultimately is what shows the analyst that the domain transition was a success.

Listing 7.1: Operative HERMES Policy Output for Querying the *passwd* Domain Transition

```
1
2           Policy: p1003
3           {
4           Description: ``HPol Policy";
5           Status: Enabled;
6           Path: [HPolStart, alice, unconfined_t, getattr,
               passwd_exec_t, passwd_t, HPolEnd];
7           }
8
9           Policy: p1004
10          {
11          Description: ``HPol Policy";
12          Status: Enabled;
13          Path: [HPolStart, alice, unconfined_t, execute,
               passwd_exec_t, passwd_t, HPolEnd];
14          }
```

An analyst seeking to find out whether Alice can execute *passwd* on a particular HPol model can just enter "alice execute passwd" and, if successful, the script will output a success message. In this case, the script was provided with the policy and device mapping generated by the SELinux-to-HPol parser, so the output displays them as well. Chapter 8 provides a more expanded querying example using this same script.

This parser may also be used to verify the policies within the HERMES output for the targeted policy generated by the parser in Chapter 6. Thus, the following sample policy extracted from the targeted policy HERMES output shown in Listing 7.3 may also be queried successfully with the interactive script (see Listing 7.4).

These query results demonstrate that HPol HERMES output can be successfully queried. This interactive script saves an analyst from having to follow a drawn-out set of steps, making it easier to verify policy implementation. Additionally, the subject-action-object form that the script takes as input lines up with the basic characteristics of any access control policy. At present, this script has been successfully testing on small sample policies only. This is because the XSB Prolog parser does not yet scale to large HERMES files in a timely manner.

Listing 7.2: Output from Querying HERMES Policy Output from the *passwd* Domain Transition

```
subject action object> alice execute passwd

[XSB Prolog Output Removed]

Query successful on policy: p1003

subject action object> alice getattr passwd

[XSB Prolog Output Removed]

Query successful on policy: p1004
```

Listing 7.3: A Sample Policy from Parsing the SELinux Targeted Policy

```
1
2       Policy: p1032
3       {
4       Description: ``HPol Policy";
5       Status: Enabled;
6       Path: [HPolStart, staff_t, send_msg, blueman_t, HPolEnd];
7       }
```

Listing 7.4: Query Result for the Sample Policy from the SELinux Targeted Policy

```
subject action object> staff send_msg blueman

[XSB Prolog Output Removed]

Query successful on policy: p1032
```

CHAPTER 8

Contribution 3: Policy Merging for Multiple Device Policy Violation Detection

In order for the HPol model to abstract policies from multiple devices, it must include formal operations to do so. For this contribution, abbreviated SELinux policies were successfully merged and queried. The policies retain all initial attributes from their state prior to merging, thus there was no information lost along the way. Policy merging has only been attempted with the SELinux HPol exemplar, but its functionality is now part of the library and should readily apply to other component-to-HPol parsers.

As in the sample SELinux domain transition policy detailed in Chapter 6, policy merging work completed for this contribution used abbreviated SELinux policy models. In these models, each individual policy represents a selection of nodes and policies from a stand-alone device (i.e. a single workstation with an active SELinux policy). Merged policies and models consist of two or more abbreviated device models. The next step in this area will be to address the merging of full SELinux policies.

To begin the process of merging multiple SELinux-to-HPol models, individual models were populated one-by-one per the same process used in Chapter 6. A merged policy must have distinct labeled policies (e.g. policy `1004` cannot be used in both Device 1 and Device 2). Distinct HPol models have an internal policy dictionary that represents policies that have already been populated. Thus, to ensure contiguouos numbering, each new model's policy dictionary was updated to include all prior policy dictionary entries before new policies were added. The `setPolicyDict()` method was added to the HPol library to allow the individual parser to accomplish this task since updating policy dictionaries is a method internal to the library itself.

Once each individual HPol model was populated, merging could take place using the `mergePolicies()` method added to the HPol library as part of this contribution. To use

```
for i in range(0, len(hpols)):

        hpol = hpols[i]

        # graph composition operation merging graphs of models
        self.__nxMultiDiGraph = nx.compose(self.__nxMultiDiGraph,
        hpol.__nxMultiDiGraph)

        # add new root node for merged policies
        self.__nxMultiDiGraph.add_edge(self.__rootNode, hpol.
            __rootNode)

        # eliminate prior start and end links
        # (merged model has 'master' start / end link for all
            policies)
        self.__nxMultiDiGraph.remove_node(hpol.__ppStartNode)
        self.__nxMultiDiGraph.remove_node(hpol.__ppEndNode)
```

Figure 8.1: Policy Merging Code from the mergePolicies() Method of the HPol Library

`mergePolicies()`, a new HPol model representing a merged policy was created. Then, its policy dictionary was updated to reflect all policies that were populated in the pre-merged models. Merging took place using the `mergePolicies()` method following this step.

Figure 8.1 demonstrates the portion of the `mergePolicies()` code where the initial steps of the HPol model merging takes place. This code iterates through all the individual models populated by the SELinux-to-HPol parser and passed to the method with the `hpols` object. With each iteration, individual models are merged into the new merged HPol model using a graph composition `compose()` method for multi-digraphs from the networkx library [20].

```
pol_path_mappings = []
# populate starting/ending nodes for all named policies
for i in range(0, len(endPaths)):
        for polID, path in endPaths[i].iteritems():
                pol_path_mappings.append(str(polID)+': '+str(path
                   ).split('/')[0])
                pol_name = path.split('/')[0]
                self.addStartLinkToPolicyPath(ppID=polID, toNode=
                   self.hpolName+'/'+pol_name+usr_path)
                self.addEndLinkToPolicyPath(ppID=polID, fromNode=
                   self.hpolName+'/'+path)

return pol_path_mappings
```

Figure 8.2: Code to Connect the Merged Start and End Node to the Named Policies from the mergePolicies() Method of the HPol Library

After composition, the individual model is incoporated with the merged model with all its links, but it is not connected to the root node of the merged model. The networkx `add_edge()` method connects the recently-populated device model with the merged model. Then, the networkx `remove_node()` method is used to remove the device model's start and end policy nodes. These are removed because a single HPol policy contains only a single start and end node. Accordingly, in the merged model, all policies originate and terminate in the start and end nodes belonging to the merged model.

When policy start and end nodes were removed from the individual device models during merging, policies belonging to the individual models were left without being anchored to the master merged policy start and end nodes. Figure 8.2 provides the code used to populate the links between the merge model's start and end nodes, and all the policies that populate

the merged model. This code iterates through all named policies and associated paths stored in `endPaths`. First, the association between the named policy's ID and the initial device it belonged to are stored as a string in `pol_path_mappings` which is later returned by `mergePolicies()`. This associated can be used by an administrator to determine what device a policy query was successful on when used by a script such as the one demonstrated in Chapter 7. Then, the master start and end links from the merged model are connected to the individual policies, completing the merging process.

To demonstrate a practical example of SELinux policy merging, assume a scenario in which an organization wishes to implement a simple Bell-LaPadula model. For this purpose they have assigned a `high` privilege level to one machine and a `low` privilege level to the other. The user Alice may log in to either machine, but only given the privilege associated with that machine. The `high` machine has a project diretory associated with the `high_t` type in SELinux. Similarly, the `low` machine has a `low_t` directory with a `low` privilege.

In accordance with Bell-LaPadula [11], when Alice uses the `high` machine she may read or write to `high_t`. Also she may read from `low_t`, but cannot write to it. Likewise, on the `low` privilege machine, Alice may read or write to `low_t`. Also, she may write to `high_t`, but cannot read from it.

In this scenario, an analyst, Bob, wishes to find out whether Bell-LaPadula is properly in place when this system is considered in its entirety. He knows that if Alice can read on `high_t` and write on `low_t`, then the Bell-LaPadula model is not enforced correctly. His analysis will only consider the device policies at face value. The result of his queries will be independent of other means to enforce the model, such as a ban on thumb drives within the organization.

First, Bob must model and query the `high` and `low` machines separately to ensure that Bell-LaPadula is in effect on either one. Figures 8.3 and 8.4 shows the HPol model for the `low` and `high` machines constructed separately. The query output found in Listings 8.1 and 8.2 shows that Bell-LaPadula is properly enforced on the separate machines: Alice cannot

Listing 8.1: Query Results for the Low Privilege HPol Model

```
subject action object> alice write low

[XSB Prolog Output Removed]

Query successful on policy: p1003

subject action object> alice read high

[XSB Prolog Output Removed]

Query failed.
```

Listing 8.2: Query Results for the High Privilege HPol Model

```
subject action object> alice write high

[XSB Prolog Output Removed]

Query successful on policy: p1002

subject action object> alice write low

[XSB Prolog Output Removed]

Query failed.
```

read `high_t` on the `low` machine and she cannot write to `low_t` on the `high` machine.

To complete his inquiry, Bob must merge the `high` and `low` machine policies into one and ensure that Bell-LaPadula holds. Listing 8.3 demonstrates that, when both systems are taken into account, Bell-LaPadula does not hold: Alice can read high and then write low.

Granted, the example policy in this section is simple to a fault. Taking that into consideration, it serves to demonstrate the ability to properly merge two separate HPol models and to query them successfully with no information loss during the process. Further, large enterprises are comprised of massive networks of workstations and even simple queries can

Listing 8.3: Query Results for the Merged HPol Models

```
subject action object> alice read high

[XSB Prolog Output Removed]

Query successful on policy: p1004

subject action object> alice write low

[XSB Prolog Output Removed]

Query successful on policy: p1003
```

be illustrative of proper policy enforcement or its absence. Along these lines, Figure 8.6 demonstrates the successful merging of three separate policies. This method may merge n separate policies, given enough memory and time.

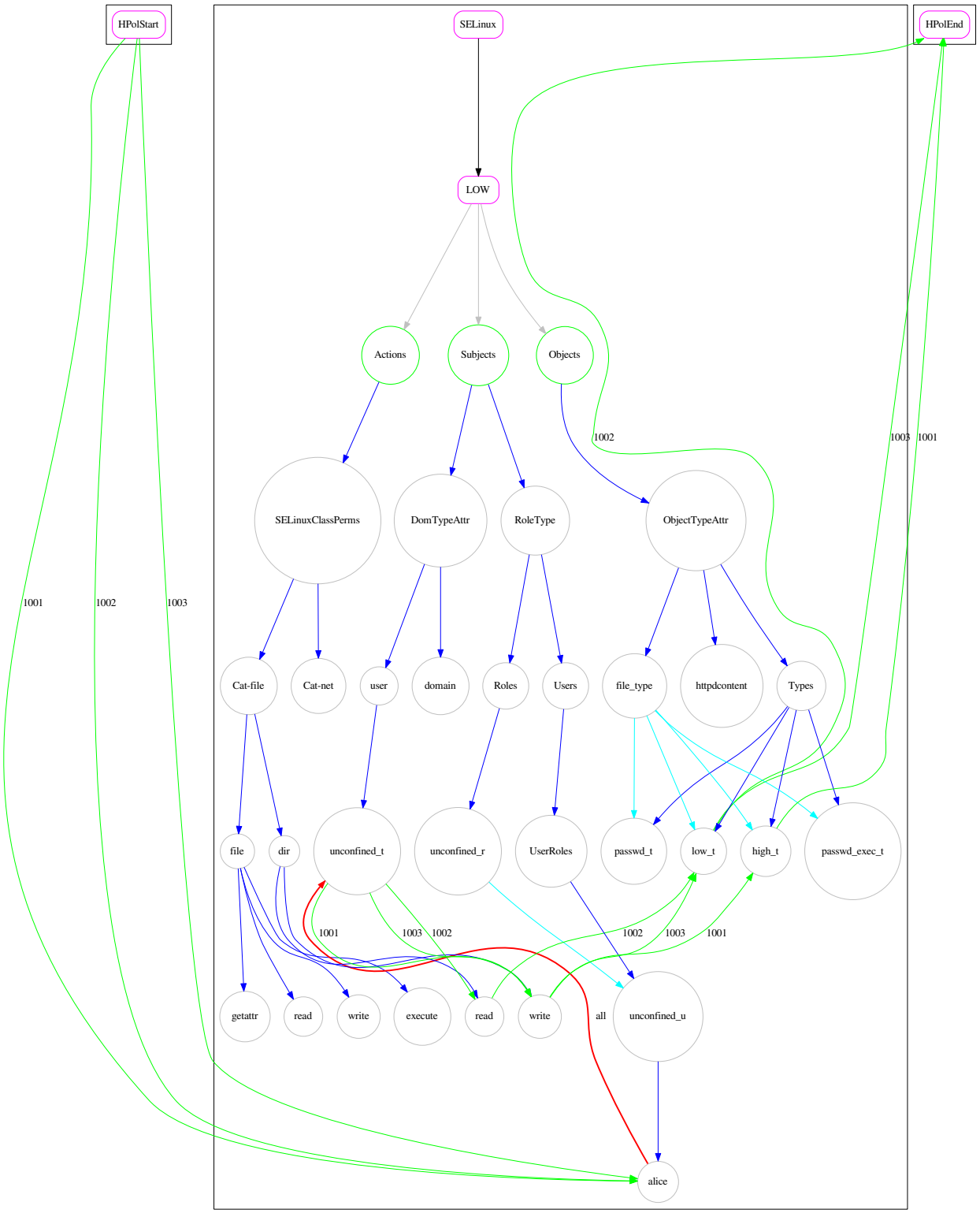

Figure 8.3: Abbreviated HPol Model Representing Interactions on the Low Privilege Machine

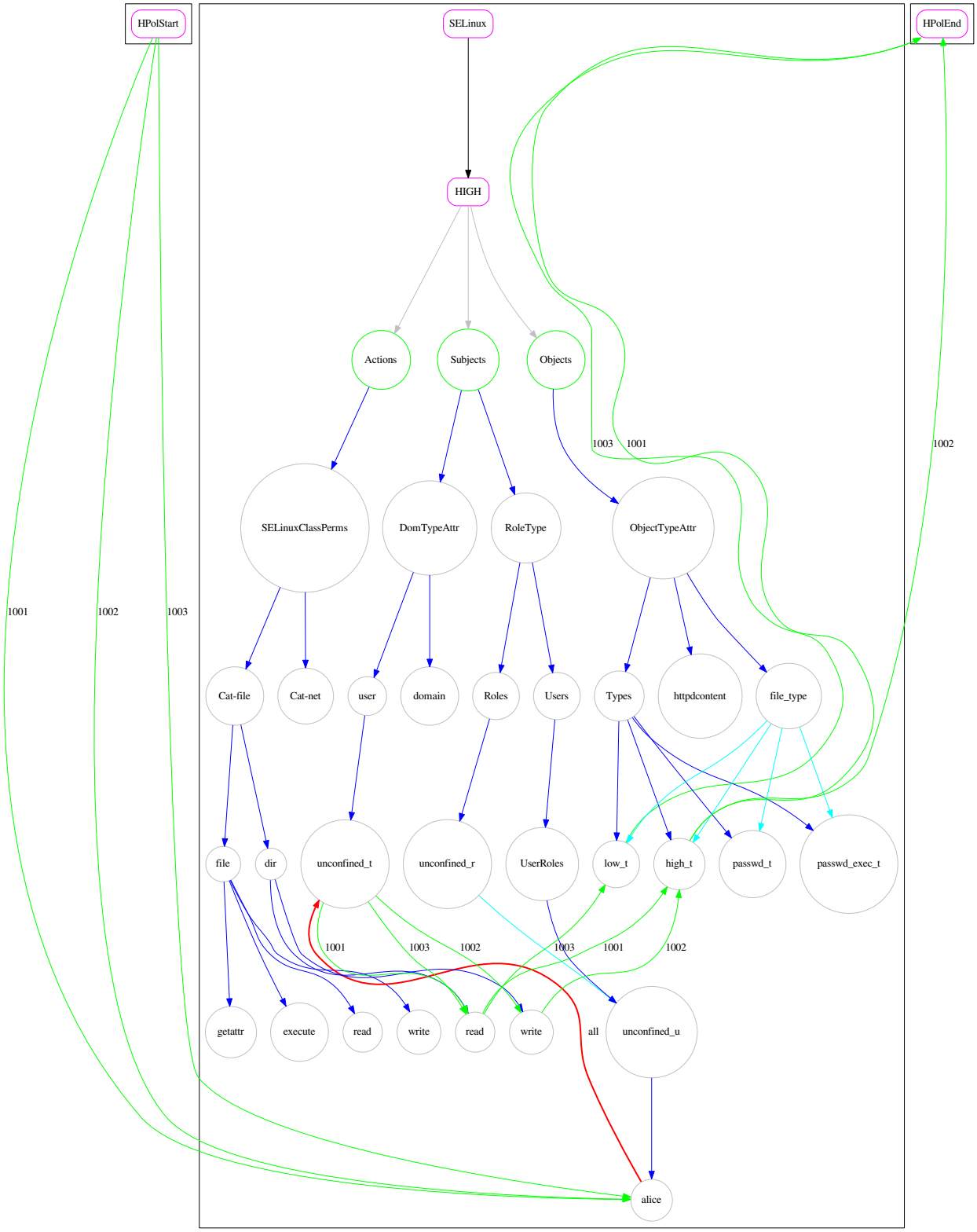

Figure 8.4: Abbreviated HPol Model Representing Interactions on the High Privilege Machine

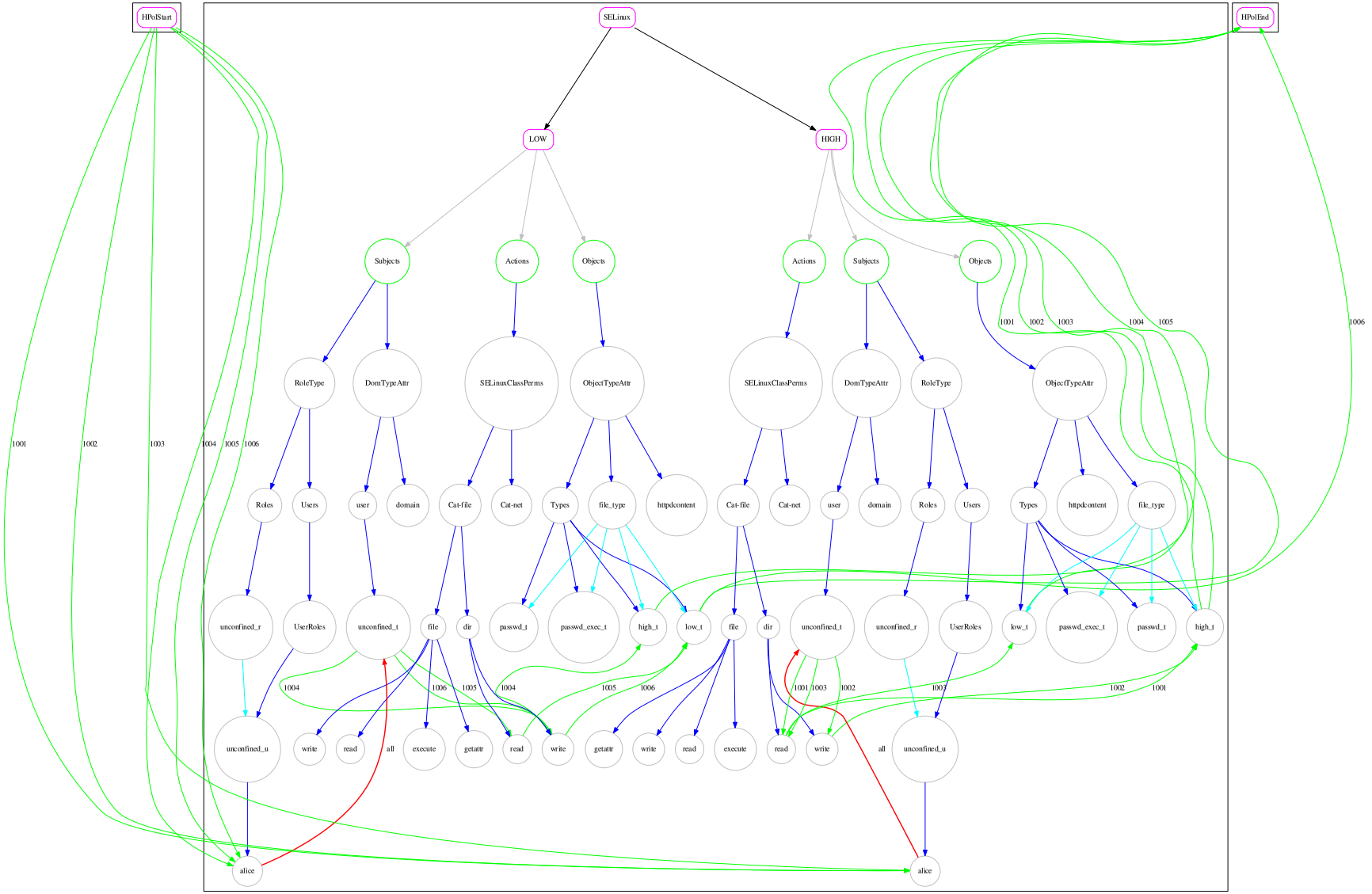

Figure 8.5: SELinux Model Resulting from Merging High and Low Privilege Machine Policies

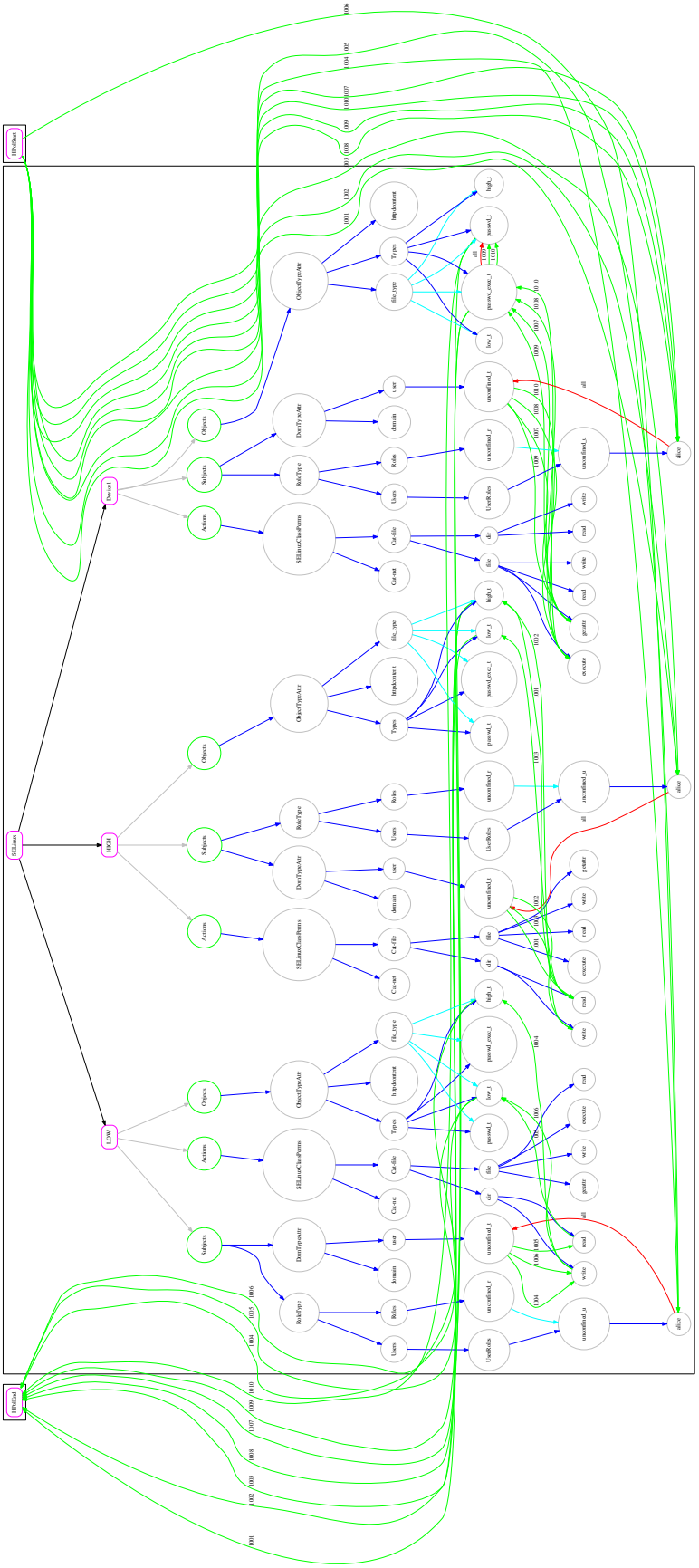

Figure 8.6: SELinux Policy Merging: Merging Three Policies

CHAPTER 9

Future Work

This chapter presents some ideas for future work that may be carried out in any of the directions presented by the contributions of the thesis.

9.1 Visualization

As it relates to SELinux-to-HPol parsing, the two biggest areas HPol efforts are intended to focus on are policy querying and policy visualization. The current parser focuses mostly on the querying problem. Further, while the model can be visualized with the current parser, graphs quickly become cumbersome when parsing large-scale policies that will necessarily end up including thousands of nodes and edges. There needs to be a sleeker, smarter way to display HPol models with many nodes. This will probably involve the usage of a different viewer or a higher-level abstraction for node and interaction display. The current parser already accounts for SELinux's attributes which are collections of types. Other HPol-specific solutions may also be possible.

9.2 Selective Parsing

A slightly tangential issue to visualization is that of the management of many nodes. While it is important to populate an entire SELinux policy in HPol, most actual uses of it will probably only require a subset of that information to achieve querying and modeling for specific problem domains. For example, an administrator interested in mapping out policies for network daemons will not have a need for information from the file types and associated permissions associated with the operating system. Thus, the model could include more functionality for dynamically culling irrelevant nodes within the hierarchical DAGs and policies that do not apply to the task at hand. To this end, a selective parser could be built to only

parse the relevant node and policy data.

9.3 Incorporation of Linux Filesystem Policies

Another issue to tackle with the SELinux-to-HPol parser is that of mapping out disparities between traditional Linux discretionary access control and SELinux policies. Since an action may be permitted in SELinux, but not on the operating system's file system (and vice versa), an incomplete view of the subject's capabilities on the given device is given under the current approach. An even more detailed model would include a means for incorporating all Linux file system ownership data, as well as all SELinux policy information. Then, there would be more assurance that policies stretching across the whole operating system were parsed and accounted for.

9.4 Creating SELinux Configurations Using a High-level Policy Decription Language

The ability to use HPol to create actual SELinux configurations a la Lobster [1] would help ensure that the intended policies were put into effect. Using a high-level policy description language that is easily-discernible to policy designers and free of component-specific knowledge would lessen the need to spend resources verifying that the correct policies were drafted at the component configuration level. This task would be the realization of an ultimate goal of HPol which is to be able to not only model policies for disparate devices, but to carry about policy implementation across the devices from a higher-level.

9.5 Expansion of the Interactive HERMES Querying Script

The interactive querying script is limited in the way that it can evaluate queries in that it can only handle a single query at a time. This limitation is largely dictated by the current iteration of HERMES and the back-end XSB Prolog modules that the script relies on. An

ideal querying script would also allow for compound queries. Operations included within the queries could also be useful; even a basic "and" operation would drastically cut down on the time it takes an analyst to process a query. For example, instead of writing separate queries to check whether alice can read and write on a file (e.g. **alice write file**), an and operation could take the form `alice read and write file`.

9.6 Merging Policies for Additional Types of Devices

A major goal of HPol is to allow for high-level operations between high-level formal policy models. The first steps to achieve this goal have been taken with the merging of multiple HPol policy models in this thesis. Though this functionality has been implemented within the HPol library itself, testing and application needs to be carried out across other HPol exemplars and on full SELinux policies. In order for this to scale well, it needs to be tested on an enterprise-grade machine and optimized accordingly.

CHAPTER 10

Summary and Conclusion

This final chapter summarizes the problem that this thesis addressed and the contributions made to address it.

10.1 Summary

In order to adequately secure their domain from unauthorized accesses and uses, large organizations are tasked with developing a security policy in which they determine who can access which resources in what manner. To verify that specific policies are implemented properly, technicians typically need to test them on a device-by-device basis, requiring significant resource consumption and domain-specific technical knowledge for staff members. These requirements limit the timeliness with which policies can be verified. They also provide only a partial view of overall policy coverage. These constraints can serve to inhibit administrators from making effective policy decisions in an informed and timely manner.

The goals of the HPol project are to provide higher assurance of policy coverage and to help eliminate the gap in understanding security policies as they are specified versus how they are actually implemented in disparate system components. This thesis demonstrated an application of this high-level, hierarchical security policy model for an exemplar mandatory access control implementation for enterprise Linux workstations equipped with the SELinux mandatory access control mechanism. With the help of an SELinux-to-HPol parser, and its related scripts, SELinux models were constructed, both for finer-grained policies (a user executing *passwd*) and for an operating system's combined policies (the default SELinux "targeted" policy). Models constructed by the parser can be visualized using several different graphing programs. In addition, their output can be queried in an intuitive and interactive way. By evaluating the responses from HPol queries, administrators verify which security polices are in effect in a timely manner with minimal domain-specific knowledge.

Beyond SELinux, this thesis demonstrated an addition to the HPol library which allows for merging of multiple HPol policy models. When an administrator can construct a merged HPol model, he or she can treat it as a holistic model of several system components incorporated into one. This saves him or her from having the spend time and resources learning domain-specific knowledge for deciphering the ramifications of policies from individual components. Instead, he or she can focus on whether policies are implemented as they were intended to be from a higher-level.

10.2 Conclusion

The work described in this thesis demonstrates that a high-level policy model be constructed from low-level SELinux policies. The scripts used to carry out this task do not represent a complete translation of SELinux configurations to the HPol format. For example, SELinux features like the ability to constrain allow rules or its ability to implement multilevel security were not included. In addition, querying and visualization in the current approach fall short in cases where scalability and resource limits prevent optimal solutions.

Instead, the work of this thesis demonstrated that a useful HPol model of SELinux mandatory access configurations is possible to construct. It accomplished this by showing different example tasks that would improve the ease and efficiency of understanding and verifying a model by administrators. It showed that typical SELinux policies can be successfully converted into an HPol model that can be verified in an intuitive way by querying model output interactively. Finally, separate component policies were merged into one, with no information loss, creating a higher layer of abstraction for policy verification.

Bibliography

[1] J. Hurd, M. Carlsson, B. Letner, and P. White, "Lobster: A domain specific language for selinux policies," Galois internal report, Tech. Rep., 2008.

[2] Z.A. Soomro, M.H. Shah, and J. Ahmed, "Information security management needs more holistic approach: A literature review," *International Journal of Information Management*, vol. 36, no. 2, pp. 215–225, 2016.

[3] K. Fisler, S. Krishnamurthi, L.A Meyerovich, and M.C. Tschantz, "Verification and change-impact analysis of access-control policies," in *Proceedings of the 27th international conference on Software engineering*. ACM, 2005, pp. 196–205.

[4] V.C. Hu, D.R. Kuhn, T. Xie, and J. Hwang, "Model checking for verification of mandatory access control models and properties," *International Journal of Software Engineering and Knowledge Engineering*, vol. 21, no. 01, pp. 103–127, 2011.

[5] J.D. Guttman and A.L. Herzog, "Rigorous automated network security management," *International Journal of Information Security*, vol. 4, no. 1-2, pp. 29–48, 2005.

[6] R. Shirey, "RFC 4949, internet security glossary, version 2," 2007. [Online]. Available: https://tools.ietf.org/html/rfc4949

[7] W. Stallings and L. Brown, *Computer Security Principles and Practice*, 3rd ed. Pearson Education, Inc., 2015.

[8] F. Mayer, D. Caplan, and K. MacMillan, *SELinux by example: using security enhanced Linux*. Pearson Education, Inc., 2007.

[9] Red Hat, Inc., "Selinux users and administrators guide," 2016. [Online]. Available: https://access.redhat.com/documentation/en-US/Red_Hat_Enterprise_Linux/7/html/SELinux_Users_and_Administrators_Guide/

[10] ——, "Targeted policy overview," 2006. [Online]. Available: https://www.centos.org/docs/5/html/Deployment_Guide-en-US/sec-sel-policy-targeted-oview.html

[11] E. Bell and J. LaPadula, "Secure computer systems: Mathematical foundations," MITRE, Tech. Rep., 1973.

[12] T. Lee, "Selinux howto," Accessed October 2016. [Online]. Available: https://wiki.centos.org/HowTos/SELinux

[13] Ananth A. Jillepalli and Daniel Conte de Leon and Stuart Steiner and Frederick Sheldon, "HERMES: A High-Level Policy Language for High-Granularity Enterprise-wide Secure Browser Configuration Management," in *Proc. 2016 IEEE Symposium Series on Computational Intelligence (SSCI-2016)*. Athens, Greece: IEEE, 06-09 December 2016.

[14] A. Jillepalli, D. Conte de Leon, "An Architecture for a Policy-Oriented Web Browser Management System: HiFiPol: Browser," in *Computer Software and Applications Conference (COMPSAC), 2016 IEEE 40th Annual*, vol. 2. IEEE, 2016, pp. 382–387.

[15] D. Conte de Leon, "Hpol: Hierarchical and formal system security policies," University of Idaho, Center for Secure and Dependable Systems, Final Research Project Report, 2015, unpublished.

[16] M. Brown, "Hierarchical formal modeling and verification of router policies with an applied case study to cisco router configurations," Master's thesis, Center for Secure and Dependable Systems, University of Idaho, Moscow, Idaho, December 2016.

[17] (2016) Pygraphviz. [Online]. Available: https://pygraphviz.github.io/

[18] J.D. Guttman, A.L. Herzog, J.D Ramsdell, and C.W. Skorupka, "Verifying information flow goals in Security-Enhanced Linux," *Journal of Computer Security*, vol. 13, no. 1, pp. 115–134, 2005.

[19] T. Nelson, C. Barratt, D.J. Dougherty, K. Fisler, and S. Krishnamurthi, "The Margrave Tool for Firewall Analysis." in *USENIX Large Installation System Administration Conference*, San Jose, CA, 07-12 November 2010.

[20] (2016) Networkx. [Online]. Available: https://networkx.github.io/