Hierarchical Security Policy Modeling and Composition for Role-Based Access Control Models with an Application to the OpenStack Cloud Platform

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

Antonius Quinn Stalick

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

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

Department Administrator: Frederick Sheldon, Ph.D.

May 2017

Authorization to Submit Thesis

This thesis of Antonius Quinn Stalick, submitted for the degree of Master of Science with a Major in Computer Science and titled "Hierarchical Security Policy Modeling and Composition for Role-Based Access Control Models with an Application to the OpenStack Cloud Platform," has been reviewed in final form. Permission, as indicated by the signatures and dates below is now granted to submit final copies for the College of Graduate Studies for approval.

Major Professor:	Daniel Conte de Leon, Ph.D.	Date
Committee Members:	Michael Haney, Ph.D.	Date
	Axel Krings, Ph.D.	Date
Department Administrator:	Frederick Sheldon, Ph.D.	Date

Abstract

Society relies on the connections between information systems and, as need for these interconnected systems has grown, so has the system complexity. Modern system security can prove difficult; fully understanding the interactions of the system's components is both technically intensive and time consuming. System security policy managers require a method by which they may clearly operate on system access security policies. This thesis describes a formal model for policy abstraction in a role-based access controlled environment, combination of multiple policies, and abstracted reconstruction of policy artifacts in this environment. This work contributes: formal role-based access control policy modeling, formal policy composition, and policy file reconstruction. This thesis demonstrates the applicability of the Hierarchical Security Policy model to role-based access control systems, capability to compose security policies from disparate sources, and capability to maintain enough information to permit modification and redeployment of security policy artifacts.

Acknowledgements

I extend my gratitude to my major professor, Dr. Daniel Conte de Leon whose continued guidance made my research and this thesis possible.

I would like to thank my committee members, Dr. Axel Krings and Dr. Michael Haney whose input and perspective made valuable refinements to this work.

Additionally, I would like to thank the other researchers who have worked on the Hierarchical Security Policy Formal Model project: Jared Zook, Matthew Brown, and Ben Cumber.

Work on this research and thesis was made possible by the National Science Foundation CyberCorp® Scholarship for Service program administered by the U.S. Office of Personnel Management and the University of Idaho Center for Secure and Dependable Systems.

Table of Contents

List of Figures

Chapter 1: Introduction

Enterprise IT staff, network administrators, and cyber-security personnel require a method by which they may clearly operate on system access security policies in a time-efficient way. Modification and management of a single policy in isolation is simple, however, modern security policies in both small business and enterprise are much larger than a single policy. To properly address the access provided to the entire system, enterprise IT staff, network administrators, and cyber-security personnel must account for multiple devices and subsystem access policies as well as the connections between them.

Formal policy modeling allows these system security policy managers to better secure their system by providing an abstraction which can incorporate multiple subsystem security policies, account for the interactions between disparate subsystem policies, and provide an interface to the manager for simple system policy management. Formal policy modeling provides system security policy managers the tools required to address the size and scope of modern security policies in a clear, time-efficient way.

This thesis demonstrates the applicability of the Hierarchical Security Policy Formal Model to Role-Based Access Control systems, displays the ability of the Hierarchical Security Policy Formal Model to combine disparate system subpolicies into full policies, and shows how maintenance of subsystem security policy semantics allows the hierarchical security policy model process to be reversed, returning the original policy artifacts to a system-usable state. Additionally, this thesis includes a case study of the OpenStack cloud platform wherein these methods are instantiated.

1.1 Problem

The core goals of Information Assurance are to ensure the confidentiality, integrity and availability of data held by a system; the security of a system can be gauged by the its ability to ensure these attributes. Misconfiguration of system security policy can lead to insecurities in each of these attributes: users may be incorrectly given rights to view confidential files, file integrity may be threatened by similar users with modification rights, and availability issues may arise if a user is unable to access files required for their work. Additionally, modern enterprise systems contain a mixture of interdependent devices, software systems and enterprise-level policies; modifying the security policies of these complex systems can lead to misconfiguration as their effects can be spread across the entire system, and not immediately apparent to the system security policy manager. System security policy managers need an method by which they can audit and manage system security policy across their entire system in a simple and efficient way.

1.2 Approach

Formal policy modeling allows system security policy managers to better secure their system by providing an abstraction which can incorporate multiple subsystem security policies, account for the interactions between disparate subsystem policies, and provide an interface to the manager for simple system policy management. The Hierarchical Security Policy Formal Model project seeks to provide this interface through a layered, hierarchical approach. This approach relies on an annotated graph abstraction of system security policy artifact data to maintain the semantics of the original system security policy. This approach is accomplished in three steps: system security policy artifact identification and organization, system security policy evaluation, and system security policy verification. Additionally, steps may be taken in the reverse order, allowing reduction of the abstraction of the hierarchical model, resulting in the original policy artifacts and the ability to redeploy potentially modified system security policies to their original systems.

1.2.1 System Security Policy Artifact Identification and Organization

This process begins by first analyzing static system security policy artifacts, such as database dumps and configuration files, dividing the policy into three groups: Subjects, Actions and Objects. Notably, as this process is performed on static system security policy artifacts, the hierarchical policy model for a system may be processed and created outside of a live operating environment. This allows the policy model to be constructed using information from the live operating environment, without threatening uptime or system reliability from the computation load potentially required by the hierarchical policy model process. The three groups are then subdivided into a representation of the semantics of the specific system policy being modeled and organized in a hierarchical fashion through directed acyclic graphs. The nodes of these graphs may include annotations which contain additional information about system security policy artifacts which may not be immediately semantically significant to the system security policy, but may become significant later in this process. This additionally allows the system security policy manager to identify system security policy artifacts from multiple locations in their system, collect and centralize each individual policy artifact, and use the hierarchical security policy model process to collate all of the system security policy artifacts simultaneously.

For example, if the system security policy incorporates grouping of users to provide access to system functions, the Subjects of the hierarchical policy model for that system may be organized such that the nodes of the graph representing the groups of users in the system security policy are the parent nodes of one or many nodes representing the specific users of the system who are members of the parent group.

In this case, the parent nodes, system user groups, will likely have multiple children as most system user groups contain more than one user. Nodes representing the users in this case are also likely to have more than one parent, as system users are often members of more than one group. Additionally, consider a system security policy which governs access to a file structure comprised of files and directories. The Objects of the hierarchical policy model representing this file structure will be organized such that the nodes representing a file directory are the parents of the nodes representing the files or file directories immediately inside them. If this method is applied to the an entire file system, the resulting directed acyclic graph for the Objects of the system will represent the hierarchical nature of the file structure used by the system.

1.2.2 System Security Policy Evaluation

The Hierarchical Security Policy Formal Model process then proceeds to evaluation of the system security policy. The second step receives the previously constructed directed acyclic graph from the first step, and adds two initially unconnected nodes denoting the Start and End of any policy paths created during this step. This step of the process entails enumeration of the accesses Subjects of the system security policy have to Actions on Objects of the system. These individual accesses are then linked together to form a single policy path, which denotes at least a Subject, Action and Object from the directed acyclic graphs created by the first step of this process. This combination of Subject, Action and Object into a sequential policy path should maintain the semantic meaning of the system security policy; if the system security policy evaluates access based on the Subjects of the system being checked for permission to perform an Action on a system Object, the policy path should first traverse the Subject graph, then the Action graph, and finally the Object graph before completion. Valid policy paths created during this step being at the Start node, traverse the graph including at least one of each hierarchical policy model group type, and complete by connecting to the End node. Policy paths created during this step should be unique; there should not be more than one policy path created based on the same system security policy artifact or sequence of artifact data. Multiple policy paths relating the same access may exist however, though these accesses should be each derived from different system security policy artifacts. Allowing multiple equivalent policy paths derived from different system security policy artifacts provides system security policy managers with a view of the access allowed to Subjects of their system and properly attributes each of the individual access mechanisms satisfied by the Subject to their respective originating system security policy artifact. Policy paths may also contain wildcard links. These wildcard links allow any policy path to traverse them, and may be included in multiple individual policy paths.

Wildcard links should not be used to provide a shorthand for the system security policy rather, wildcard links should be used to maintain the semantics of the system security policy. It is important to note that the policy paths created during this step of the hierarchical security policy model process do not modify the underlying directed acyclic graphs from the first step. Additionally, specific individual policy paths are considered independent from others and policy paths may loop back onto themselves, as the order of the nodes in the path is tracked and provides semantic significance.

Continuing the examples from above, assume the Action portion of the hierarchical security policy model directed acyclic graph has been filled with a representation of operations system users and groups may wish to take on the files and directories. Each individual access allowed by every user in the system is then enumerated and designated its own unique policy path. The user groups handled by the system security policy also receive policy paths. The policy paths for user groups are likely to use wildcard links, as any user who is a member of a group has permission to perform Actions on Objects allowed to the group.

1.2.3 System Security Policy Verification

The third step of the Hierarchical Security Policy Formal Model process retains the abstractions of the prior steps but moves the system security policy data into a logical database, where the data can be queried in a human-readable manner. Transitioning the abstraction from the directed acyclic graph overlaid with policy paths to a logical database simplifies the interface for the system security policy managers from a large graph visualization to a command line or a formal query structure in the High-Level Easy-to-Use Reconfigurable Machine Environment Specification language, HERMES [6]. The query database provides an abstraction wherein every policy path developed in the previous step can be tested against simultaneously, allowing the system security policy manager to perform auditing for undesired access. Utilizing the query database, the system security policy manager is able to efficiently and effectively audit system security policies across multiple subsystem policies through a consistent interface.

The logical database supports two main types of query: queries checking existence of a specific access, and queries as a form of filter over the set of hierarchical security policy paths. Both of these query forms return results as a list of policy path identifiers, or an empty list if no policy satisfies the query. The first form of query allows system security policy managers to check if a specific access is allowed or disallowed by the system security policy abstraction created in the previous steps of this process. In the second form of query, system security policy managers can create filters which return lists of policy path identifiers

containing the queried sequential hierarchical security policy model graph nodes.

Again continuing the example of this section, assume all policy paths have been enumerated for the access allowed to every user and user group governed by the system security policy. The valid policy paths are then loaded into a logical database which is then prepared to receive queries from the system security policy manager. The security policy manager could then issue a query to the database filtering for a single file Object, receiving policy path identifiers including various hierarchical security policy model Subjects and the Actions allowed to them on the specified Object. The system security policy manager may then further investigate these specific policy paths for potential enterprise policy violations such as users who have been removed enterprise-level projects, but have not yet been removed from the system security policy user groups granting access to the file Object used by these projects.

1.2.4 Policy Management by Removing Abstraction

The straightforward application of the Hierarchical Security Policy Formal Model process allows system security policy managers to audit access control policy across multiple system security subpolicies and system security policy artifacts in a effective and efficient way. The process is structured in a way which enables the abstraction layers to be removed and the original system security policy artifacts reconstructed. This reverse application of this process allows system security policy managers to actually manage system security policy from the abstracted view provided by the higher levels of the hierarchical security policy model. This application of the process is further discussed in Chapter 7.

1.3 Overview of this Thesis

Chapter 2 describes work related to the research questions and contributions of this thesis. Chapter 3 describes the background of the OpenStack platform, Hierarchical Security Policy Formal Model and works related to this model. Chapter 4 contains a description of the research objectives of this thesis, as well as a brief overview of each of the major contributions of this thesis. Chapter 5 provides application of the Hierarchical Security Policy Formal Model to a Role-Based Access Control system through the OpenStack cloud computation platform. Chapter 6 shows how, in addition to the contribution provided by Chapter 5, subsystem security policies for Role-Based Access Control systems may be combined in the Hierarchical Security Policy Formal Model through composition. Chapter 7 illustrates the reverse of the Hierarchical Security Policy Formal Model process, reducing the abstraction layer by layer and reconstructing the original system security policy artifacts, even through supported modification of the system security policy. Both Chapter 5 and Chapter 6 contain information supporting the contribution

made by Chapter 7, as reconstruction of the system security policy and subpolicies requires maintenance of various policy metadata throughout the abstraction process. Chapter 8 provides a case study of the application of the Hierarchical Security Policy Formal Model to the Role-Based Access Control system of OpenStack, and details the techniques described in Chapter 5, Chapter 6, and Chapter 7. Chapter 9 provides insight into potential future directions of the Hierarchical Security Policy Formal Model. Chapter 10 concludes this thesis.

Chapter 2: Related Work

2.1 Role Based Access Control

The work by Sandhu et al. [15] provides model of a Role-Based Access Control. As an alternative to discretionary or mandatory access control, Role-Based Access Control provides a more concrete mapping of enterprise-level positional mappings to system security policy accesses. The core sets of entities used by most Role-Based Access Control systems are Users, Roles and Permissions. Additionally, the concepts of Objects and Sessions are often included in Role-Based Access Control systems. Users are actual human beings, though this entity set may be generalized to encompass any entity with access to the system, autonomous agents or robots included. Roles represent job functions or titles within an organization and often confer authority or responsibility related to Users who hold the Role in a system. Permissions allow specific mode of access to objects of the system, used synonymously with access right, authorization or privilege. Sessions allow instantiation of User Permissions on a system by activating one or many roles they are a member of [15]. The Role-Based Access Control system must have some sense of what it controls access to, identifying Objects as an separate entity set outside the bounds of the role assignment system but related in that Permissions grant access to them. Objects may represent both data object or resources of the computer system [15].

The core definitions for Role-Based Access Control are as follows [15]:

U, R, P, and S which represent sets of Users, Roles, Permissions and Sessions respectively,

$PA \subseteq P \times R$, a many-to-many permission to role assignment,

$UA \subseteq U \times R$, a many-to-many user to role assignment,

$user : S \rightarrow U$, a function mapping each session s_i to a single user $user(s_i)$, and

$roles : S \rightarrow 2^R$, a function mapping each session s_i to a set of roles $roles(s_i) \subseteq \{r \mid (user(s_i),\ r)\ \in\ UA\}$ and session s_i has the permissions $U_{r\ \in\ roles(s_i)}\ \{p \mid (p,\ r)\ \in\ PA\}$

These definitions expect each role is assigned at least one permission, and each user to be assigned to at least one role, though this expectation is not a requirement of the model [15].

The purpose of an organization as a whole is unlikely to change as often as user positions in the organization, so system access policies are abstracted into various roles throughout the system to which users are assigned as needed [15]. Role-Based Access Control has risen as a extensible way to provide access control to complex systems with many users and resources [10]. However, issues arise in rRole-Based Access Control systems where sequences of operations must be performed by members of roles

in which their membership cannot overlap. A common example of this is an interaction between a Requisitioning role and a Purchasing role. Despite this, Role-Based Access Control allows for simpler implementation in the enterprise environment and ease of use whilst implementing policies supporting concepts such as least privilege, separation of privilege and separation of duty [14] [7]. Additionally, and usefully for the Hierarchical Security Policy Formal Model process,Role-Based Access Controll systems often provide built-in data abstraction with regard to user privileges, as these are baked into whatever roles the user is a member of. This allows the process to concern itself separately with the accesses allowed to the roles and the users in each role. This is discussed in greater detail in Chapter 5. It is important for the Hierarchical Security Policy Formal Model to enable system security policy managers to utilize the model for management of Role-Based Access Control systems.

2.1.1 The Role Graph Model and Conflict of Interest

The work by Nyanchama and Osborn [10] utilizes directed acyclic graphs to represent the roles present in a Role-Based Access Control system [10]. The Hierarchical Security Policy Formal Model uses a similar graph structure to store information about the Subjects of a system security policy, which, as described in Chapter 5, includes the roles available to users of a system. The work by Nyanchama and Osborn also includes some allusions to the hierarchy of Role-Based Access Control systems, in that the nodes of the directed acyclic graphs used in this work to represent roles inherit access from their parents, which allows accesses to be tracked from the least role in the system to the greatest [10]. The model displayed by Nyanchama and Osborn is used to determine possible conflicts of interest, defined as the result of the policy abstraction presented where certain unwanted sets of accesses are present. This enables analysis of security concepts such as separation of duty, mutual exclusion, separation of privilege and least privilege [10] [14]. While similar to the Hierarchical Security Policy Formal Model, the work by Nyanchama and Osborn only pertains to Role-Based Access Control, whereas the Hierarchical Security Policy Formal Model sees more broad application [3] [17].

2.1.2 Role-Based Authorization Constraints Specification

The work by Ahn and Sandhu [2] focuses on constraint specifications, an important part of Role-Based Access Control, using the Role-Based Constraints Language 2000, RCL2000 [2]. This language builds off previous work by the same authors detailing a similar Role-Based Access Control specification language RSL99, the Role-Based Separation-of-Duty Language 99 [1] [2]. This language provides methods for translation to a restricted form of first order predicate logic, which provides a formal language for

denoting Role-Based Constraints on the system. These constraints can be used to evaluate potential conflicts of interest, similar to the role graph model [10] [1]. This work possibly provides an additional layer atop the Hierarchical Security Policy Formal Model for automated testing of system security policy against formal definitions of system security policy in the Role-Based Constraints Language. While Role-Based Authorization Constraints Specification focuses on providing formal definitions of separation of duties principles, the Hierarchical Security Policy Formal Model focuses on providing easy of use to the system security policy managers through the logical database and HERMES, in addition to formal policy modeling and verification.

2.1.3 Role-Based Access Control Verification for Real-Time Systems

The work by Masood et al. [7] presents a colored Petri-net based framework for verifying consistency in Role-Based Access Control policies. The application of this framework allows this work to verify the consistency of Role-Based Access Control policies using the Petri-net reachability analysis [7]. Similar to the Hierarchical Security Policy Formal Model, this work presents methods by which system security policy may be analyzed in an abstract manner. Where the Hierarchical Security Policy Formal Model process relies on the use of policy paths over directed acyclic graphs of subjects, actions and objects, Role-Based Access Control Verification for Real-Time Systems utilizes directed bipartite graphs of transitions and places known as Petri-nets [11]. Petri-nets are often used to abstract information flow and, with the work presented, are used to model limits of access to users of a Role-Based Access Control system. The methods presented in Role-Based Access Control Verification for Real-Time Systems are similar to those used by the Hierarchical Security Policy Formal Model process, though no accompanying policy specification language such as HERMES is available to manage system security policy configurations in conjunction with the analysis tools provided.

2.2 High Fidelity Browser Policy

The work by Jillepalli and Conte de Leon [5] detailing a High Fidelity Browser Management System provides a high level policy specification language and environment, security policy detection and issue resolution, and distributed configuration deployment enabling automated application of high level policies [5]. The High-Level Easy-to-Use Reconfigurable Machine Environment Specification language, HERMES, is used to provide these functions to system security policy managers [6] [5]. This work is closely related to the research presented in this thesis, and has similar end goals. Whereas the Hierarchical Security Policy Formal Model research begins by examining system security policy artifacts

and abstracts them into policy models used for analysis, the High Fidelity Management System begins with the high-level specification of the system security policy and seeks to provide the ability to push modifications down to system components, specifically browsers.

2.3 HERMES Language

Additionally, Jillepalli, Conte de Leon, Steiner and Sheldon [6] detail a High-Level Easy-to-Use Reconfigurable Machine Environment Specification language, HERMES, which seeks to provide a security policy specification language enabling specification of domain security policies in a hierarchical manner. This language attempts to do so in a generic fashion enabling applicability to many devices at various levels of granularity. It also seeks to bridge the gap between enterprise-level security policy and the actual system component security configuration [6]. The HERMES language was developed in conjunction with the High Fidelity Policy Management System described previously [5] [6].

The work described in this thesis is similar to the HERMES language as the end result of both allows the system security policy managers the ability to manage their system security policy at an abstract level. Additionally, the HERMES language can be used to query the Hierarchical Security Policy Formal Model logical database [3] [17]. Similarly to the High Fidelity Policy Management System, the HERMES language is to be used at the enterprise level to dictate system security policies which are then pushed down to individual devices. This differs from the approach of the Hierarchical Security Policy Formal Model process in that the process described in this thesis begins with the system security policy artifacts and provides layers of abstraction to aid system security policy managers with analysis of system security policy.

2.4 Formal Verification and Visualization of Security Policies

The work by Wahsheh, Conte de Leon and Alves-Foss [16] utilizes a graph-based visualization tool which is used to validate policies and provide system security policy managers with analytics for policy review and subsystem interactions. Similar to the contributions presented in this thesis, this work presents methods for system security policy analysis, but focus on the entities in a secured system, their individual security levels, various security enclaves of the system and the interactions between these three. Similar to the Hierarchical Security Policy Formal Model process, this work constructs a Prolog logical database which can have queries ran against it to verify system security policy [16]. The application of security policy hierarchy in the models presented by this work differ from those included in the Hierarchical Security Policy Model; this work presents hierarchy as a result of increasing classification levels and

security enclaves of the same classification level whereas the Hierarchical Security Policy Formal Model focuses on the hierarchy inherent to a disparate system of devices and their interactions. This work uses a Simple Hierarchical Multi-Perspective tool, SHriMP, to aid visualization. This tools is designed to enable exploration of complex software programs and knowledge bases in a nested, hierarchical fashion [4].

2.5 Hierarchical Security Policy Formal Model Prior Research

To properly address the needs of system security policy management personnel the Hierarchical Security Policy Formal Model project must address more than just Role-Based Access Control. System security policy management personnel are often given responsibility over every level of system security on an Enterprise network, and the Hierarchical Security Policy Formal Model project seeks to provide application at every level. Additionally, system security policy management personnel are required to be knowledgeable about the interactions between different levels of their security domain, requiring insight into the interactions between system policy configurations and enterprise policy. The Hierarchical Security Policy Formal Model project seeks to provide methods by which system security policies can be merged and examined concurrently, providing system security policy managers tools which simplify interactions in this complex environment. The Hierarchical Security Policy Formal Model is further described in Chapter 3.

2.5.1 Device Configuration Verification: Router Policies

The work by Brown [3] provides application of the Hierarchical Security Policy Formal Model to Cisco router configurations for site-to-site VPN scenarios. In this application, the hierarchical policy model track the ability of different origin IP addresses to send network packets across a VPN. The Hierarchical Security Policy Formal Model is applied by treating source IP addresses as the Subjects of the model, various router actions required by VPN traffic, such as send and encryption, as hierarchical policy model Actions, and destination IP addresses as Objects. In effect, this application of the Hierarchical Security Policy Formal Model checks if source IP address ranges are able to connect through the VPN and access the destination IP address ranges. Application of the hierarchical security policy model in this way simplifies the system for network administrators seeking to evaluate router policies [3].

2.5.2 Operating System Verification: Security Enhanced Linux

The work by Zook [17] provides application of the Hierarchical Security Policy Formal Model to auditing operating system level access control policy mechanisms by analyzing Security Enhanced Linux

policies. To apply the hierarchical security policy model in this way, the Security Enhanced Linux roles and user types are grouped into the Subjects of the model, all operating system permissions checked by Security Enhanced Linux are added as Actions, and file types under the operating system's control are designated as Objects. This use of the model allows the permissions of a user in an operating system to be audited by a systems administrator, and can give a clear view of user permissions on the system [17].

Due to the underlying annotated graph used by the hierarchical security policy model, operations may be performed on the graph which increase the ability of system security policy managers to effectively and efficiently manage policies. The work by Zook explores the ability to merge full Hierarchical Security Policy Formal Model policies by mapping similar policy artifacts through multiple policies and combining the hierarchical security policy graphs based on this mapping [17]. This is similar to the contribution Formal Policy Model Composition, discussed in Chapter 4 and Chapter 6 of this thesis. The contribution this thesis makes deals with the composition of hierarchical policy model subpolicies; policies which may cover a combination of Subjects, Actions and Objects from the same system, unlike Merge, which operates on full policies from separate systems [17].

Chapter 3: Background

3.1 The Hierarchical Security Policy Formal Model

The Hierarchical Security Policy Formal Model project, referred to as HPol or the hierarchical policy model, is an ongoing research project at the University of Idaho which seeks to address increases in complexity of security policy management. This is accomplished through a layered, hierarchical, model which ultimately provides system security policy managers an abstracted interface to verify multiple static security policies across similar computer systems, networks, or services. For the purposes of this thesis, the Hierarchical Security Policy Formal Model is comprised of three primary partially ordered element sets representing the Subjects, Actions, and Objects of the system. These element sets are ordered individually based on the hierarchy of the target system, for example, an organization chart provides an enterprise-level mapping of a Subject Hierarchy. These element sets are then evaluated by the system security policy authenticator, an oracle which allows or disallows access to the system. This can be described as evaluating *who*, a Subject, can perform *what*, an Action, on *which* resource, an Object. This evaluation results in the creation of a set of policy paths, each of which represent an allowed sequence of Subjects, Actions, and Objects. The policy paths may be examined in a variety of ways, but serve to provide the system security policy managers with a simple interface for verification with the target system.

Figure 3.1 displays a generic Hierarchical Security Policy Formal Model for a simple system. In the system modeled, there exists only one element of each element set: 1) User, a Subject, 2) Activity, an Action, and 3) Resource, an Object. The complete policy path 1001 traverses the Subjects by passing through User, traverses the Actions by passing through Activity, and traverses the Objects by passing through Resource. Thus, this generic model displays the ability of User to perform Activity with Resource.

The model begins by using a forest of directed acyclic graphs to group the Subjects, Actions and Objects of concern from a system security policy. This system security policy may potentially be comprised of multiple subsystem security policies, each of which may govern one or more subgraphs in the model. The hierarchical policy model then evaluates the ability of each Subject to perform each Action on each Object, creating an additional layer of policy paths atop the previously constructed directed acyclic graphs. Notably, this step introduces two predefined nodes, Start and End, which denote the beginning and ending of a policy path. For the hierarchical policy model to consider policy paths complete, they must start at the Start node, contain at least one of each: Subject, Action and Object, and finish at the End node. Hierarchical policy model paths are numbered and unique, though it is possible to create wildcard links to preserve the semantics of the original system policies, and to reduce complexity

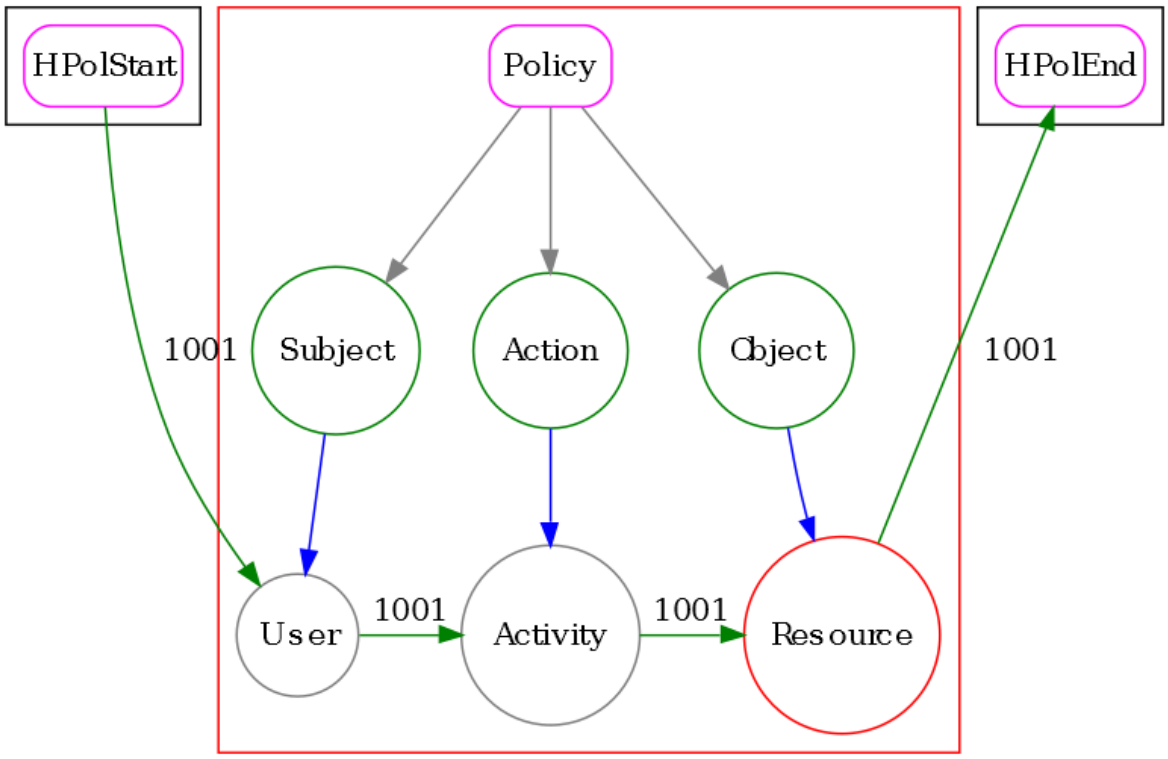


Figure 3.1: A Generic Hierarchical Security Policy Formal Model

while viewing a the graphical depiction of the hierarchical policy model. The use of wildcard policy links indicates the ability of any policy path to include the link, and can be used to show satisfaction of rules or to illustrate required steps in a process. Additionally, wildcard policy links may be used to provide abstraction of the system security policies, improve usability or decrease the complexity of the hierarchical security policy model. Finally, the policy paths and the direct acyclic graphs are used create a query engine which accepts human readable input questions and can query the hierarchical policy model for answers.

For example, the OpenStack cloud computing software [13] has a service for user identity, Keystone, which manages user's roles and access levels in the system. The OpenStack Keystone service provides everything required to statically assess and query a user's access to the OpenStack system, providing everything required by the Subject group of the Hierarchical Security Policy Formal Model. OpenStack also has numerous cloud computing related services, such as the block storage service Cinder, which manages storage allocation for different projects on an OpenStack deployment, each with their own

security policy. The other service, in this case Cinder, provides a list of the actions potentially available to the user of the service and the service component these actions will be performed upon; everything required to statically assess the Action and Object groups required by the Hierarchical Security Policy Formal Model. After identifying the Subjects, Actions and Objects, each of these groups are placed in a directed acyclic graph.

Once the directed acyclic graphs have been created for the Subjects, Actions and Objects, the hierarchical policy model moves to evaluate access allowed to each user in the Subject graph. OpenStack uses Role-Based Access Control to manage user authorization and determine if a user is able to perform an action on an object given the user's role and the user's project. This Role-Based Access Control allows the hierarchical policy model to evaluate user access to Actions and Objects by analyzing the user, the user's role, and the project in which the user has this role. The hierarchical policy model can then create policy paths based on the directed acyclic graphs and pass both the policy paths and the directed acyclic graphs on to the query engine. The process of creating hierarchical security policy models from OpenStack is discussed in greater detail in Chapter 5.

3.1.1 Hierarchical Security Policy Formal Model Merging

Due to the underlying annotated graph used by the Hierarchical Security Policy Formal Model, operations may be performed on the graph which increase the ability of system security policy managers to effectively and efficiently manage policies. The work by Zook [17] explores the ability to merge full Hierarchical Security Policy Formal Model policies by mapping similar policy artifacts through multiple policies and combining the hierarchical security policy graphs based on this mapping [17]. This is similar to the contribution Formal Role-Based Access Control Policy Composition, discussed in Chapter 4 and Chapter 6 of this thesis, though the contribution this thesis makes deals with the combination of hierarchical policy model subpolicies; policies which may cover a combination of Subjects, Actions and Objects, but not all three.

Ongoing research in this area seeks to provide a generic method by which to merge hierarchical security policy model graphs, and methods to perform policy merges on policies derived from different devices or systems.

3.2 Role Based Access Control

As an alternative to Discretionary or Mandatory Access Control, Role-Based Access Control provides a simplified mapping of enterprise-level positional mappings to system security policy accesses. The

core sets of entities used by most Role-Based Access Control systems are Users, Roles and Permissions. Additionally, the concepts of Objects and Sessions are often included in Role-Based Access Control systems. Users are actual human beings, though this entity set may be generalized to encompass any entity with access to the system, autonomous agents or robots included. Roles represent job functions or titles within an organization and often confer authority or responsibility related to Users who hold the Role in a system. Permissions allow specific mode of access to objects of the system, used synonymously with access right, authorization or privilege. Sessions allow instantiation of User Permissions on a system by activating one or many roles they are a member of [15]. The Role-Based Access Control system must have some sense of what it controls access to, identifying Objects as an separate entity set outside the bounds of the role assignment system but related in that Permissions grant access to them. Objects may represent both data object or resources of the computer system [15].

As mentioned in Chapter 2, the core definitions for Role-Based Access Control are as follows [15]:

U, R, P, and S which represent sets of Users, Roles, Permissions and Sessions respectively,

$PA \subseteq P \times R$, a many-to-many permission to role assignment,

$UA \subseteq U \times R$, a many-to-many user to role assignment,

$user : S \rightarrow U$, a function mapping each session s_i to a single user $user(s_i)$, and

$roles : S \rightarrow 2^R$, a function mapping each session s_i to a set of roles $roles(s_i) \subseteq \{r \mid (user(s_i),\ r)\ \in\ UA\}$ and session s_i has the permissions $U_{r\ \in\ roles(s_i)}\ \{p \mid (p,\ r)\ \in\ PA\}$

These definitions expect each role is assigned at least one permission, and each user to be assigned to at least one role, though this expectation is not a requirement of the model [15].

Figure 3.2 displays a generic Role-Based Access Control system as modeled by the hierarchical policy model. This example system has one of each of the following policy artifacts: User, Role, Permission, Activity and Resource. Note that in this example Role-Based Access Control system, the combination of User and Role constitutes a Session. The policy artifacts are arranged into their appropriate Hierarchical Security Policy Formal Model element sets, Subject, Action, and Object. In this system, *Role* has been given access to *Permission*, and *User* has been given access to *Role*. Permission gates access to *Activity* and *User* wishes to perform *Activity* on *Object*. The complete policy path 1001 traverses the Subjects through *User* and *Role*, traverses Action through *Permission* and *Activity*, and traverses Object through *Resource*. This complete policy path denotes the assignment of *User* to *Role*, the assignment of *Role* to *Permission*, and the ability of *User* to perform *Activity* on *Object*.

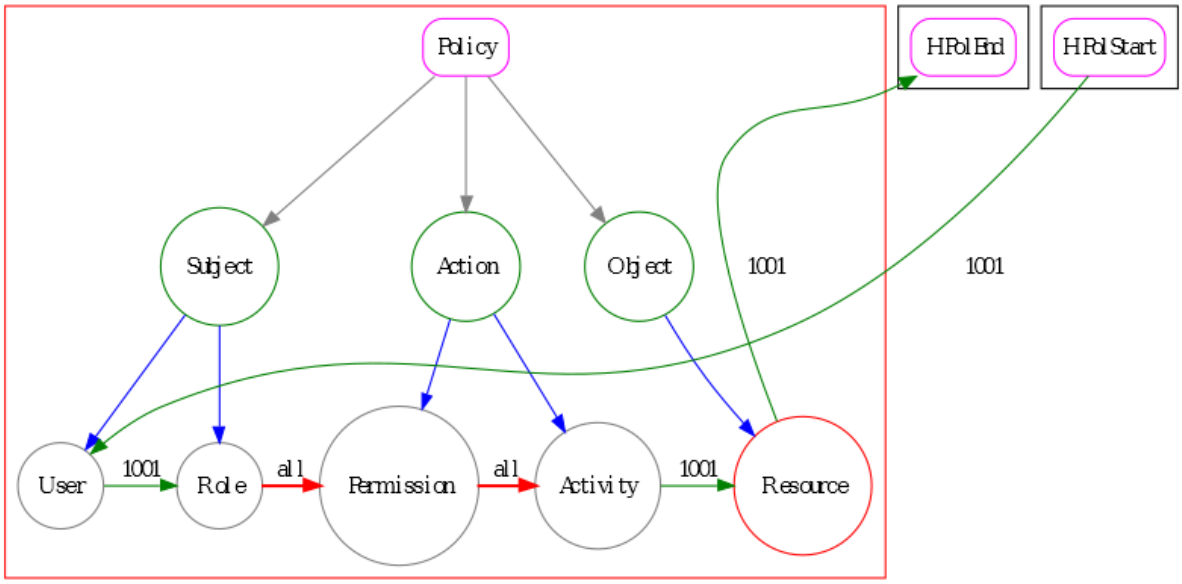


Figure 3.2: A Role-Based Access Control Formal Model

3.3 OpenStack

OpenStack is an open source cloud computation platform [13] which allows users to control compute, storage and network resources at the datacenter scale, through either a dashboard or remote API. OpenStack offers a virtual layer atop an existing heterogeneous physical datacenter environment, and allows the datacenter administrators to allocate resources between projects using the datacenter as a cloud service provider. OpenStack is organized into separate services, each of which controls one of the multiple facets required of a cloud computation platform such as user identity, compute, storage or network [13].

Core to these security policies is the identity service, *Keystone*. This service provides the user authorization, specifically which roles users have in which projects, enabling the Role-Based Access Control used by the rest of the OpenStack deployment to successfully authenticate and authorize users of the system. Each OpenStack service provides its own resources for allocation to the projects of the OpenStack deployment, as well as per-service policies governing access to allocate and modify the resources of

the service. The individual OpenStack services each provide a JSON formatted policy file [12] to manage Permissions and Actions available on the service. The elements of these policies file are key-value pairs, with the key either representing a Permission alias or a target API function call. The values of both types of keys are ultimately Permission definitions, though layers of indirection may be used to delegate Permission accesses to other Permissions or combinations of Permissions.

Permission definitions can be [12]:

1. always true, written as the empty string, [ ] or "@",
2. always false, written as "!",
3. a special check, described below,
4. a comparison of two values from the second list below, or
5. a boolean expression of other Permission definitions,

Special checks as mentioned above are [12]:

1. $< role >:< rolename >$, testing if API credentials contain this role,
2. $< rule >:< rulename >$, testing an aliased rule, or
3. $http : // < targetURL >$, delegating the boolean check to a remote server,

Possible values to compare as mentioned above are [12]:

1. constants: Strings, numbers, True and False,
2. API attributes,
3. target object attributes, or
4. the flag *is_admin*

Allowed API attributes are *project_id*, *user_id* and *domain_id*. Target object attributes are held in the service's database a fields of the Object being targeted by the access. This can be used to check if the Object being accessed belongs to the same project as the User Session attempting the access. The *is_admin* flag tests if the Session has been granted the admin token, which grants administrative privileges similar to the admin role and allows initialization of the Keystone identity database before any roles exist in the OpenStack environment [12].

For the purposes of the Hierarchical Security Policy Formal Model, OpenStack provides a well-documented exemplar of Role-Based Access Control. OpenStack's Keystone identity service primarily deals with users, roles and projects. An OpenStack user is a single OpenStack account for a given OpenStack deployment, and can store details about the user such as unique user ID, hashed password and email address. An OpenStack role is an attribute associated with a user in the context of a specific project, and has a unique role ID. The role allows the user access to any actions permitted by holders of that role. An OpenStack project designates a subset of assigned data center resources controlled by a given OpenStack deployment, and each has a unique project ID. OpenStack projects may also be referred to as "tenants" of an OpenStack deployment. User to role assignments in OpenStack require a projects to be named in which the user will hold the role. By giving a user a role in a project, the user is given access to all actions permitted by that role with regard to the project in which the user holds the role. For example, a user *Alice* is given the role of *Member* on the project *Work*. *Alice* can perform any action allowed to regular users on the resources dedicated to the *Work* project by the OpenStack administrator. Role names may be the same, but roles are assigned to users on a per-project basis. If a user *Bob* is given the role *Member* on the project *Play*, he may perform any action allowed to *Members* on the resources of the *Play* project, but has no access to resources controlled by the *Work* project, despite having the same role as *Alice*. As such, individual users may hold multiple roles in different projects on the same OpenStack deployment, but only have access to the resources handled by their projects in the capacity that their roles in those projects allow. This construction of user, role, and project forms the basis of the Role-Based Access Control mechanism utilized by OpenStack to authenticate user interactions in the system.

Being open source, OpenStack also offers direct insight to the methods by which system security policy is evaluated, allowing a clear and accurate translation of system security policy from the actual OpenStack software deployment to the Hierarchical Security Policy Formal Model. Additionally, the method OpenStack uses to isolate its system security policy per service is used in this thesis to demonstrate addition and composition of Hierarchical Security Policy Model subpolicies, and illustrate the requirements of the Hierarchical Security Policy Model to maintain individual policy artifact origins through such an operation.

Chapter 4: Research Questions, Objectives and Contributions

4.1 Research Question and Objectives

The goal of the Hierarchical Security Policy Formal Model is to provide system security policy managers with an effective and efficient workflow to manage their system security policy. This is accomplished through abstraction of the system security policy artifacts to a single system security policy for the target system. This abstraction would cover an entire enterprise system domain and should be able to adapt to provide the system security policy manager for this domain the ability to analyze and manage the entire system domain without reductions to usability. Ideally, the Hierarchical Security Policy Formal Model project seeks to provide an abstraction which can incorporate multiple system security policies, account for the interactions between disparate system policies, and provide the system security policy manager an interface for system security policy analysis and management.

With these overarching objectives in mind, the research questions of this thesis are thus:

1. Can the Hierarchical Policy Model be applied to Role-Based Access Control systems?
2. Is there a method by which the Hierarchical Policy Model can combine subpolicies from separate parts of a target system to form a complete policy?
3. Can the Hierarchical Policy Model return the original system security policy artifacts from the constructed system security policy model?

The first of these questions seeks to answer a piece of the overall problem of applicability of the Hierarchical Security Policy Model as successful application of the model to Role-Based Access Control systems in a generic fashion demonstrates the power and potential of the model to aid system security policy managers in a variety of systems. The second question addresses a different facet of the applicability problem in that currently the Hierarchical Security Policy Formal Model has no solid definition for the original state of system security policy artifacts, nor does the model have a codified method by which to combine subsystem policies. While research has been performed demonstrating the methods required to merge entire hierarchical security policy models [17] of similar systems, the objective of this thesis is concerned with an operation similar to merge which operates at the subpolicy level. The third research question prompts and increase in scope of the Hierarchical Security Policy Formal Model, as the model currently only supports the analysis of system security policies through analysis of the directed acyclic graphs, overlaid policy paths or through querying the logical database. Providing a successful answer to the third research question improves the Hierarchical Security Policy Formal Model as it would allow the model to be used for system security policy management in addition to the analysis currently supported.

These three research questions prompt the following research objectives:

1. Provide a generic method by which the Hierarchical Security Policy Formal Model may be applied to Role-Based Access Control systems, and provide an instance of this application to an actual Role-Based Access Control system.

2. Provide a method by which separate policies from a common target system may be combined into a single system-level security policy model.

3. Provide a method by which potentially modified system security policy artifacts may be reconstructed from a system security policy model.

As this thesis addresses the applicability of the Hierarchical Security Policy Formal Model to Role-Based Access Control systems through examination of the OpenStack cloud computation platform, this thesis addresses the combination of subsystem security policy models and reconstruction of system security policy artifacts through examination of OpenStack as well.

4.2 Contribution 1: Role-Based Access Control Policy Modeling

First, this thesis addresses the applicability of the Hierarchical Security Policy Formal Model to Role-Based Access Control systems. It is important to seek application of the model where it may be most impactful for system security policy managers in an enterprise environment. Analysis of the applicability of the Hierarchical Security Policy Formal Model to Role-Based Access Control systems is necessary as many popular enterprise-level systems use some form of Role-Based Access Control to manage user access to their systems. This research question emerges to test, in part, the applicability of the Hierarchical Security Policy Formal Model to real systems. The system targeted in this thesis, the OpenStack cloud platform, utilizes Role-Based Access Control to manage access to various resources managed by the system.

This thesis presents an application of the Hierarchical Security Policy Formal Model to Role-Based Access Control policies both in a generic fashion and with respect to the OpenStack cloud platform. Describing application of the process to Role-Based Access Control systems in a generic fashion is included to lay the groundwork for future applications of the model. The applicability of the Hierarchical Security Policy Formal Model to Role-Based Access Control systems is discussed in greater detail in Chapter 5.

4.3 Contribution 2: Hierarchical Security Policy Composition

Second, this thesis seeks addresses the ability of the Hierarchical Security Policy Formal Model to be expanded to combine system subpolicies originating from separate parts of the same system. This topic stems from two concerns: modern systems are complex, and modern systems incorporate separation to better facilitate software development and secure operation. Both of these concerns cause the software system security policy, with regard to the hierarchical security policy artifacts of the system, to become spread across multiple devices or logical areas of the software system. This requires adaptation by the Hierarchical Security Policy Formal Model in order to display applicability to these systems, prompting this research topic. This thesis presents an addition to the Hierarchical Security Policy Formal Model to accommodate these complex systems in the form of a subpolicy Composition operation. This operation is similar to the Merge [17] and Concatenate [3] operations supported by other applications of the Hierarchical Security Policy Formal Model in that it combines two directed acyclic graphs, though is different in that the combination of graphs to be combined need not represent entire system security policies. This expansion to the Hierarchical Security Policy Formal Model codifying the Composition of system security policy with separated system security policy artifacts and policy files is detailed in Chapter 6.

4.4 Contribution 3: Policy File Reconstruction

Finally, this thesis seeks to address the ability of the Hierarchical Security Policy Formal Model to reconstruct potentially modified system security policy artifacts into deployable system security policy files. Recall how the modeling process begins by identifying and extracting system security policy artifacts from the systems the model is to be applied to. From the extracted system security policy artifacts, the hierarchical security policy model then creates a set of directed acyclic graphs and forms policy paths over these graphs representing system users accesses in the system. The process completes by abstracting the directed acyclic graphs and overlaid policy paths into a logical database, which the system security policy manager may query to determine user access to the system. The hierarchical security policy model process provides system security policy managers with an abstract view of the system, simplifying their analysis of disparate system components. This topic seeks to provide system security policy managers not only the ability to analyze their systems, but the ability to manage their systems from the abstract view provided by the hierarchical security policy model. This is accomplished by expanding the ability of the Hierarchical Security Policy Formal Model to tag the origin of elements used to construct the directed acyclic graphs and their overlaid policy paths. Chapter 7 describes Policy File Reconstruction.

Chapter 5: Role-Based Access Control Policy Modeling

Before describing the Hierarchical Security Policy Formal Model process in its application to Role-Based Access Control policies, it is important to form a mapping between the Role-Based Access Control model entity sets and the elements of the Hierarchical Security Policy Formal Model. This thesis will use the following notation to denote various set-based operations:

X, denoting a set,

$X \times Y$, denoting Cartesian product of sets,

$X \setminus Y$, denoting set difference,

$X \prec Y$, denoting precedence for orderings,

$X \subseteq Y$, denoting X is a subset or equivalent set of Y,

$X \rightarrow Y$, denoting a function mapping between sets,

2^X, denoting the power set of a set; the set of all subsets of a set,

$\mathbf{X} = (X, P)$, denoting a partially ordered set with X as a ground set and P as the partial ordering,

A chain of $\mathbf{X}$ contains only elements which are comparable to each other,

$\mathbf{X}^c$, denoting the set of all chains in $\mathbf{X}$

Additionally, the term poset may be used as shorthand for partially ordered set. This shorthand is used in listings to improve readability by reducing line breaks, though the non-listing content uses partially ordered set.

Recall the core definitions for Role-Based Access Control [15]:

U, R, P, and S which represent sets of Users, Roles, Permissions and Sessions respectively,

$PA \subseteq P \times R$, a many-to-many permission to role assignment,

$UA \subseteq U \times R$, a many-to-many user to role assignment,

$user : S \rightarrow U$, a function mapping each session s_i to a single user $user(s_i)$, and

$roles : S \rightarrow 2^R$, a function mapping each session s_i to a set of roles $roles(s_i) \subseteq \{r \mid (user(s_i),\ r) \in UA\}$ and session s_i has the permissions $U_{r \in roles(s_i)} \{p \mid (p,\ r) \in PA\}$

These definitions expect each role to be assigned at least one permission, and each user to be assigned at least one role, though this expectation is not a requirement of the Role-Based Access Control model [15]. This covers the case where a User has no Roles or a Role grants no Permissions.

The Hierarchical Security Policy Formal Model may also be described in a similar set notation for individual full policies:

Sub, Act, and Obj which represent sets of Subjects, Actions and Objects respectively,

$\mathbf{S} = (Sub, Sub_{\prec})$, a poset of Subjects ordered by the Subject Hierarchy with chains $\mathbf{S}^c$,

$\mathbf{A} = (Act, Act_{\prec})$, a poset of Actions ordered by the Action Hierarchy with chains $\mathbf{A}^c$,

$\mathbf{O} = (Obj, Obj_{\prec})$, a poset of Objects ordered by the Object Hierarchy with chains $\mathbf{O}^c$, and

$PP \subseteq \mathbf{S}^c \times \mathbf{A}^c \times \mathbf{O}^c$, the set of hierarchical policy paths

The definitions of the partial orderings of the partially ordered sets are left up to the instantiation of the Hierarchical Security Policy Formal Model, though they should represent a partial order in the hierarchy of the system security policy being modeled.

Once the Hierarchical Security Policy Formal Model has been described this way, a mapping to Role-Based Access Control becomes apparent:

$\mathbf{S}_U = (S_U, Sub_{\prec}) \subseteq \mathbf{S}$, the poset of Users U of a system ordered by the Subject Hierarchy,

$\mathbf{S}_R = (S_R, Sub_{\prec}) \subseteq \mathbf{S}$, the poset of Roles R of a system ordered by the Subject Hierarchy,

$\mathbf{S}_S = (S_S, Sub_{\prec}) \subseteq \mathbf{S} \setminus (S_U \cup S_R)$, the poset ordered by the Subject Hierarchy of additional data used to complete a Session on the system,

$\mathbf{A}_P = (A_P, Act_{\prec}) \subseteq \mathbf{A}$, the poset of Permissions P of the system ordered by the Action Hierarchy,

$PA \subseteq \mathbf{S}_R^c \times \mathbf{A}_P^c$, the role-to-permission assignment with chains of $\mathbf{S}_R$ and $\mathbf{A}_P$, and

$UA \subseteq \mathbf{S}_U^c \times \mathbf{S}_R^c$, the user-to-role assignment with chains of $\mathbf{S}_U$ and $\mathbf{S}_R$

Note that $\mathbf{A}_P$, the set of Permissions, need not contain every Hierarchical Security Policy Formal Model Action element. The remainder of the set of Actions denote activities Users may take on the system, gated by their assigned Permission. This thesis separates the Permissions and the rest of the Action elements without explicitly naming the remainder, though a name such as Activities may be appropriate depending on the system being modeled. This mapping also leaves the Subject Hierarchy and Action Hierarchy up to the instantiation of the Hierarchical Security Policy Formal Model. The default orders for

this system attempt to express the semantics of Role-Based Access Control such that Users are combined with Roles and form Sessions, and Permissions grant access to non-Permission Actions of the system.

$$Sub_{\prec} = \mathbf{S}_U \prec \mathbf{S}_R \prec \mathbf{S}_S$$

$$Act_{\prec} = \mathbf{A}_P \prec \mathbf{A} \backslash \mathbf{A}_P$$

The default partial order for Hierarchical Security Policy Formal Model Objects $Obj_{\prec}$ is left unable to relate any two elements in Obj as the hierarchy for Objects is left up to the system managed by the Role-Based Access Control, not the Role-Based Access Control itself.

Additionally, the definition of the policy path set PP is refined to represent the presence of the new subsets of the various policy elements. This refinement replaces the chains of $\mathbf{S}$ with the chains of its subsets, and prepends $\mathbf{A}$ with the chain of Permissions:

$$PP \subseteq \mathbf{S}_U^c \times \mathbf{S}_R^c \times \mathbf{S}_S^c \times \mathbf{A}_P^c \times (\mathbf{A} \backslash \mathbf{A}_P)^c \times \mathbf{O}^c$$

As the Hierarchical Security Policy Formal Model takes a static view of the system security policy, temporal assignments to Sessions are evaluated simultaneously; every Session is evaluated in the policy path set PP by enumerating each combination of chains in the subsets of $\mathbf{S}$. The new definition of PP organizes the user Session in the subsets of $\mathbf{S}$, and enumerates their accesses to chains of $\mathbf{O}$ through chains of $\mathbf{A}_P$ and $\mathbf{A}$.

Figure 5.1 displays an example Role-Based Access Control policy model incorporating divergent complete policy paths as well as hierarchy in both the Permissions and Roles of the system. The displayed system contains one User, two roles, *HighRole* and *LowRole*, two Permissions, *HighPermission* and *LowPermission*, one Activity, and one Resource. In the system, *HighRole* has access to *HighPermission*, and *LowRole* has access to *LowPermission*. Both *HighRole* and *HighPermission* are hierarchically superior to their Low counterparts. *User* has been given *HighRole*, and *LowPermission* gates access to *Activity*. Note that policy path 1001 diverges after *HighRole*; it is able to either travel further through *LowRole*, or through *HighPermission*. Either way, 'policy path 1001 verifies *User*'s ability to perform *Activity* with *Resource*.

5.1 Extracting System Security Policy Artifacts From Policy Files

To begin application of the Hierarchical Security Policy Formal Model to a Role-Based Access Control system we first seek out the system security policy files of that system. System security policy files need not be strictly textual files stored on a system, rather, policy files in this context may refer to any number of databases, configuration settings or data objects managed by the system, including textual files. Recall

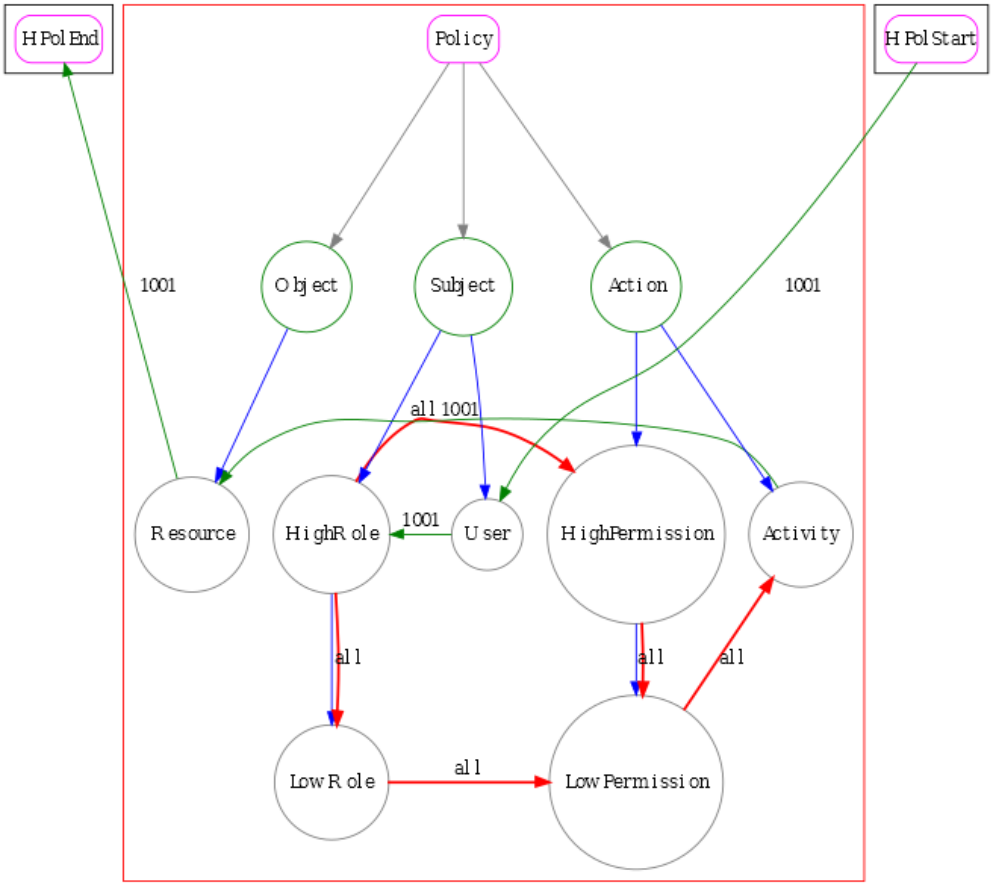


Figure 5.1: A Complex Role-Based Access Control System Policy Model

that one goal of the Hierarchical Security Policy Formal Model process is that the construction of the directed acyclic graphs and overlaid policy paths may occur outside the target system, thus, it is important to identify and extract the system security policy files before the Hierarchical Security Policy Formal Model process is applied. Performing the extraction of system security policy files from the target system before the individual system security policy artifacts are identified is necessary to maintain abstraction between the layers of the application of the Hierarchical Security Policy Formal Model process effecting the target system and providing set identification prior to graph construction. Additionally, performing the extraction of system security policy files before identification of system security policy artifacts allows the entirety of the system security policy file to be maintained by the Hierarchical Security Policy Formal Model for the duration of the process and return similarly whole system security policy files to the target system for redeployment as described in Chapter 7 of this thesis.

After the system security policy files have been extracted from the target system, the Hierarchical

Security Policy Formal Model process seeks to identify system security policy artifacts held within these files. System security policy artifacts are defined as elements of the system security policy files which, in part or in whole, map to elements of the sets or subsets used by the application of the Hierarchical Security Policy Formal Model targeting the system. For the purposes of Role-Based Access Control systems, the Hierarchical Security Policy Formal Model process is primarily concerned with extraction of Users, Roles, Sessions available to Users, Permissions or permissive rules, Actions allowed by these permissions, and Objects on the system. Additionally, the Subject, Action and Object Hierarchies partial orderings specific to the system may be defined as the system security policy artifacts are identified from the system security policy files. As these various system security policy artifacts are removed from their respective system security policy files into the hierarchical policy model, the Hierarchical Security Policy Formal Model process takes care to tag the artifacts with their origin, and notes how to reintegrate extracted system security policy artifacts with their original policy file. As some of the system security policy artifacts may be closely coupled with other data related by the artifact's system security policy file, closely coupled data is also extracted and maintained throughout the Hierarchical Security Policy Formal Model process.

5.2 Identifying and Structuring Hierarchical Security Policy Model Sets

Now that the Hierarchical Security Policy Formal Model process has identified the relevant system security policy artifacts and their closely coupled data, the process continues by organizing the discovered policy artifacts into the element sets used by the Hierarchical Security Policy Formal Model. Recall the core sets of elements used by the Hierarchical Security Policy Formal Model: Subjects, Actions and Objects. Recall too the sets of entities used by Role-Based Access Control: Users, Roles, Permissions, Objects and Sessions [15]. In order to effectively apply the Hierarchical Security Policy Formal Model to Role-Based Access Control systems, the process must identify generic categorizations of the entity sets used to describe Role-Based Access Control. This has the added benefit of guiding the implementation of the Hierarchical Security Policy Formal Model for any instantiation of the process on a real Role-Based Access Control system.

The Hierarchical Security Policy Formal Model process now declares its sets, Subjects, Actions and Objects, as well as their mappings to the entity sets used by Role-Based Access Control. The Subjects set of the policy model is give the subsets User and Role. The Actions set is given the subsets Permissions and Actions. The Objects set simply contains the subset Objects. After the Hierarchical Security Policy Formal Model process has declared this organization of sets and subsets, the system security policy

artifacts extracted in the previous step are sorted into their respective subset. While many of the Role-Based Access Control system entity sets have been strictly classified here, one has not: Sessions. The idea of a user session is difficult to define in the context of the Hierarchical Security Policy Formal Model, as Sessions require some activation [15] and implied temporal element. As the Hierarchical Security Policy Formal Model process evaluates only static access control policy, traditional Role-Based Access Control Sessions are accounted for simultaneously; all possible Sessions available to a User are evaluated during the process. The evaluation of policy performed by the Hierarchical Security Policy Formal Model is similar to the non-deterministic approach taken to define role-based constraint specifications by RSL99 and RCL2000 [1] [2], though both process work for different goals and at different abstraction levels.

Role-Based Access Control systems often exhibit some form of hierarchy in their role assignment patterns, and such patterns are necessary to reflect in the Hierarchical Security Policy Formal Model. To this end, the previously sorted subsets are organized to reflect any level of hierarchy in the original system security policy. This may include, but is not limited to, role assignment hierarchies, permissions hierarchies, and object hierarchies. In representing these hierarchies, the Hierarchical Security Policy Formal Model transforms the sets and subsets of system policy artifacts into directed acyclic graphs with the policy artifacts as children of other policy artifacts or the subsets derived from the Role-Based Access Control model, the subsets as children of the sets used by the Hierarchical Security Policy Formal Model and the sets all joined under a common root parent of the graph. The use of a directed acyclic graph provides the Hierarchical Security Policy Formal Model with the highest degree of adaptability to different access control models while also maintaining the usability of the data structure and logical consistency of policy paths which may run over it. In providing directed acyclic graphs at this abstraction level, the Hierarchical Security Policy Formal Model provides possible hooks for policy evaluation for identification of conflicts of interest, as well as various other operations which rely on this role graph model [8] [9] [10].

5.3 Identifying Hierarchical Security Policy Model Policy Paths

Now that the Hierarchical Security Policy Formal Model process has identified and structured the element sets required by the hierarchical policy model, the process must determine the accesses allowed to the users of the Role-Based Access Control system; the Subjects of the Hierarchical Security Policy Formal Model. Determining accesses is a virtual operation left undefined by the Hierarchical Security Policy Formal Model, with the implementation of the operation specific to the particular instantiation of the hierarchical policy model. Generically, the Hierarchical Security Policy Formal Model process takes in the directed acyclic graphs presented by the previous step and overlays policy paths of nodes atop this

data structure. Additionally, the policy paths created during this step of the Hierarchical Security Policy Formal Model process must start at the predefined Start node and end at a similar End node, both of which are added at the beginning of this step unconnected to any nodes of the directed acyclic graph. The term 'path' used, while applied atop a graph structure, does not indicate a list of nodes adjacent in the graph, rather a policy path of the Hierarchical Security Policy Formal Model denotes a list of nodes in the graph which begin at the Start node, end at the End node, and contain at least one of each: Subject, Action and Object. Implementing this functionality of the Hierarchical Security Policy Formal Model process for Role-Based Access Control systems refines the definition of a policy path to better fit Role-Based Access Control policy semantics. Hierarchical Security Policy Formal Model policy paths in Role-Based Access Control systems must start at the Start node, end at the End node, contain exactly one User, one or more Roles, and any number of other Subjects which collectively form a Session with the User and Roles, contain one or more Permissions, exactly one Action, and exactly one Object.

As with other implementations of the Hierarchical Security Policy Formal Model process, the order of the policy path should be as listed to maintain the semantics of the original system security policy. Notably, Roles included in a policy path should be included in their hierarchical order, if one exists [15], listing the highest access role first such that the lowest Role granted the related Permission is listed closest to the Permission in the policy path. In the same vein, this stricter definition of a Hierarchical Security Policy Formal Model policy path allows Permissions to exhibit hierarchy; Permissions may delegate their access to other Permissions. This is dealt with similarly to the hierarchy exhibited by Roles, the Permission granted by a Session is listed closest to the final node of the Session while the Permission controlling access to an Object is listed closest to the Object. This stricter definition of a policy path fits the Role-Based Access Control model, as each individual unique policy path enumerates exactly one potential access allowed to a User of the Role-Based Access Control system. Through allowing multiple Roles in a policy path, this definition allows system security policy managers accessing Hierarchical Security Policy Model policy paths to view the entirety of the role hierarchy allowed to a User resulting in their access to a specific Permission.

In order to construct policy paths for its specific Role-Based Access Control system, the modeling process must have access to the policy enforcer or authenticator used by the system. This authenticator acts as a oracle for the Role-Based Access Control system, and must be able to evaluate Session access to the Permissions of the system. The authenticator oracle must be able to provide the relations between the Subjects, Actions and Objects of the entire system outside the relations conveyed by the subsystem policy files. To maintain the abstraction of the Hierarchical Security Policy Formal Model process, this authenticator must be able to be used outside the Role-Based Access Control environment, as the

modeling process will extract the authenticator from the system for local use. The authenticator should be able to, using supplied Role-Based Access Control elements, provide a boolean answer denoting access allowed through the Permission to the supplied Session. In its application to the Hierarchical Security Policy Formal Model process, the authenticator is used to construct policy paths links between the Subject, Action, and Object element sets. This method is used to evaluation all possible Sessions against all possible Permissions, Actions and Objects, generating uniquely identified, individual policy paths for each successful evaluation.

Once all possible Hierarchical Security Policy Formal Model policy paths have been evaluated by the process, the directed acyclic graphs and the overlaid policy paths are passed to the logical database which processes and prepares them for system security manager queries [3].

Chapter 6: Hierarchical Security Policy Composition

Recall the three element sets of the Hierarchical Security Policy Formal Model:

Sub, *Act*, and *Obj* which represent sets of Subjects, Actions and Objects respectively,

$\mathbf{S} = (Sub, Sub_{\prec})$, a poset of Subjects ordered by the Subject Hierarchy with chains $\mathbf{S}^c$,

$\mathbf{A} = (Act, Act_{\prec})$, a poset of Actions ordered by the Action Hierarchy with chains $\mathbf{A}^c$,

$\mathbf{O} = (Obj, Obj_{\prec})$, a poset of Objects ordered by the Object Hierarchy with chains $\mathbf{O}^c$, and

Additionally, recall the definition of the Hierarchical Security Policy Formal Model policy path set:

$PP \subseteq \mathbf{S}^c \times \mathbf{A}^c \times \mathbf{O}^c$, the set of hierarchical policy paths.

6.1 Differentiating Composition, Merge and Concatenate

In order to properly define the Hierarchical Security Policy Formal Model Composition operation, it is important to first cover the other operations provided by the Model.

The Concatenate operation proposed by Brown [3] combines two Hierarchical Security Policy Formal Model directed acyclic graphs and the policy paths laid over them. Effectively, this operation provides a method by which the Hierarchical Security Policy Formal Model may reason about connected systems with no immediate interdependence. This operation is described in Matthew Brown's Master's Thesis [3], which uses Cisco network router policy as the target system security policy for its instantiation of the Hierarchical Security Policy Formal Model. The policy targeted by this work implements a VPN tunnel which integrates two individual Cisco network router policies through Concatenation though this operation is designated under Future Work; no formal implementation is described. A possible Concatenation operation described by this work is performed by overlapping the Start and End nodes of similar Hierarchical Security Policy Formal Model policy path sets, effectively $PP_{Concat} \subseteq PP_1 \times PP_2$ for two policy path sets PP_1 and PP_2. Note that this operation does not maintain any semantic data through the Concatenation process, and only links the two policy paths together. The work describing this operation notes that once a Hierarchical Security Policy Formal Model is able to identify its constituent Hierarchical Security Policy Formal Models, the Concatenate operation should be ready for implementation [3].

The Merge operation presented by Zook [17] also operates on entire Hierarchical Security Policy Formal Model directed acyclic graphs and overlaid policy paths. This operation provides a method by which the Hierarchical Security Policy Formal Model may evaluate multiple systems simultaneously, and may be used by system security policy managers to notice discrepancies in system security policies across

multiple machines. This operation was initially described in Zook's Master's Thesis [17], which uses this operation to verify a simple implementation of Bell-LaPadula across Security Enhanced Linux devices. Unlike Concatenate, this operation is implemented. This operation is performed on two Hierarchical Security Policy Formal Models representing similar systems by creating a new node as a parent of the root nodes of the systems being Merged. This causes some issues with the uniqueness of policy paths, as policy paths are identified with an incrementing number during the construction of each individual policy graph. To resolve this, the current Merge operation re-identifies these policy paths, returning uniquely identified policy paths such that $PP_{Merge} \subseteq PP_1 \cup PP_2$. Application of this operation to an implementation of Bell-LaPadula across two Security Enhanced Linux systems, High and Low, allows a system security policy manager to identify the same user on both systems with different access, allowing them to read high and write low, violating Bell-LaPadula. This example relies on the system security policy manager's knowledge of the users of the system, and methods for further automation through incorporation of enterprise-level mappings such as this are described in Chapter 9.

Unlike both the Concatenate and Merge operations, the Composition operation described by this thesis operates below the abstraction level of full Hierarchical Security Policy Formal Model directed acyclic graphs and overlaid policy paths. Instead, the Composition operation provides a method by which disparate system security policy artifacts may be combined at the subpolicy level, effectively allowing creation of partial policy paths which may link various Subjects, Actions, and Objects without traversing from Start to End. The OpenStack system examined in this thesis presents a unique example in that each element set of the Hierarchical Security Policy Formal Model is contained within a separate system security policy file, and system security policy artifacts relating the element sets are identified from these files individually before being Composed into a full Hierarchical Security Policy Formal Model.

6.2 Formal Model Policy Composition

The Hierarchical Security Policy Formal Model Composition operation operates on sets or subsets of the elements of the Hierarchical Security Policy Formal Model; Subject, Actions and Objects. This operation takes two policy element sets and combines them, creating a larger subset of the complete Hierarchical Security Policy Formal Model. Effectively, this operation provides partial policy sets as a subset of the Cartesian product of two of the element sets or their subsets constrained by the system security policy, and modifies the directed acyclic graph of the model by combining the two operand sets to a single graph. If the resulting Hierarchical Security Policy Formal Model from the Composition operation still does not contain at least one of each of the three element sets, the hierarchical policy model will only

be able to create partial policy paths. These partial paths are used as placeholders for future full policy paths, and are not reflected in the final logical database interface presented to the system security policy manager.

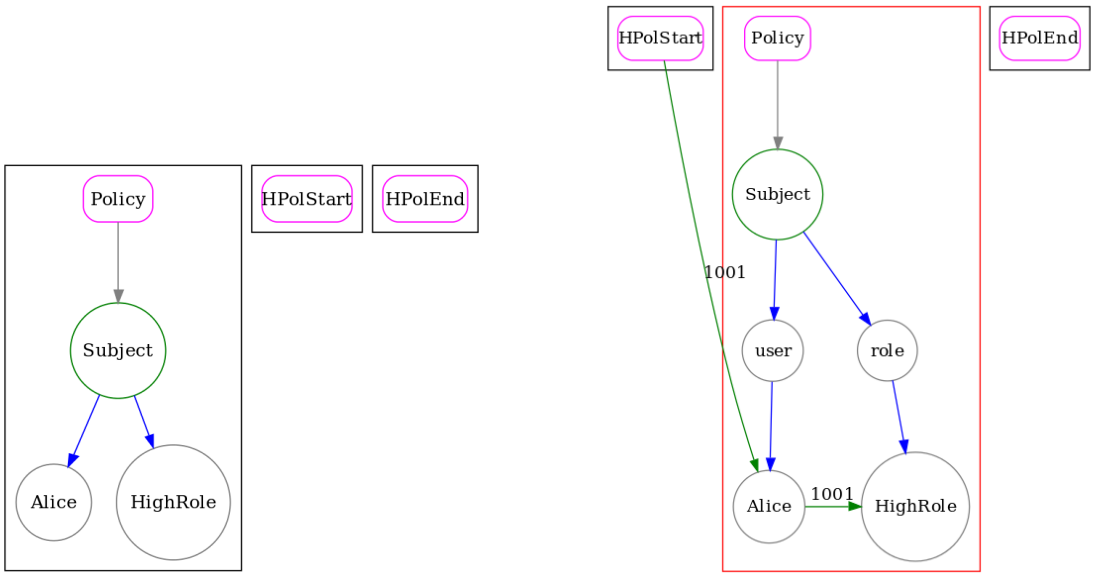
The Composition operation may be performed on subsets of the Hierarchical Security Policy Formal Model already containing multiple types of elements, though these applications of the Composition operation are simplified to the Composition of meaningful subsets, described later in this section. For example, if a Hierarchical Security Policy Formal Model already has identified system security policy artifacts for the Actions and Objects and has incorporated them into a single instance of the model, it still requires the set of system security policy artifacts representing the Subjects to become a complete Hierarchal Security Policy Formal Model. The Composition operation is used to combine the subsets $Compose(\mathbf{S}, \mathbf{AO})$, emulating $\mathbf{S} \times (\mathbf{A} \times \mathbf{O})$. Recall the policy path construct disregards the non-associativity of the Cartesian product due to its method of traversing the resulting set, effectively reducing this example to $\mathbf{S} \times \mathbf{A} \times \mathbf{O}$, generating an entire set of hierarchical policy model policy paths. As such, while the Cartesian product is a convenient, similar, existing operation, its use with regard to the Hierarchical Security Policy Formal Model only serves as shorthand for the Compose operation.

The Composition operation can be described generically for each possible combination of the three sets used by the Hierarchical Security Policy Formal Model. The meaningful possible combination Compositions are alluded to in the definition of the Hierarchical Security Policy Formal Model policy paths and, while other combinations are possible, they either do not represent meaningful additions to the Hierarchical Security Policy Formal Model in terms of policy paths, or require additional work to maintain the semantics specific to the system security policy. This thesis focuses on the meaningful combinations of element sets for Composition in Role-Based Access Control systems: Subject-Subject, Subject-Action, Action-Action, Action-Object, and Object-Object.

As the Composition operation is similar to the Cartesian product operation used in policy path creation, the list of meaningful Compositions may be derived from the definition of Hierarchical Security Policy Formal Model policy paths:

$$PP \subseteq \mathbf{S}^c \times \mathbf{A}^c \times \mathbf{O}^c$$

Notably the Composition of Subject-Object is not meaningful, as the definition of the Hierarchical Security Policy Formal Model policy paths does not include the Cartesian product between Subjects and Objects explicitly, $\mathbf{S} \times \mathbf{O}$. As such, the Composition operation is unable to derive additional policy paths from this pairing. These meaningful Compositions are further divided into three classes: 1) Single Composition, 2) Similar Composition, and 3) Differing Compositions. Single Compositions serve to insti-



(a) Subjects Before Composition

(b) Subjects After Composition

Figure 6.1: Single Formal Model Composition applied to a set of Subjects

tute partial orderings over unordered element sets, and prepare the element set for further Composition. Similar Compositions cover application of the Composition operation where the element sets being combined contain the same type of elements: Subject-Subject, Action-Action and Object-Object. Differing Compositions cover the rest of the meaningful Compositions, which Compose element sets containing different types of elements: Subject-Action and Action-Object.

6.2.1 Single Formal Model Compositions

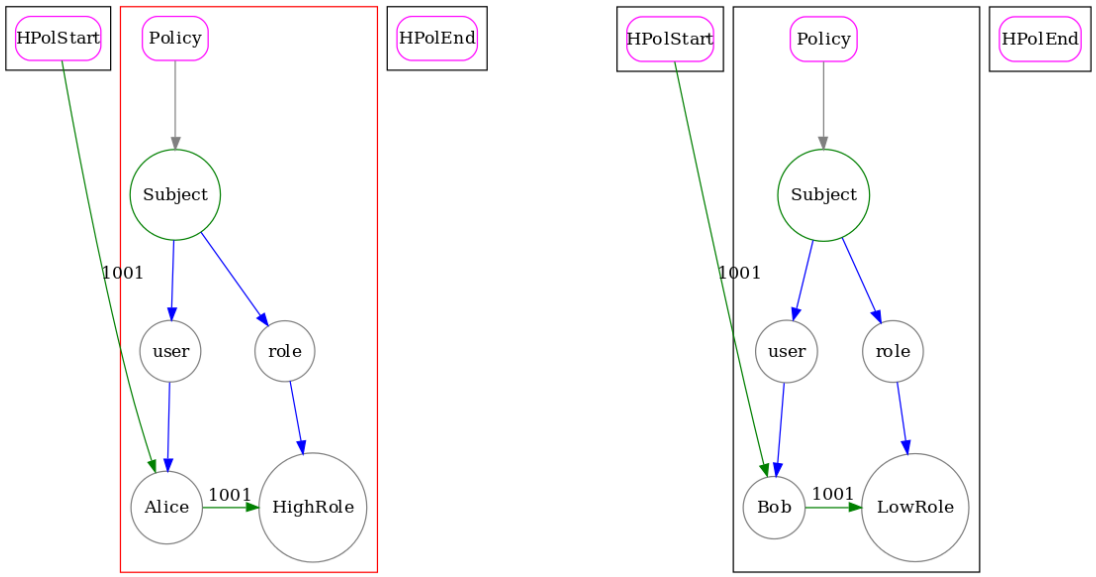
This unary Composition operation serves to begin the system security policy modeling process by providing a method by which to move from a Hierarchical Security Policy Formal Model with zero element sets to a Model with one element set. The use of this unary Composition operation has the benefit of preparing the directed acyclic graphs with the system security policy artifact origin information required to properly reconstruct disparate system security policy files. Creation of the directed acyclic graphs includes instituting the partial order for the target element set over the members of that element set. This process typically creates a graph node between the element set top-level node and the highest hierarchical nodes in the system security policy artifact set denoting the origin of the artifacts in the set.

Figure 6.1 displays the transformation of an element set of Subject policy artifacts undergoing Single Formal Model Composition. Initially, the policy artifacts *Alice* and *HighRole* are not separated into their respective subset and no connections have been made between them. After Single Formal Model

Composition, the policy artifacts have been placed in their subset, and the process has gathered enough information to connect the two.

6.2.2 Similar Formal Model Compositions

This application of the Composition operation seeks to combine two of the same element set type in the Hierarchical Security Policy Formal Model. Compositions of this class typically serve to extend the hierarchy of their element set type. In its application to Role-Based Access Control system security policy, the Hierarchical Security Policy Formal Model uses the Composition operation with similar element sets to combine the subsets of both the Subject and Action element types. Object types may be combined as well, though this application of the operation only serves to introduce new Objects to the model for Role-Based Access Control systems.



(a) Subjects Containing *Alice* and *HighRole*

(b) Subjects Containing *Bob* and *LowRole*

Figure 6.2: Similar Formal Model Subject Sets Before Composition

Recall the application of the Hierarchical Security Policy Formal Model to Role-Based Access Control declaring the subsets of Subject $\mathbf{S}_U$, $\mathbf{S}_R$, and $\mathbf{S}_S$, as well as the subsets of Action $\mathbf{A}_P$ and $\mathbf{A}\backslash\mathbf{A}_P$. Additionally, recall the Subject and Action hierarchy orderings for these partially ordered sets:

$$Sub_{\prec} = \mathbf{S}_U \prec \mathbf{S}_R \prec \mathbf{S}_S$$

$$Act_{\prec} = \mathbf{A}_P \prec \mathbf{A}\backslash\mathbf{A}_P$$

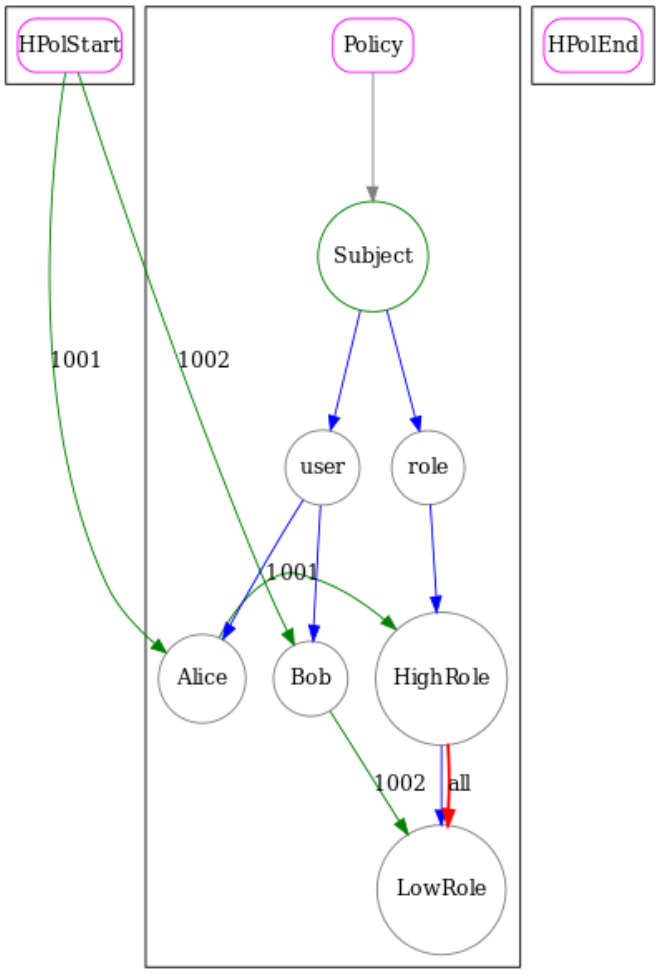


Figure 6.3: Similar Formal Model Composition of Two Sets of Subjects

With these definitions, similar model Compositions operating solely on Subjects may meet the requirements of the Subject element set in Role-Based Access Control systems if they contain all necessary data about Sessions in the system security policy. Similarly, Compositions of solely Action element types meet the requirements of the Action element set in Role-Based Access Control systems if they contain both Permissions and Actions protected by the Permissions.

Figures 6.2 and 6.3 display before and after of a Similar Formal Model Composition operation on two sets of Subjects. The first initial Subject set is the result of the Single Formal Model Composition displayed in Figure 6.2. The second is similar, though it contains the User *Bob* and the Role *LowRole*. When these two sets are Composed, the hierarchical orderings of each set is maintained, as well as new orderings present in the resulting subsets. *Alice* and *Bob* are placed equal in the hierarchy of Users, but

HighRole is place above *LowRole*. The partial policy paths present in the initial sets were both number 1001, causing a slight conflict. Recall that while the identifier of the policy path is not important, the maintenance of the unique policy paths is. As such, the partial policy path denoting *Bob*'s *LowRole* is changed to number 1002, while the actual path stays the same. Additionally, note that while the partial policy path node location may change the actual policy nodes in the paths do not. Recall the Hierarchical Security Policy Formal Model partial orders are such that the most restrictive set elements precede less restrictive elements. As such, note that partial policy path 1001 has been extended by the incorporation of the *HighRole* $\prec$ *LowRole* hierarchy. This hierarchy is a construct of this example, and, in an actual application, a similar hierarchy would require determination specific to the system being modeled.

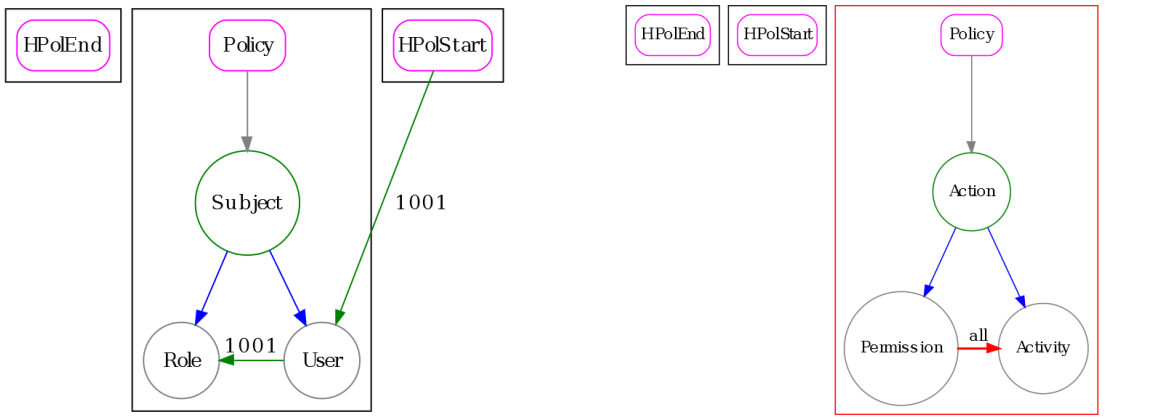
6.2.3 Differing Formal Model Compositions

This application of the Composition operation seeks to combine two different element set types in the Hierarchical Security Policy Formal Model. Compositions of differing element set types typically serve to extend the partial policy paths currently in the system security policy model. In its application to Role-Based Access Control system security policy, the Hierarchical Security Policy Formal Model uses differing Composition to extend policy paths from the Subjects to the Actions, and from the Actions to the Objects. Unlike Similar Model Composition described previously, differing model Composition does not seek to further define the hierarchy of the Hierarchical Security Policy Formal Model, as different element sets have their own set-specific hierarchical partial orderings and these orderings cannot stretch across element set types.

Recall the Hierarchical Security Policy Formal Model definition for policy paths as they traverse the directed acyclic graphs, $PP \subseteq \mathbf{S}^c \times \mathbf{A}^c \times \mathbf{O}^c$. The differing model Composition serves to provide the implementation of the Cartesian product between these disparate sets.

An instantiation of Hierarchical Security Policy Formal Model applied to a Role-Based Access Control system may use the Composition operation to combine disparate system security policy artifacts relating Subjects to Actions and Actions to Objects. With regard to Role-Based Access Control, these applications relate chains of Users, Roles and Session data to Permissions they satisfy and Actions authenticated by these Permissions, as well as the Permissions and Actions to the Objects the Actions may be performed upon.

Figures 6.4 and 6.5 display generic sets of Role-Based Access Control Subjects and Actions before and after Differing Formal Model Composition. The wildcard link connecting *Role* and *Permission* is provided by the example system's knowledge of the mapping of *Role* and *Permission*. The partial policy



(a) Subjects Containing a User and a Role (b) Actions Containing a Permission and an Activity

Figure 6.4: Differing Formal Model Sets Before Composition

path 1001 can be extended to traverse both the Subject and Action element sets after this addition.

An additional Differing Formal Model Composition using the result in Figure 6.5 and an element set of Objects may be performed to produce Figure 6.6. The Model displayed in this Figure has extended the previously existing partial policy paths to contain elements from each element set, as well as traverse the policy graph from the Start node to the End node.

6.3 Maintaining Policy Artifact Origins

The Composition operation seeks to operate on system security policy artifacts while maintaining the ability of the Hierarchical Security Policy Formal Model process to undo abstraction steps and return the original system security policy file to the system. This property is required to provide system security policy managers with the ability to manage policy from the Hierarchical Security Policy Formal Model. Through use of the Single Model Composition, the constituent element sets of a Hierarchical Security Policy Formal Model have been labeled such that the origins of their members are properly maintained. Both the Concatenate [3] and Merge [17] operations also support the maintenance of the origins of their operands, though neither explicitly state this.

The Concatenate operation [3] maintains the origin markers of both of the Hierarchical Security Policy Formal Models whose policy paths are Concatenated by only connecting the Start and End nodes of the policy paths. This operation does not affect the Hierarchical Security Policy Formal Model policy graphs for either of its operands, and cannot ruin any origin maintenance in the hierarchical policy model. The Merge operation [17] also maintains any existing origin markers of the Hierarchical Security Policy Formal Models it operates on. While the Merge operation does modify the Hierarchical Security Policy

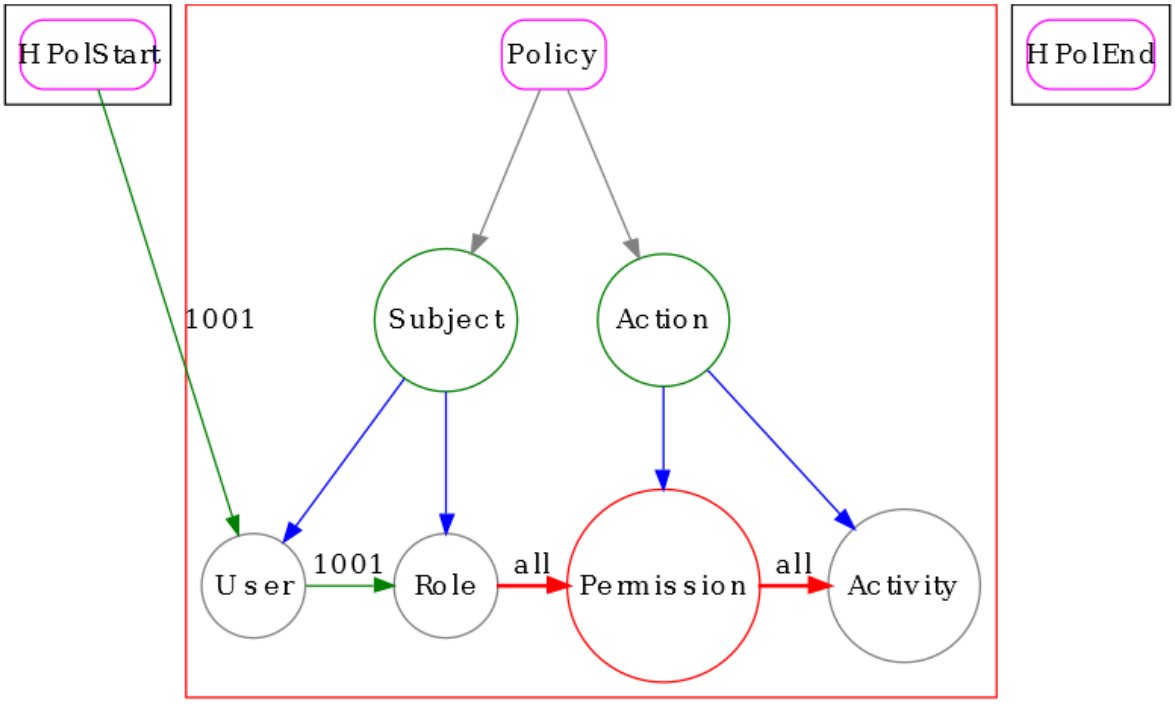


Figure 6.5: Differing Model Composition of Subjects and Actions

Formal Model of its two operands, the structure of both of the operands' directed acyclic graphs remains unchanged when they are combined under a single parent. The Merge operation does change the unique identifiers used by the policy paths of its operands, but this does not affect the structure of the policy paths nor the uniqueness of the policy paths were the Merge operation to be undone.

While the Composition operation does maintain the origin markers of system security policy artifacts, it may modify the directed acyclic graphs and overlaid policy paths of the system security policy. This may occur if the Composition operation is tasked to combine two of the same Hierarchical Security Policy Formal Model element sets, at which point the Composition operation, similar to the Merge operation, creates separate subpolicy directed acyclic graphs representing its operands. This creates an extra graph node between the element set top-level node its current children denoting the system security policy file the system security policy artifacts in the set originate from. Note that if every system security policy element set in the Hierarchical Security Policy Formal Model began as a single model Composition the graph nodes denoting the system security policy file origins should already be in place, and the directed acyclic graphs should not require modification to properly track artifact origins during Composition.

As creation of hierarchical policy model policy paths simply requires evaluating the current directed acyclic graph chains with the proper system authenticator, the policy paths may be removed for Compo-

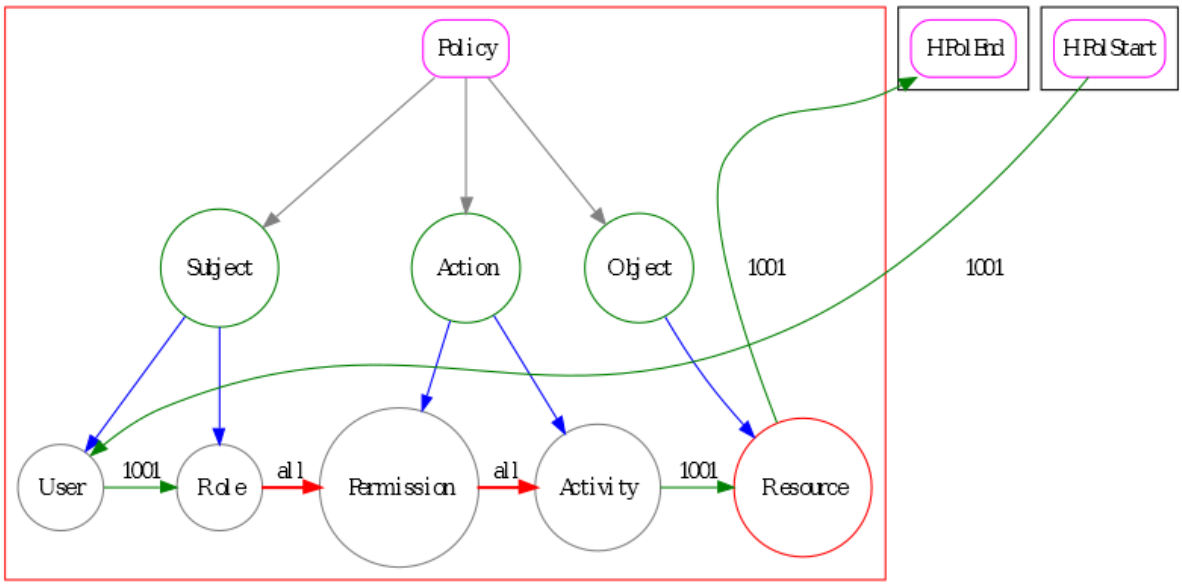


Figure 6.6: A Fully Composed Role-Based Access Control System Formal Model

sition and recreated after the operation or sequence of operations is complete. If the Hierarchical Security Policy Formal Model was created using Single Model Composition, the paths to the nodes in the directed acyclic graphs should not change during the Composition, and the previously existing hierarchical policy model policy paths will still be valid policy paths for the model. This also maintains the uniqueness of each policy path, though, unless the specific Hierarchical Security Policy Formal Model instantiation keeps track of the unique policy path identifiers outside the model, the unique identifiers of the policy paths may change.

Chapter 7: Policy File Reconstruction

One of the goals of the Hierarchical Security Policy Formal Model is to provide system security policy managers with a tool from which they may manage system security policies from disparate systems. Currently, the Hierarchical Security Policy Formal Model process has no declared methods by which it may dictate system security policy. While system security policy may be verified at the abstraction level of the logical database and the directed acyclic graphs, system security policy modifications at these levels have no way of being reflected down to actual system security policy files. Similar work has been done tangentially related to the Hierarchical Security Policy Formal Model with High-Fidelity Browser Policy Management tools, but these tools currently lack the functionality required to transfer system security policy knowledge from the logical database to specific configuration files [5]. The work on High-Fidelity Browser Policy management approaches the issue from a top-down perspective, operating primarily at a high abstraction level and allowing system security policy managers to speak about policy verification in terms of facts in the policy specification language HERMES [6].

The Hierarchical Security Policy Formal Model instead approaches the problem from the lowest level, focusing on abstracting system security policies to a more human-usable form. When the Model process begins at the lowest level of abstraction it is convenient to tag the system security policy artifacts with their origin system security policy files as they are extracted and identified, as described in Chapter 5 of this thesis. The application of these various origin tracking mechanisms occurs throughout the Hierarchical Security Policy Formal Model process in its application to Role-Based Access Control systems, and includes markings such as artifact specific tags or the artifact's position in the directed acyclic graph. Information closely coupled with the system security policy artifacts are included alongside the origin information, as the closely coupled information is often required to properly reconstruct system security policy files.

Once a Hierarchical Security Policy Formal Model has been constructed out of the directed acyclic graphs, a system security policy manager may modify the model in ways compatible with the system, and push these changes down to the system security policy file. Compatibility is measured by the similarity to system security policy artifacts originating in the target system security policy file; assuring the modifications or additions made by the system security policy manager are within the bounds of the system allows the modification to be pushed to the relevant system security policy file. As the system security policy manager seeks to modify the Hierarchical Security Policy Formal Model through any interface, they should be required to provide enough information to properly situate the modified system security policy artifact alongside compatible artifacts originating from the same target system security policy file as the modified artifact. While the Hierarchical Security Policy Formal Model serves

to facilitate the verification and management of system security policy files, it is still a tool which system security policy managers must use correctly to ensure valuable results.

Currently, there exists no interface for modification or addition of system security policy artifacts outside of directly modifying the directed acyclic graphs or overlaid policy paths. This potential addition is designated as future work, and discussed in further detail in Chapter 9.

7.1 Identifying Policy Artifact Origins

As the Hierarchical Security Policy Formal Model process performs various operations in order to abstract the system security policy files, origin markers are placed and maintained on extracted system security policy artifacts. This is difficult to describe in a generic fashion, as every system security policy file contains information about different parts of the Hierarchical Security Policy Formal Model, and generalization of this information is one of the key points the Model seeks to address. However, once system security artifacts have been identified for incorporation into the model, they may be individually marked with their system security policy file of origin. These origin markings may come in the form of a key-value pair added to the dictionary of node information for the artifact's node its directed acyclic graph, or the origin marking may determine the path to the node in its graph. Additional origin information may be implicitly stored in the hierarchies used to process policy paths across the directed acyclic graph, but, as with the various authenticators used to process these paths during construction, recovering origin information stored in this way requires an implementation specific to the target system security policy file.

The Hierarchical Security Policy Formal Model operations leave key-value origin markers unchanged; none of the currently available operations, Concatenate, Merge, or Compose operate on individual system security policy artifacts. The Hierarchical Security Policy Formal Model operations Merge and Compose may affect the path from the root graph node to system security policy artifact node in the directed acyclic graph, but should maintain the structure of the graphs below the Hierarchical Security Policy Formal Model element sets. For the purposes of reconstructing system security policy files, the origin information stored in the structure of the directed acyclic graph is contained below the nodes of the graph representing the Hierarchical Security Policy Formal Model element sets or their immediate subsets. Hierarchical Security Policy Formal Model operations may modify the policy paths of the system, though all operations should provide at least the same set of policy paths as they began with, which guarantees at least maintenance of the origin markers contained in the policy paths.

Actual identification of policy artifact origin markers must be done in a way specific to the system

security policy file origin of the artifact, similar to how we identified and extracted the system security policy artifacts during the construction of the Hierarchical Security Policy Formal Model. Recall Chapter 5 of this thesis; every system security policy file may be different, but once they have been extracted from the target system and abstracted into their system security policy artifacts, they may be operated upon by the Hierarchical Security Policy Formal Model process in a generic fashion.

7.2 Policy File Reconstruction

Once the system security policy file origin markers of a system security policy artifact have been identified, the artifact is then prepared to be returned to its original policy file. Recall that, as the Hierarchical Security Policy Formal Model process is intended to be performed on a separate system from the target system, the system security policy files from the target system were extracted in their entirety and placed on the system performing the Model process. Thus, the system security policy files of the target system should exist on the system performing the Hierarchical Security Policy Formal Model process, and the process for reconstructing policy files should have access to them. This allows the reconstruction process to load the previous system security policy file and reincorporate the potentially modified system security policy artifacts into the original file. In some cases, it is simpler to reconstruct the target system security policy file from scratch, though policy files should still be maintained by the system performing the Model process for the duration of the process.

For example, the reconstruction of the Keystone identity database in the OpenStack system requires the previously extracted Keystone identity database file in order to provide proper reconstruction, as the identity database contains more information than is used in the application of the Hierarchical Security Policy Formal Model process. Where the Keystone identity database system security policy file requires the majority of the file to be provided, the Cinder block storage service permission policy configuration file does not. This is mainly thanks to the JSON formatting used by the Cinder policy file, though it is also concise enough that the entirety of the file can be expressed by the Hierarchical Security Policy Formal Model. Both of these examples are further expanded upon in the context of the application of the Hierarchical Security Policy Formal Model to the OpenStack platform in Chapter 8.

7.3 Policy File Redeployment

After the system security policy artifacts have been recovered and reintegrated into their original system security policy files, the policy files are ready to be returned to their target systems. As operations such as Merge and Compose are available to the Hierarchical Security Policy Formal Model, the proper

maintenance of the origins of each policy file becomes paramount. If multiple system security policy files describing similar systems are incorporated into a single Hierarchical Security Policy Formal Model the files must be returned to their original target system to have the correct effect on the system. Much like the extraction of system security policy files from the first steps of the application of the Hierarchical Security Policy Formal Model process, described in Chapter 5, the redeployment of system security policy files is specific to the target system. Continuing the examples of the previous section, the Keystone identity database requires a method by which the database may be populated, inverting the process used to obtain the system security policy file. In the case of the Cinder policy configuration file, the file is only required to be dropped into the correct directory of the system, potentially overwriting the existing policy file.

This process could be automated to facilitate quick turnaround for system security policy managers modifications, but, in reality, the redeployment process should likely contain some safeguards to prevent incompatible files being pushed to a production environment. Of course, if the new system security policy file originated from a Hierarchical Security Policy Formal Model of the target system which implements a proper modification operation and a diligent system security policy manager redeployment should be problem-free.

Chapter 8: Case Study: OpenStack

The Hierarchical Security Policy Formal Model Project chose to use the OpenStack cloud platform [13] as the system from which to model Role-Based Access Control systems. As such, this thesis examines an example OpenStack deployment consisting of two services: the Keystone identity Service, and the Cinder block storage service. The application of the Hierarchical Security Policy Formal Model to this example OpenStack environment cares about three primary system security policy files: the Keystone database, the Cinder policy configuration file, and the Cinder database. The application of the model makes heavy use of the Composition operation as the three target system security policy files are spread across the entire example OpenStack deployment.

An enumerated summary of the Hierarchical Security Policy Formal Model process application to OpenStack follows:

1. Extract the three system security policy files from their respective service.
2. For each policy file, identify the system security policy artifacts appropriate to that policy file.
3. For each policy artifact, identify its closely coupled data and associate the artifact with the data.
4. For each policy artifact, identify its appropriate Hierarchical Security Policy Formal Model element set.
5. For each element set, Subject, Action, and Object, perform Single Model Composition on the set, preparing it for future use.
6. Perform Differing Model Composition with the Subject and Action element sets.
7. Perform Differing Model Composition with the result of the previous Composition and Object element sets.

As OpenStack is a Role-Based Access Control system, the Single Model Composition of each of the element sets will institute the appropriate partial ordering over the set, as well as tagging the individual policy artifacts with their origin policy file. The application above could be further broken down, incorporating Single Model Compositions for each subset of Subjects and Actions, though this is unnecessary, as the Subject and Action partial orderings already account for these subsets. Additionally, the authenticator oracle supplied by the OpenStack platform must be able to determine Subject access to the Actions and Objects. To do so, the authenticator requires three parameters: 1) A credential dictionary detailing the Session, 2) a rule to be tested against, and 3) an object to test access to. As the Keystone database, the Cinder policy configuration file, and Cinder database were all extracted, the authenticator may be

supplied with whatever information is required to pass judgment. Thus, after each Differing Model Composition, the Hierarchical Security Policy Formal Model process will be able to further evaluate partial policy paths, eventually creating complete paths after the final Composition.

Additionally, the potentially modified OpenStack system security policy files may be recovered from the Hierarchical Security Policy Formal Model of the system. Thanks to the origin policy file data used to tag the system security policy artifacts during the Single Model Composition, each policy artifact contains information as to were it belongs in the system. The process of policy file reconstruction, reintegration, and redeployment is enumerated below:

1. For each hierarchical policy model element set, identify each individual policy artifact.
2. For each policy artifact, query its tags to procure its original system security policy file.
3. Reconstruct each policy artifact by extracting its closely coupled data and organize it such that it is consistent with its origin policy file.
4. Reintegrate each policy artifact by inserting it and its closely couple data into its original policy file.
5. Redeploy the system security policy files by returning them to the system from which they originated.

Note that, much like extracting the system security policy artifacts, each step detailed above requires some knowledge of the specific system policy file being generated. This knowledge was gathered by the Hierarchical Security Policy Formal Model during application to OpenStack, thus allowing each step to occur.

8.1 Role-Based Access Control Modeling with OpenStack

The first case study deals with application of the Hierarchical Security Policy Formal Model process to the OpenStack system. This step begins by identifying the system security policy files which concern the Subjects, Actions, and Objects of the system. As this application of the Hierarchical Security Policy Formal Model examines a Role-Based Access Control system, recall the sets of entities used in such systems: Users, Roles, Sessions, Permissions, Actions and Objects. The chosen system for this application of the policy model is OpenStack, which uses an identity service Keystone to identify users on its domain, policy files on each service to link various Permissions to Actions, and a database on each service to list the attributes of Objects managed by the service. For the purposes of this example deployment, the

Cinder block storage service was used in conjunction with the Keystone identity service to construct the Hierarchical Security Policy Formal Model for an OpenStack deployment.

The OpenStack Keystone identity service controls a relational database which contains, among other things, individual Users, Roles and Projects used to create user Sessions in the OpenStack environment. The relationship between Users, Roles and Projects is contained in the Assignment database table, which has columns for user identifiers, role identifiers and project identifiers. Each entry in this table denotes a single User Role pairing for the specified Project. The User, Role, and Project elements in this table are referenced by a unique identifier with the actual data for the element stored in a separate table for that class of element. Each of these element class tables stores a mapping of the unique identifier to various element data such as display name and description. A database parser is used to dump the contents of the database into a file in the form of nested dictionaries. This file is then extracted from the database service in its entirety and placed on the system performing the Hierarchical Security Policy Formal Model process. The approach of dumping the database to a single file may not scale to a larger OpenStack distribution, and an actual database dump may be extracted and imported instead. The database provides the Subject set required by the Hierarchical Security Policy Formal Model, and all the Role-Based Access Control information required to enumerate every static Session available on the system.

All OpenStack services, in this case the Cinder block storage service, contain a JSON formatted policy file denoting the Role-Based Access Control policy determining who of the OpenStack system users may access which OpenStack service Object through what Action. OpenStack manages access to service Objects through Permissions-based rules for various API functions so, while this file does not relate the OpenStack service Objects directly, it does provide access to the Permissions and Actions associated with them [12]. The elements of these policies file are key-value pairs, with the key either representing a Permission alias or a target API function call. The values of both types of keys are ultimately Permission definitions, though layers of indirection may be used to delegate Permission accesses to other Permissions. By forming a hierarchy of Permission aliases with their actual evaluated permissive tests, and a hierarchy of Actions linked to these Permissions, the policy file of the target service provides the Action set required by the Hierarchical Security Policy Formal Model, as well as the Role-Based Access Control information required to test Sessions against the OpenStack policy enforcer authenticator. The policy file is extracted in its entirety and placed on the system performing the Hierarchical Security Policy Formal Model process.

The final set required by the Hierarchical Security Policy Formal Model are the Objects of the system. Each service provides various different sets of Objects depending on the service provided. As this instance of the Hierarchical Security Policy Formal Model process targets the Cinder block storage service, the

Objects of the system are volumes representing data storage blocks in the OpenStack infrastructure. The Cinder service controls a relational database which contains information about every existing block storage volume managed by the Cinder service. The database entries regarding the volumes managed by Cinder represent the volume primarily by a unique identifier, and contain information about the volume such as its status, display name, size, type, if the volume is bootable, and the current attachment status of the volume. In an OpenStack deployment these volumes are attached to the Nova compute service, but this attachment is abstracted in a such a way that Nova service relies on the Cinder service to provide attachment information; for the purposes of this thesis, Nova is not necessary to derive meaningful benefit from the target OpenStack deployment. Like the database from the Keystone identity service, the Cinder service database is dumped to a nested dictionary and extracted from the system to the machine performing the Hierarchical Security Policy Formal Model process.

At this point, our application of the Hierarchical Security Policy Formal Model process to OpenStack has extracted a database representing the Subjects of the system from the Keystone identity service, a JSON policy file representing the Actions from the Cinder block storage service, and another database file representing the Objects from the same Cinder block storage service. Note that these various system security policy files were extracted in their entirety, and are expected to be returned similarly after any modification by higher abstraction levels of the Hierarchical Security Policy Formal Model process. This expectation allows the higher abstraction levels of the Hierarchical Security Policy Formal Model process to modify the system security policy artifacts and return them to the target system in fully usable system security policy files.

This is accomplished by associating the closely coupled data from the database system security policy files into dictionaries held by their respective directed acyclic graph nodes. This action is performed for each of the Hierarchical Security Policy Formal Model partially ordered element sets: $\mathbf{S}$, the Subjects, $\mathbf{A}$, the Actions, and $\mathbf{O}$, the Objects. In our application of the Hierarchical Security Policy Formal Model to Role-Based Access Control we further refined these sets into various partially ordered subsets: $\mathbf{S}_U$, $\mathbf{S}_R$, and $\mathbf{S}_S$ dividing the Subjects set into Users, Roles, and extraneous Session information, and $\mathbf{A}_P$, the set of Permissions in the system. To form the policy paths of the system, the Hierarchical Security Policy Formal Model process must identify the sets $\mathbf{S}_U$, $\mathbf{S}_R$, $\mathbf{S}_S$, $\mathbf{A}_P$, $\mathbf{A} \backslash \mathbf{A}_P$ and $\mathbf{O}$ from the system security policy files.

We begin the system security policy artifact identification process with the set of Role-Based Access Control system Users, $\mathbf{S}_U$. In OpenStack, these system security policy artifacts are found in the database file extracted from the Keystone identity service. The artifacts identified are primarily the names of the Users on the OpenStack deployment, as well as the unique identifier for each User. Additionally, the User

system security policy artifacts have closely coupled data in the form of various User data such as email addresses and hashed passwords. The closely coupled data is extracted from the Keystone database file and associated with the name of the User it belongs to.

Similarly to the identification of the set $\mathbf{S}_U$, the sets $\mathbf{S}_R$ and $\mathbf{S}_S$ have their artifacts identified from the Keystone identity service database. The primary artifacts of the $\mathbf{S}_R$ set are the names of the Roles in the system, with closely coupled data in the form of Role descriptions. Recall the purpose of the set $\mathbf{S}_S$ is to encompass Role-Based Access Control Session data which is not Users or Roles. In the context of OpenStack, this extraneous Session data comes in the form of Projects with which the User's Roles are associated. Thus, the primary artifacts of the $\mathbf{S}_S$ set are the names of the Projects in the system, with closely coupled data in the form of Project descriptions.

In order to provide hierarchy to these partially ordered subsets, the Hierarchical Security Policy Formal Model process must define a Subject Hierarchy reflecting the semantics of OpenStack's Role-Based Access Control model. The partial order $Sub_{\prec}$ is unable to compare any two elements of any of the subsets of $\mathbf{S}$, though still provides the generic ordering:

$$Sub_{\prec} = \mathbf{S}_U \prec \mathbf{S}_R \prec \mathbf{S}_S$$

The semantics of this partial order in the Hierarchical Security Policy Formal Model relate a User having a Role and forming a Session from that Role and other extraneous Session data. From this extracted Session data, partial policy paths may be created traversing the Subjects of the Hierarchical Security Policy Formal Model.

Figure 8.1 details the Subjects of the OpenStack example system. In our example, the system has Users *Alice* and *Bob*, the Roles *_member_* and *admin*, and the extraneous Session data Projects *admin* and *bob_project*. In this example system, *Alice* has two Roles, *admin* and *_member_*. Recall that, for OpenStack, Roles are reused between Projects, and while *Alice* has the *admin* Role in the *admin* Project, *Alice* only has the *_member_* Role in the *bob_project* Project. Similarly, *Bob* has the *_member_* Role, but only in the *bob_project* Project. The partial policy paths 1001, 1002, and 1003 denoted these accesses respectively.

The set of Action system security policy artifacts derived from the OpenStack Cinder policy JSON file do exhibit some hierarchy beyond the default partial order. Recall the rule language used to establish Permissions in the OpenStack platform system security policy file for the individual platform services. The elements of these policies file are key-value pairs, with the key either representing a Permission alias or a target API function call. The values of both types of keys are ultimately Permission definitions, though layers of indirection may be used to delegate Permission accesses to other Permissions or combinations

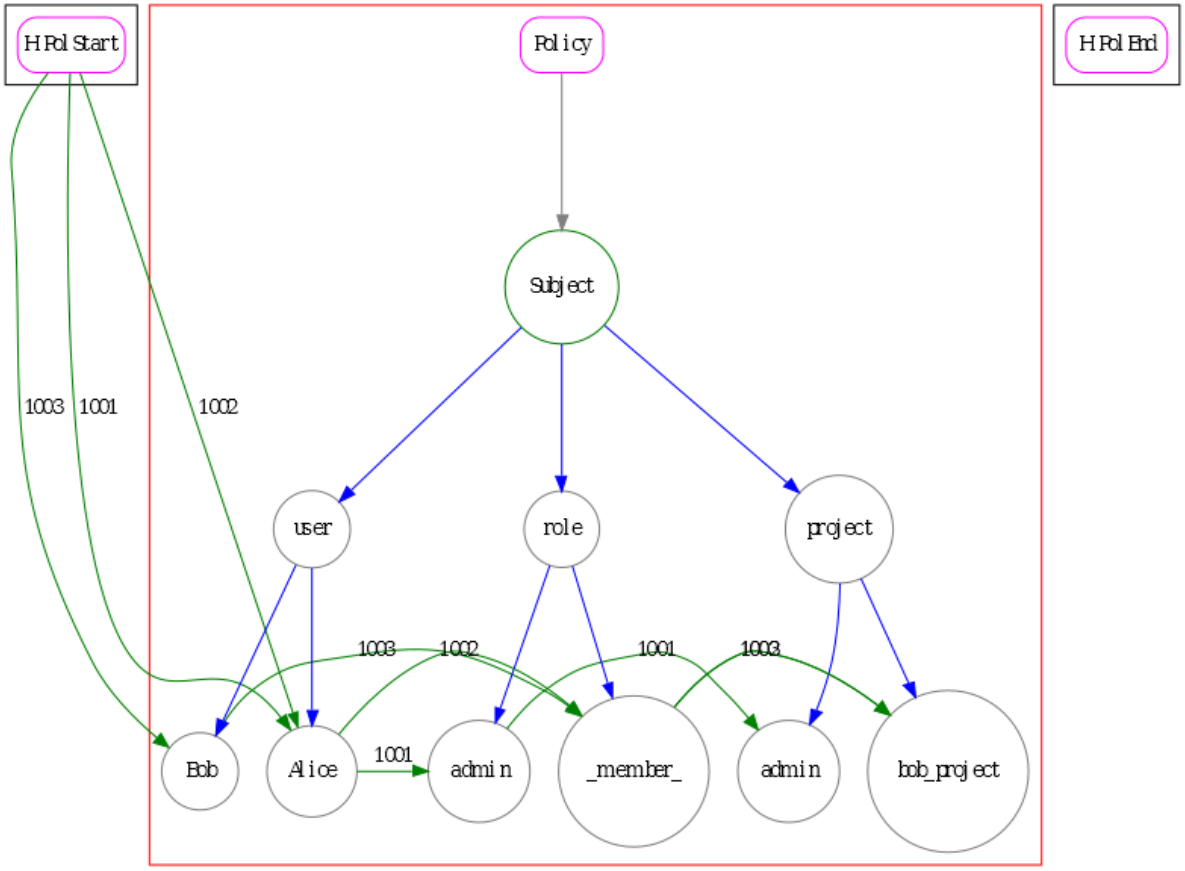


Figure 8.1: OpenStack Subjects

of Permissions. Permission definitions can be [12]:

1. always true, written as the empty string, [ ] or "@",
2. always false, written as "!",
3. a special check, described below,
4. a comparison of two values from the second list below, or
5. a boolean expression of other Permission definitions,

Special checks as mentioned above are [12]:

1. $<$ *role* $>$:$<$ *rolename* $>$, testing if API credentials contain this role,
2. $<$ *rule* $>$:$<$ *rulename* $>$, testing an aliased rule, or

3. $http://<targetURL>$, delegating the boolean check to a remote server,

Possible values to compare as mentioned above are [12]:

1. constants: Strings, numbers, True and False,
2. API attributes,
3. target object attributes, or
4. the flag *is_admin*

Allowed API attributes are *project_id*, *user_id* and *domain_id*. Target object attributes are held in the service's database a fields of the Object being targeted by the access. This can be used to check if the Object being accessed belongs to the same project as the User Session attempting the access. The *is_admin* flag tests if the Session has been granted the admin token, which grants administrative privileges similar to the admin role and allows initialization of the Keystone identity database before any roles exist in the OpenStack environment [12].

Given these rule language constraints, developing a hierarchy relating rule indirections for $\mathbf{A}_P$ is possible, though the default mapping remains in place for the rest of the Actions set $\mathbf{A} \backslash \mathbf{A}_P$. The Action Hierarchy for $\mathbf{A}_P$ in OpenStack is:

1. Permission language rules rank above aliased rules,
2. Aliased rules aliasing permission language rank above aliased rules aliasing other rules,
3. Aliased rules aliasing aliased rules rank above the rules they alias,
4. Boolean expression rules relating other rules rank depending on the relative rank of the rules they relate and the operation performed

This allows the Permissions of $\mathbf{A}_P$ to be ordered by proximity to and permissiveness of their permission language ancestor. This new rule hierarchy partial ordering is added to the default partial ordering for Actions, enabling more refined hierarchies to become apparent in $\mathbf{A}$ for OpenStack. This process establishes the partial policy paths across the Actions, using wildcard links to connect Permissions to their non-Permission Action. These policy path links are defined in the Cinder service policy JSON file and by the refined rule hierarchy partial ordering relating rule indirections. Hierarchical Security Policy Formal Model wildcard links are used for these sections of policy path, maintaining the Role-Based Access Control allowance that any Session which satisfies a Permission, possibly through a Permission Hierarchy,

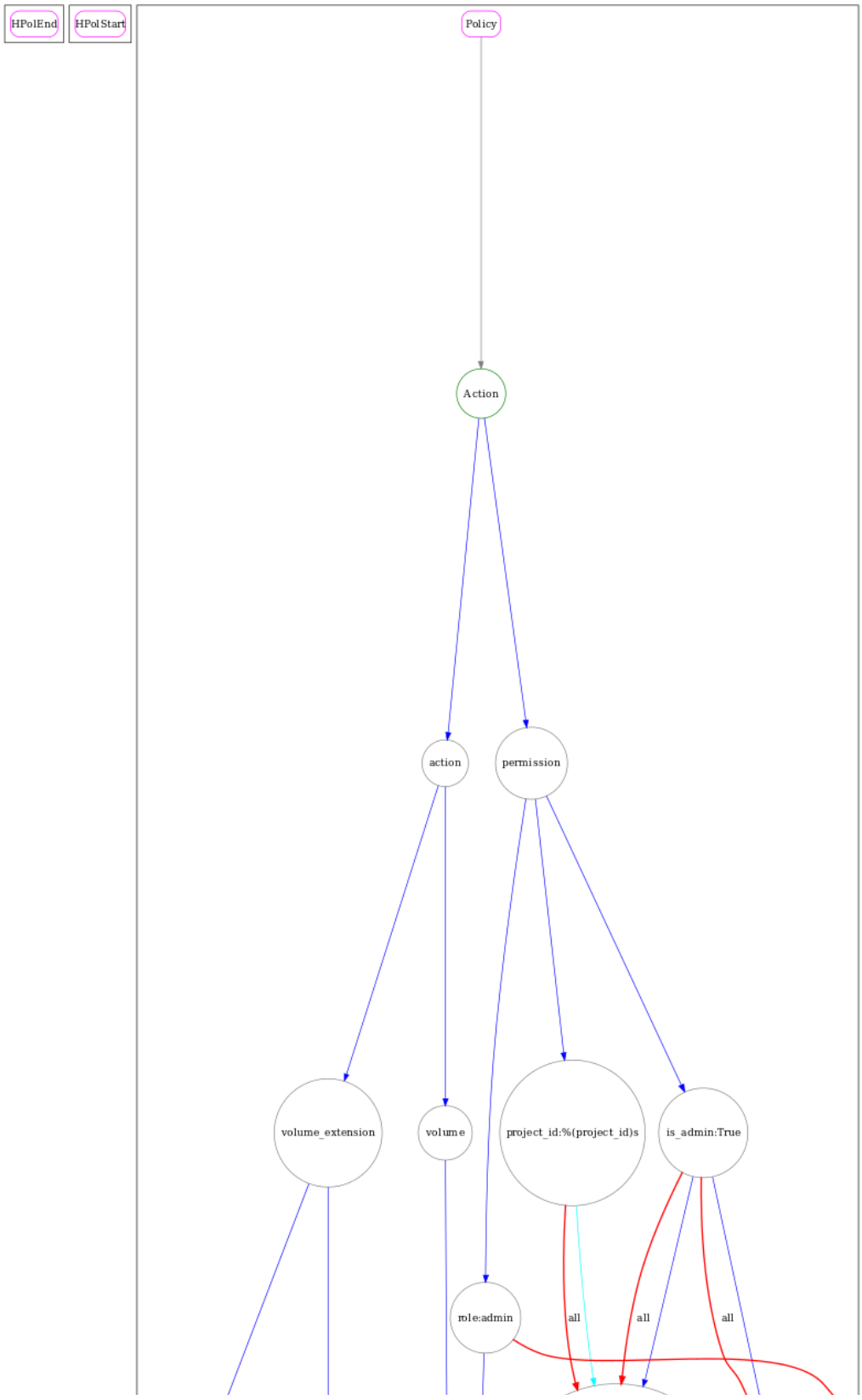


Figure 8.2: OpenStack Actions

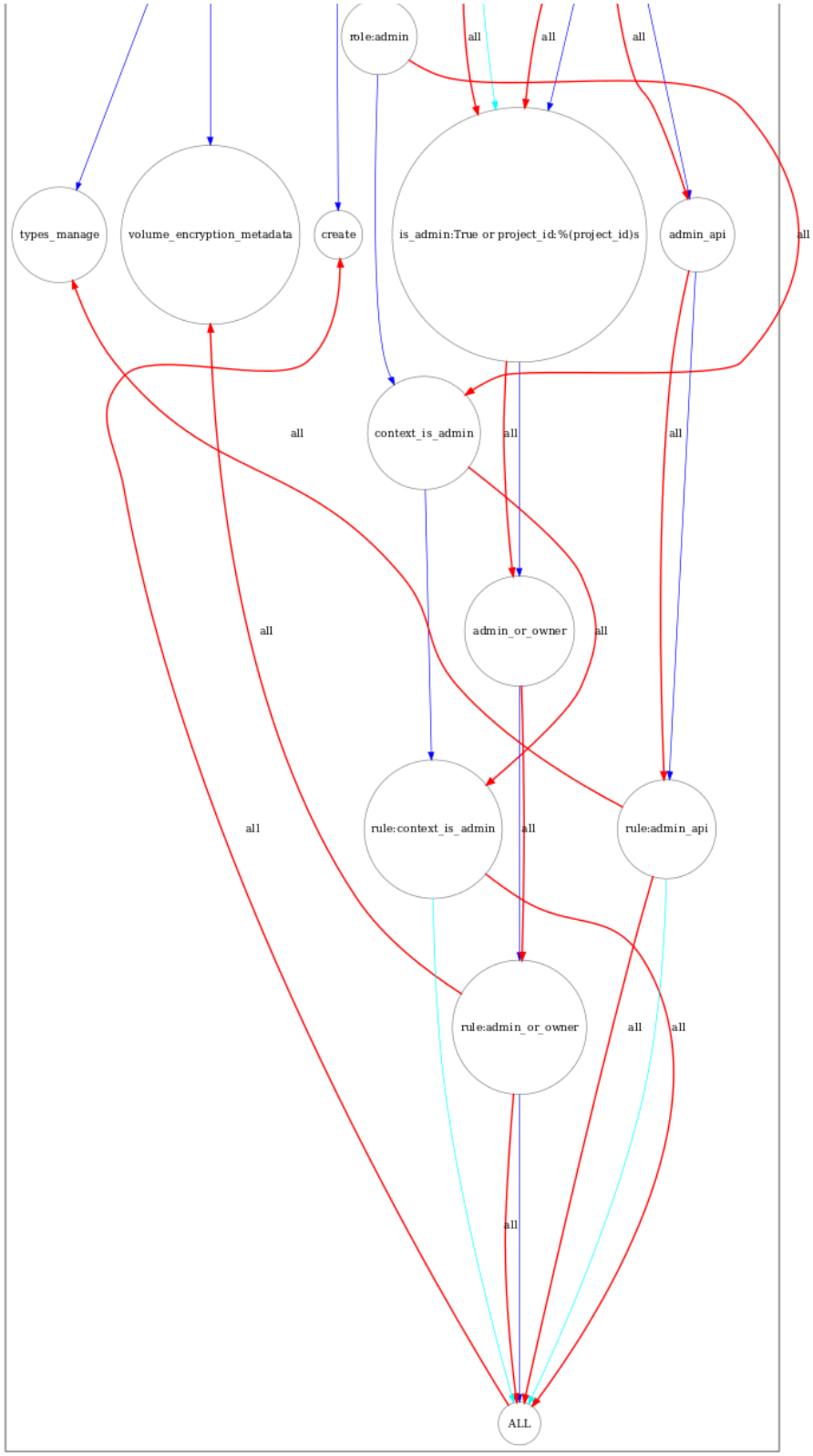


Figure 8.3: OpenStack Actions Extended

has the ability to perform the permitted Action on the system Object. Effectively, this creates the links the elements of $\mathbf{A}_P$ and $\mathbf{A} \backslash \mathbf{A}_P$.

Figures 8.2 and 8.3 display the Hierarchical Security Policy Formal Model for the Action element set. In this example OpenStack deployment, there are only six rules in the Cinder policy configuration file:

1. "context_is_admin": "role:admin",
2. "admin_or_owner": "is_admin:True or project_id:%(project_id)s",
3. "admin_api": "is_admin:True",
4. "volume:create": "",
5. "volume_extension:type_manage": "rule:admin_api",
6. "volume_extension:volume_encryption_metadata": "rule:admin_or_owner"

Note that the fourth rule has no rule language test associated with it. In OpenStack, the lack of a rule language test signifies the ability of any User to perform the action. Additionally, as the current modeling software is unable to display nodes with no title, this minimum Permission is displayed by the node *ALL*.

The Permissions displayed in Figures 8.2 and 8.3 are split into three rule language ancestors. Each of these ancestors have children representing their rule alias, as well as grandchildren representing the reference to these aliased rules. The Permissions directed acyclic graph concludes with the minimum Permission, *ALL*. The non-Permission Actions of this example configuration file are split between the ancestor nodes *volume* and *volume_extension*. This apparent hierarchy is used to minimize the horizontal size of the display, though it does not actually denote anything of value to the Hierarchical Security Policy Formal Model. If the directed acyclic graphs were constructed without this hierarchy, all non-Permission Actions would be at the same level.

The Hierarchical Security Policy Formal Model Object set for OpenStack replaces the default partial ordering for $\mathbf{O}$, $Obj_{\prec}$, with an ordering denoting the attachment of each Object to its Nova compute service instance. In practice, since Cinder policy does not grant access to Nova, this distinction only serves to facilitate management of Cinder volumes by providing paths for each known Nova service to the system security policy manager.

Figure 8.4 displays the Hierarchical Security Policy Formal Model Objects present in the example OpenStack deployment. The Cinder service primarily deals with block storage volumes, here represented by the nodes *volume_1*, *volume_2*, and *volume_3*. These volumes have been placed under the Project

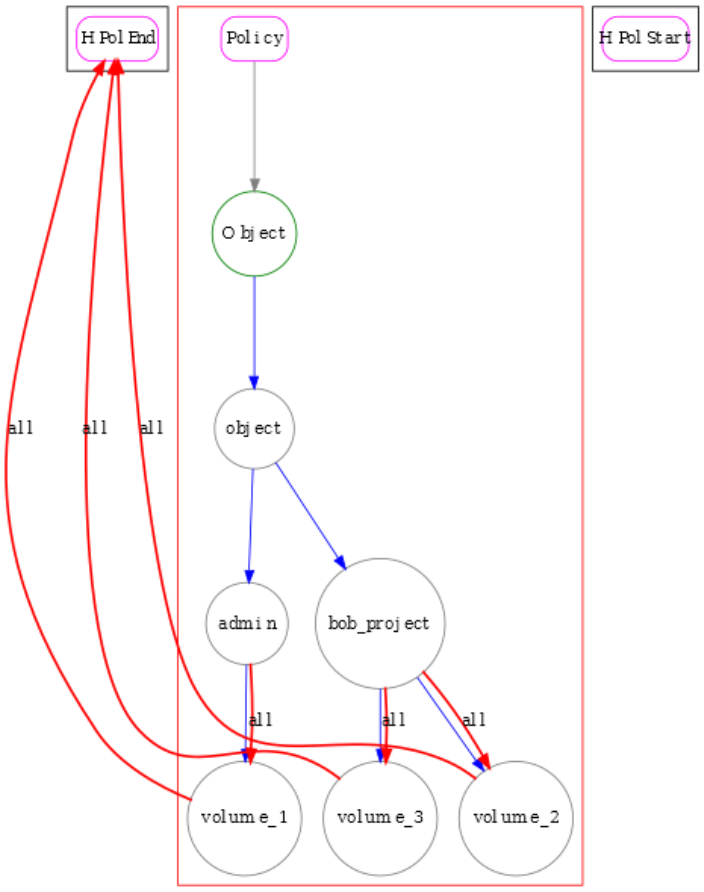


Figure 8.4: OpenStack Objects

which currently owns them. The partial policy paths present in the Object directed acyclic graph connect the Projects to the End node through their owned volumes. Thus, when the Object Formal Model is Composed with another Model, partial policy paths connecting to the Project nodes will be extended to the End node.

Once the three partial orders have been declared and refined for the application to OpenStack, the sets of the Hierarchical Security Policy Formal Model are organized into directed acyclic graphs representing these orderings. The directed acyclic graphs provide the structure used by the Hierarchical Security Policy Formal Model to evaluate potential complete policy paths as the element sets are composed, the next step of the model.

Recall the refined definition for Hierarchical Security Policy Formal Model Policy Paths when related to Role-Based Access Control:

$$PP \subseteq \mathbf{S}_U^c \times \mathbf{S}_R^c \times \mathbf{S}_S^c \times \mathbf{A}_P^c \times (\mathbf{A} \backslash \mathbf{A}_P)^c \times \mathbf{O}^c$$

The policy path creation step begins by adding the *Start* and *End* nodes to the Hierarchical Security Policy Formal Model graph, unconnected to any other node. The Hierarchical Security Policy Formal Model then seeks to evaluate every Session against each Permission, resulting in policy path evaluations for $\mathbf{S}^c \times \mathbf{A}_P^c$. These evaluations use the authentication oracle provided by OpenStack, the Policy Enforcer. The Policy Enforcer is initialized with the Cinder service policy JSON file, and is able to evaluate the rule indirections reflected in the partial order denoted by $Act_{\prec}$ refined earlier. Using the Policy Enforcer, the Hierarchical Security Policy Formal Model processes every element of the set $\mathbf{S}^c \times \mathbf{A}_P^c$. Each successful authentication extends its respective Session's partial policy path to connect with the Permission successfully evaluated. At this point, partial policy paths exist beginning at the Start node, traversing the Subjects, connecting to a Permission, and traversing the Actions. This constitutes the first Differing Model Composition for OpenStack, linking the Subject directed acyclic graph to the Action directed acyclic graph.

The final Differing Model Composition combines the previously constructed partial policy paths through **S** and **A** with the partial policy paths present in **O**. To do this, the partial policy paths are extended by use of the authentication oracle to connect the non-Permission Action with the appropriate Project policy artifact in **O**. This completes the partial policy paths, as they now begin at the Start node, traverse **S**, connect to and traverse **A**, connect to and traverse **O**, and connect to the End node.

To complete the Hierarchical Security Policy Formal Model policy paths linking the Actions to the Objects, the unique policy path identifier is used to form links from the Action to the OpenStack Project subset of the **O** directed acyclic graph. Wildcard links are then used to link the Project with its children, the Cinder block storage Objects which are attached to the Project. The policy path generation completes by using wildcard links from each Cinder block storage Object to the End node.

8.2 Hierarchical Security Policy Composition with OpenStack

The application of the Hierarchical Security Policy Formal Model process to the OpenStack system necessitates heavy use of the Composition operation, as the majority of OpenStack system security policy files are found on disparate systems in the OpenStack environment. For the example OpenStack deployment featured in this thesis, three main system security files are extracted: the Keystone identity database, the Cinder block storage permissions configuration file, and the Cinder block storage database. Each of these system security policy files serves to provide an element set used in the application of the Hierarchical Security Policy Formal Model to the OpenStack deployment. The Keystone identity database

provides the Subjects of the policy model, including the Users, Roles and other Session information, the Cinder permissions file provides the Actions, including the Permissions and the Actions they allow, and the Cinder database, which manages the block storage Objects used in this example deployment. OpenStack also provides an authenticator in the form its Policy Enforcer, which authenticates Sessions for access against the Permissions of the system. This authenticator is used to evaluate the hierarchical policy model policy paths. Note that, while the definition of the Composition operation states that the policy paths may be reevaluated after the operation completes, the implementation of the application of the Hierarchical Security Policy Formal Model to OpenStack waits until all the constituent system security policy artifacts have been Composed to evaluate policy paths. If additional OpenStack services were to be included in the example deployment, their inclusion would perform Composition on their each of their element sets derived from system security policy artifacts before reevaluating the Hierarchical Security Policy Formal Model policy paths for the entire Model. During the application of the Hierarchical Security Policy Formal Model process to the example OpenStack deployment there are five meaningful combinations of system security policy artifacts which prompt Composition: Users with Roles, Users and Roles with Session data, Permissions with non-Permission Actions, Subjects with Actions, and Subjects and Actions with Objects. These Compositions take the form of Subject-Subject, Subject-Subject, Action-Action, Subject-Action, and Action-Object respectively.

The first and second Compositions deal with the system security policy artifacts identified from the Keystone identity database. The results of this Composition are displayed in Figure 8.1. In this case, each database table relating OpenStack Role-Based Access Control information is treated as its own system security policy file. Additionally a fourth table details the pairings of OpenStack Users, Roles and Session data, which is used to create partial policy paths at this step of the hierarchical policy modeling process. The first Composition pairs the Users with the Roles they hold. Both the User unique identifiers and the Role unique identifiers are found in their respective tables in the Keystone database. These unique identifiers are then mapped, along with the Project in which the User holds the Role, in the Keystone database, providing an association of Users and Roles in the OpenStack deployment. This association can then be used to provide partial policy paths across the Users and Roles Subject subsets' directed acyclic graphs. The second Composition pairs the result of the previously constructed Users and Roles Composition with the additional Session data held in the Keystone identity database. For the purposes of this example OpenStack deployment, the Session data contains the OpenStack Project in which the User holds the Role. The second Composition is implemented similarly to the first, as uses the Keystone database to retrieve the unique identifier of each Project in the OpenStack deployment, and uses the same database to associate these OpenStack Projects with the User and Role pairings created by the first

Composition.

The third Composition in the application of the Hierarchical Security Policy Formal Model process to the OpenStack platform combines the sets of Permissions and OpenStack Actions retrieved from the Cinder service policy configuration file. The results of this Composition are displayed in Figures 8.2 and 8.3. This Composition is implemented based on the permission language rule value associated with each Action key in the Cinder service policy configuration file. This dictionary mapping is used to create the partial policy paths of this Composition. Notably, the wildcard policy paths may be used for these connections as any Session which satisfies a Permission has authentication to perform the associated Action.

As there is only one system security policy file detailing the Objects in the example OpenStack deployment, only Single Model Composition is required to prepare the Objects element set to maintain the origin of the system security policy artifacts and establish partial policy paths. This Composition is displayed in Figure 8.4.

The fourth Composition is performed with the Subjects constructed in the first Composition, and the Actions constructed in the third. This Composition uses the OpenStack Policy Enforcer to evaluate attempted Session access to Actions against the Permissions required by the Action. If the Policy Enforcer allows access to the Action, the partial policy path constructed during the first Composition representing the Session is extended to the required Permission. Note that a Session may satisfy more than one Permission, and the partial policy path created during the first Composition may be duplicated to retain the uniqueness of the policy paths.

Figures 8.5 and 8.6 display the Differing Formal Model Composition between the Subjects and Actions of the OpenStack deployment's Hierarchical Security Policy Formal Model. In addition to the maintenance of the hierarchies in the individual directed acyclic graphs, this Differing Model Composition is able to extend the partial policy paths by connecting the partial policy paths in the Subjects to the partial policy paths in the Actions. This extension is performed by querying the authenticator oracle for the OpenStack deployment, providing the Session information and the Permission to be tested. Note that this application of the Hierarchical Security Policy Formal Model for Role-Based Access Control only matches against the rule language Permissions, and allows the hierarchy present in the Permissions directed acyclic graph to extend Session access to its fullest. Additionally, each of the three existing uniquely identified partial policy paths now have potential to be split during traversal of the Actions directed acyclic graph.

The fifth and final Composition seeks to extend the previously created partial policy paths of Subjects and Actions through the Objects to which Sessions are allowed access. This again is accomplished by

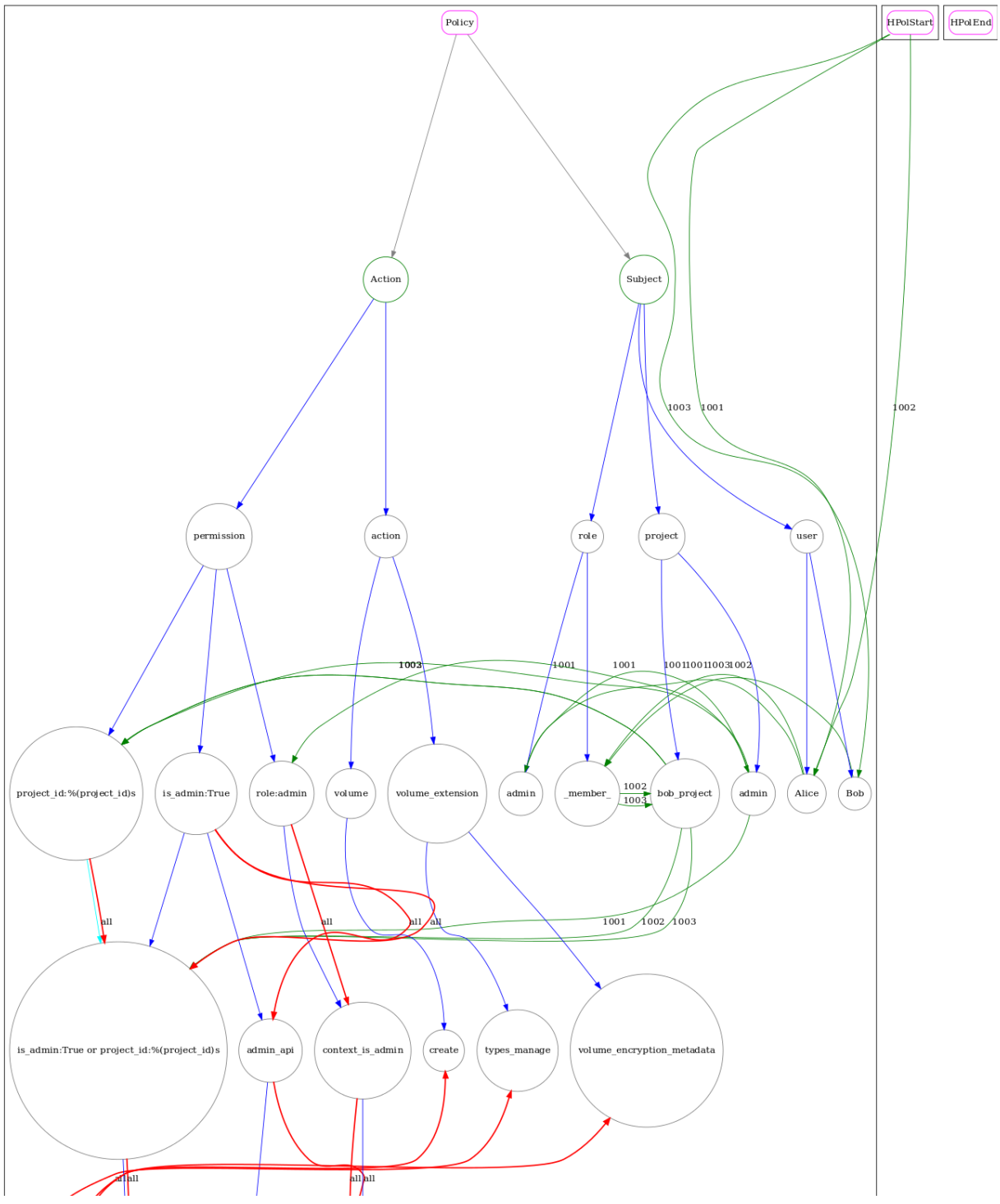


Figure 8.5: OpenStack Subject Action Composition Policy Paths

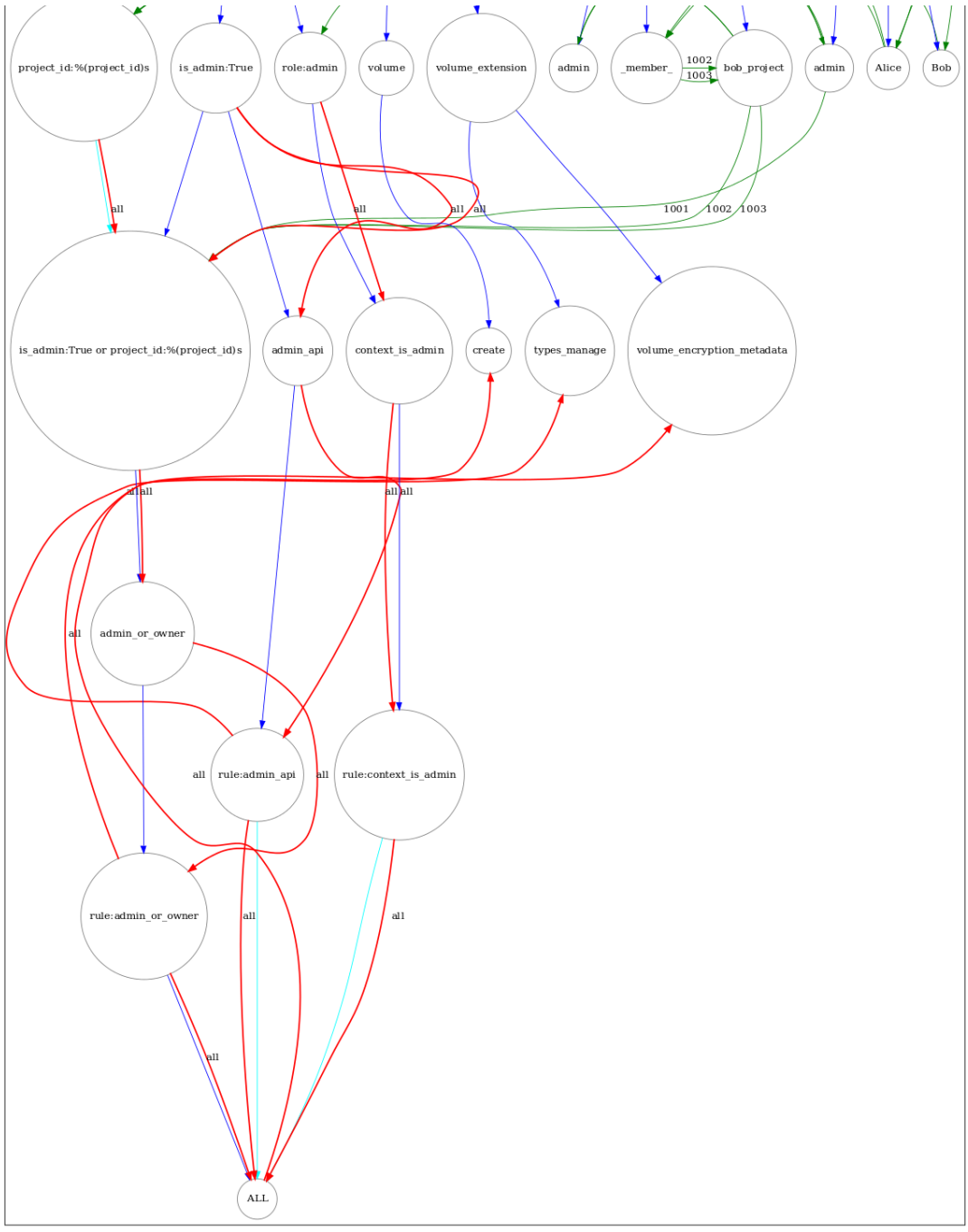


Figure 8.6: OpenStack Subject Action Composition Extended

use of the OpenStack Policy Enforcer, which checks if a Session has Permission to perform an Action on a given Cinder block storage Object. The fourth and fifth Composition may be implemented to occur simultaneously but, to maintain generality of the Composition operation, are described as separate and implemented separately for the purposes of this thesis.

After all five Composition have been completed, the authenticator is able to evaluate the Subjects, Actions and Objects, creating whole Hierarchical Security Policy Formal Model policy paths specific to the example OpenStack deployment as described previously in this Chapter. After the authenticator has been used to create policy paths for the Hierarchical Security Policy Formal Model, the Model is ready to be used by the logical database or immediately by the system security policy manager.

Figures 8.7 and 8.8 display the final Differing Formal Model Composition between the results of the fourth Composition and the Objects element set. With this full Model, the partial policy paths are able to be extended into complete policy paths which successfully enumerate access allowed to users of the system. As the policy path connection between Subject and Action directed acyclic graphs was made in a previous Composition, the only connection remaining is that between the Action and Object directed acyclic graphs. As the existing partial policy paths contain information about access from Sessions to non-Permission Actions through their associated Permissions, all that must be done is evaluate the partial policy paths against possible Objects. To do so, partial policy paths are extended from their current furthest node, the non-Permission Action, to the Project denoted in the extraneous Session information held in the Subjects of the path being evaluated. These connections create path-specific links between the non-Permission Actions and the Object directed acyclic graph. Once the partial policy paths have been connected this way, they are extended to traverse the Objects and connect to the End node. This forms complete policy paths which begin at the Start node, traverse the Subjects, Actions, and Objects, and connect to the End node.

8.3 Policy File Reconstruction with OpenStack

Once the Hierarchical Security Policy Formal Model has been applied to OpenStack and presented to the system security policy manager for verification, it is likely that the manager will wish to modify the system security policy and push the modifications back down to the individual systems in their domain. Recall what was created by the application of the Hierarchical Security Policy Formal Model process and the origin markings left on system security policy artifacts during the process. All the elements of the Subject set and its subsets originate from the Keystone identity database, all of the Actions originate from the Cinder policy configuration file, and all the Objects originate from the Cinder database. Additionally,

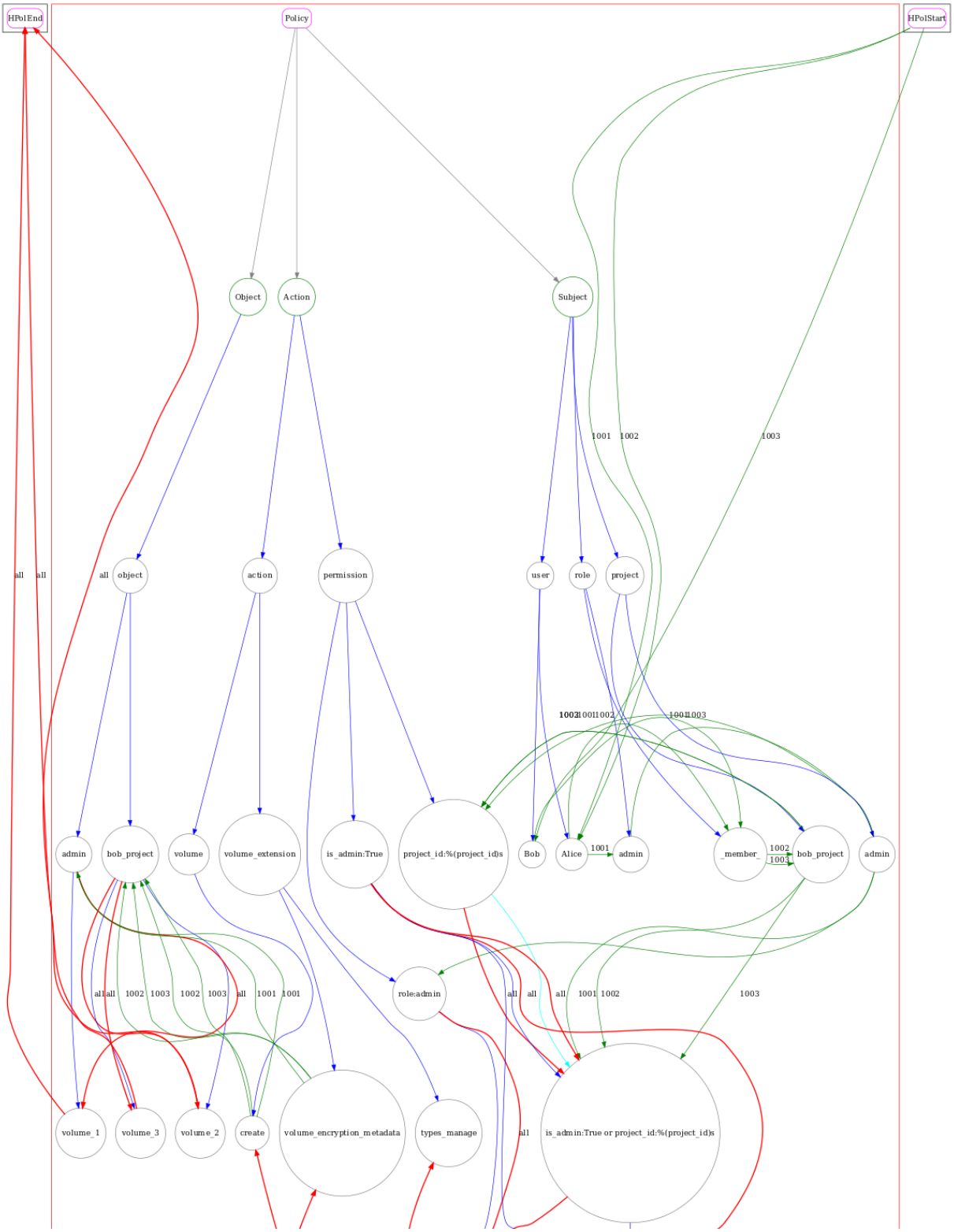


Figure 8.7: OpenStack Full Composition

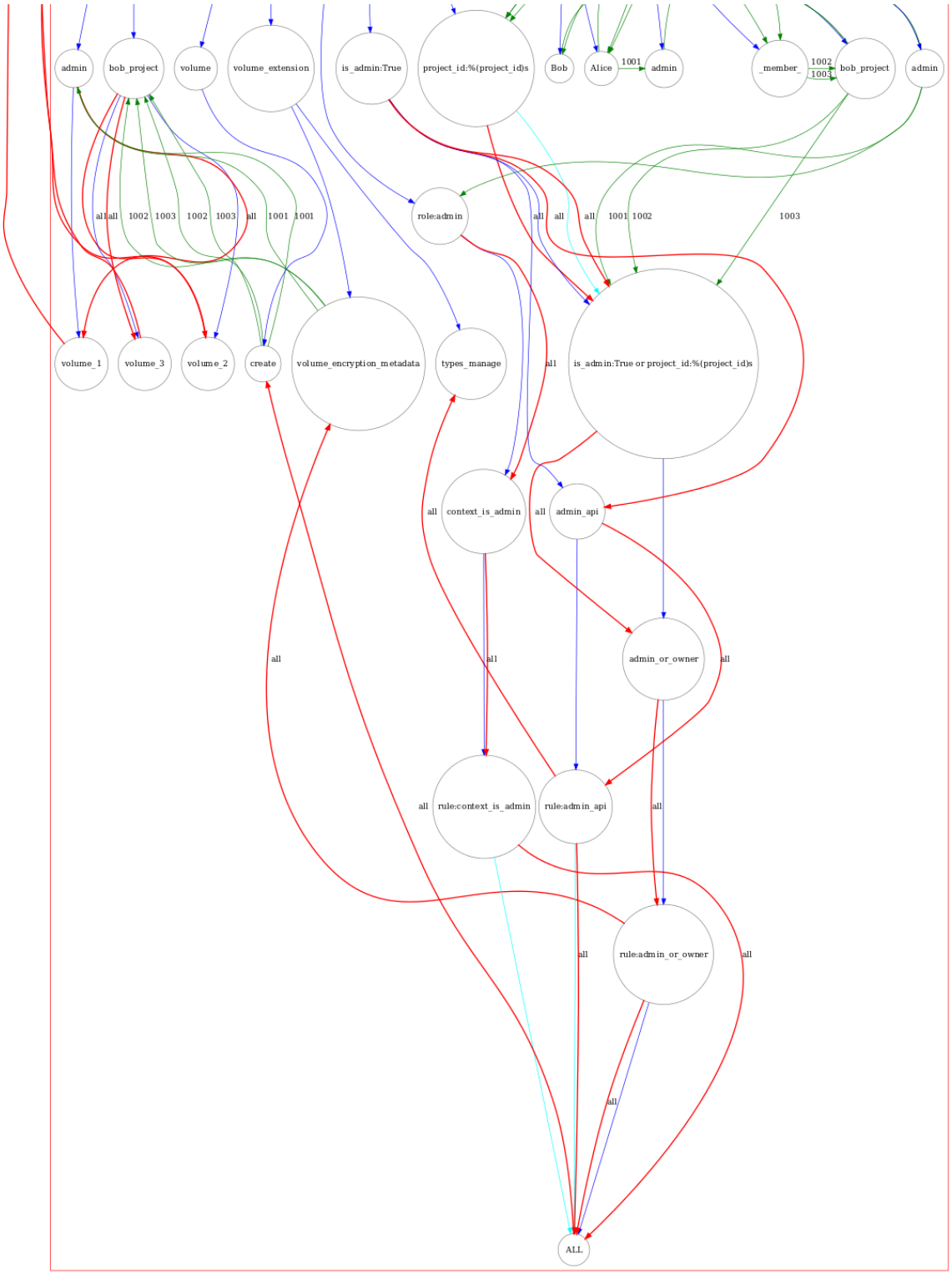


Figure 8.8: OpenStack Full Composition Extended

the system which performed the Hierarchical Security Policy Formal Model process should have access to the original system security policy files extracted from the target system. These files will be used to recreate the required tables in both the Keystone and Cinder databases.

In order to reconstruct the system security policy files, the reconstruction process must examine each of the three element sets. First, the directed acyclic graph of the Subject set is examined to collect the Users, Roles and Projects managed by the OpenStack deployment. With each of these also comes its closely coupled information; information with which it shares a row in the Keystone database. Second, the Subjects of the policy paths must be examined to recover pairings of Users, Roles and Projects. After these are performed, the Subjects of the Model are ready to be reintegrated with the original Keystone database system security policy file. The Action set directed acyclic graph is examined to extract the definitions of the rule aliases, and the policy paths overlaying the Action graph are used to recover the mapping between rules and their target Actions. Once these are performed, the Cinder policy configuration file is ready to be reintegrated. Note that the key-value pairs extracted from the Action directed acyclic graph and overlaid policy paths are enough to completely replicate the Cinder policy file, and no original system configuration file is necessary. Finally, the Object set of the Model is examined similarly to the Subject set, returning the Cinder database used to map specific block storage objects to Projects in the OpenStack environment. This allows the Objects of the Model to be reintegrated into the Cinder database retained by the system performing the Model evaluations.

Once all three of the primary system security policy files have been reintegrated with the system security policy artifacts used by the application of the Hierarchical Security Policy Formal Model, they are ready to be redistributed to their respective systems and redeployed. For both the Keystone and Cinder databases, a script is used which overwrites the existing database on the system with the provided database. In this case, the databases created by the reconstruction are used to overwrite the databases existing on the original target systems. The Cinder configuration policy file can simply overwrite the existing policy file, which immediately replaces the system security policy with the security policy reflected in the new file.

While modification of these files is not specifically addressed, the methods by which the system security policy files are reconstructed, reintegrated and redeployed generically apply to any system security policy artifact present in the model which is compatible with the artifacts from the original target system security policy file. Thus, if the instantiation of the Hierarchical Security Policy Formal Model applied to OpenStack is modified, either by direct manipulation of the Model or through a higher-level interface, the system security policy files should be successfully reconstructed, reintegrated and redeployed.

Chapter 9: Future Work

The Hierarchical Security Policy Formal Model is an ongoing research project and, as such, plenty of opportunity exists for further work and exploration. The future work detailed in this thesis expands from the contributions of this thesis, though future work is definitely present in other areas [3] [17]. Themes of the future work described below include expansion, integration, applicability and scalability. Each of these themes is important in its own right, though addressing an individual future work item likely entails research consideration of the others.

9.1 Further Application to Role-Based Access Control Systems

The application of the Hierarchical Security Policy Formal Model uses OpenStack to prototype the application of the Model to Role-Based Access Control systems. As such, the implementation of the Hierarchical Security Policy Formal Model is specifically designed to be applied to OpenStack. To further increase applicability of the Hierarchical Security Policy Formal Model, a software engineering effort should be undertaken which implements the Hierarchical Security Policy Formal Model in a generic fashion with applicability to other Role-Based Access Control systems.

Additionally, work on the Hierarchical Security Policy Formal Model application to Role-Based Access Control systems should seek out other challenging systems utilizing Role-Based Access Control. Microsoft's Active Directory seems like a perfect fit for an additional application, as it is widely used in enterprise networks, and contains information similar enough to OpenStack and Security Enhanced Linux that the existing case studies may be used as a springboard for future work.

Additional work should also be undertaken refining the element set mapping to Role-Based Access Control systems. Specifically, implementation of further applications of the Model to Role-Based Access Control systems which exhibit a form of User hierarchy, such as User Groups, or a form of Role hierarchy, such as enterprise organizational charts.

9.2 Integration of Enterprise-Level Policy Mapping

This future work topic has been alluded to in previous sections of this thesis, and boils down to a point: application of the Hierarchical Security Policy Formal Model to any system depends on the system's place in an actual enterprise network. The system's place in the network is what provides the system security policy manager with the relevant meaning of the Model. In the application of the Hierarchical Security Policy Formal Model process to Security Enhanced Linux, the resulting hierarchical policy model is used to verify correct implementation of Bell-LaPadula across multiple systems [17]. This verification process

is only available to the system security policy manager because the manager knows that the system User whose access is being verified is the same actual person across both systems. This prompts future work which would create an application of the Hierarchical Security Policy Formal Model for a system which maps actual people to their various accounts throughout an enterprise network. Making this addition to the Hierarchical Security Policy Formal Model would allow system security policy managers to administer their knowledge of the system they manage to the Model, paving the way for future mappings between systems where the actual people using the system control differently identified accounts. Pursuit of this topic would also begin to open the door to fully support automation of system security policy verification and construction through the existing Hierarchical Security Policy Formal Model operations. This, when combined with something similar to RSL99 [1] or RCL2000 [2], could be used to automatically verify Role-Based Access Control systems across multiple devices where actual system users control differently identified accounts.

9.3 Expanding Higher-level Abstractions to Support Policy Modification

This thesis details the reverse application of the Hierarchical Security Policy Formal Model steps such that original system security policy file are reconstructed and prepared to be redeployed to the target systems. As noted in Chapter 7, which details the reconstruction process, there does not currently exist an interface suitable for use by the system security policy managers which would allow for modification of system security policy artifacts, the directed acyclic graphs, or the overlaid policy paths. Development of such an interface and its integration to the existing logical database interface usable by system security policy mangers is necessary to provide the tools required to manage system security policy. Additionally, the implementation of the interface is required to maintain the ability of the reconstruction process to derive system security policy artifact origins and successfully reconstruct them into their original policy files. This means that any implementation would also be required to provide the means to satisfy the compatibility requirements of the reconstruction process; it would require a generic implementation operating at the artifact and policy path level which would determine the information required from the system security policy manager to successfully make modifications to the system security policy. If the Hierarchal Security Policy Formal Model is to become useful in an enterprise environment, verification of system security policy is not enough; the Model must also be able to provide system security policy managers the ability to dictate system security policy across their enterprise domain.

9.4 Addressing Scalability Concerns

As the Hierarchical Security Policy Formal Model seeks to provide system security policy managers with a holistic view of their entire system domain, scalability of the Model to large domains becomes a concern. In its application to Security Enhanced Linux, the Hierarchical Security Policy Formal Model was only able to support a subset of the entire system security policy [17]. This issue has multiple possible solutions: assure the system used to perform the Hierarchical Security Policy Formal Model process is capable enough to handle large datasets in appropriate time frames, enhance the high-level abstraction of the Model to compact and simplify system security policies into more manageable numbers, or provide methods for system security policy selection which limit the scope of the Model. Each of these solutions comes with their own downsides and difficulties, but proper exploration of this topic is necessary to further expand the applicability of the Hierarchical Security Policy Formal Model.

Chapter 10: Summary and Conclusions

10.1 Summary

Recall the original research questions posed by this thesis:

1. Can the Hierarchical Policy Model be applied to Role-Based Access Control systems?
2. Is there a method by which the Hierarchical Policy Model can combine subpolicies from separate parts of a target system to form a complete policy?
3. Can the Hierarchical Policy Model return the original system security policy artifacts from the constructed system security policy model?

Each of these research questions were then addressed with their respective research objectives:

1. Provide a generic method by which the Hierarchical Security Policy Formal Model may be applied to Role-Based Access Control systems, and provide an instance of this application to an actual Role-Based Access Control system.
2. Provide a method by which the combination of separate policies from a common target system may be made into a single system security policy model.
3. Provide a method by which the original system security policy artifacts may be reconstructed from a system security policy model.

Each of these objectives were met through application of the Hierarchical Security Policy Formal Model to Role-Based Access Control systems, specifically application to the OpenStack platform. Additionally, this thesis detailed a mapping of the Role-Based Access Control entity sets [15] to the Hierarchical Security Policy Formal Model element sets. This mapping was used throughout this thesis to present the results of the research objectives in a generic way, applicable to any Role-Based Access Control system.

10.2 Conclusions

In order to assist system security policy managers in successfully assuring the confidentiality, integrity and availability of their system, the Hierarchical Security Policy Formal Model seeks to abstract the management process, providing a interface for simple policy management. Policy management performed in the Hierarchical Security Policy Formal Model is not effected by the location of the system security policy files in the domain and does not effect the uptime of production systems, while still providing

system security policy managers with the tools necessary to verify correct and intended operation of systems across their domain.

This thesis first presented an application of the Hierarchical Security Policy Formal Model to Role-Based Access Control systems, both in a generic mapping of element and entity sets, and with an example implementation applied to the OpenStack cloud computation platform. Second, this thesis presented a Formal definition of the Compose operation used to construct hierarchical policy models. Finally, this thesis presented a method by which the Hierarchical Security Policy Formal Model may be used by the system security policy managers to push potentially modified system security policy files to the original target systems.

Each of these three contributions furthers the goal of the Hierarchical Security Policy Formal Model project, which is to be able to provide system security policy managers with a holistic system view for verification and modification. The contributions are each listed both in a generic fashion, applicable to hierarchical policy models outside this thesis, and as implementation of the Hierarchical Security Policy Formal Model applied to the OpenStack system. This thesis also provides an outline of various paths for future development of the Hierarchical Security Policy Formal Model, its associated processes, and potential future applications.

Bibliography

[1] Gail-Joon Ahn and Ravi Sandhu. The rsl99 language for role-based separation of duty constraints. In *Proceedings of the Fourth ACM Workshop on Role-based Access Control*, RBAC '99, pages 43–54, New York, NY, USA, 1999. ACM.

[2] Gail-Joon Ahn and Ravi Sandhu. Role-based authorization constraints specification. *ACM Trans. Inf. Syst. Secur.*, 3(4):207–226, November 2000.

[3] Matthew Brown. *Hierarchical Formal Modeling and Verification of Router Policies with an Applied Case Study to Cisco Router Configurations*. Master's Thesis, 2016.

[4] CHISEL Research Group. Shrimp. http://www.thechiselgroup.org/shrimp/, March 2017.

[5] A. Jillepalli and D. C. d. Leon. An architecture for a policy-oriented web browser management system: Hifipol: Browser. In *2016 IEEE 40th Annual Computer Software and Applications Conference (COMPSAC)*, volume 2, pages 382–387, June 2016.

[6] A. Jillepalli, D. C. de Leon, S. Steiner, and F. T. Sheldon. Hermes: A high-level policy language for high-granularity enterprise-wide secure browser configuration management. In *2016 IEEE Symposium Series on Computational Intelligence (SSCI)*, pages 1–9, Dec 2016.

[7] Ammar Masood, James Joshi, Arif Ghafoor, and Basit Shafiq. A role-based access control policy verification framework for real-time systems. *10th IEEE International Workshop on Object-Oriented Real-Time Dependable Systems*, 00:13–20, 2005.

[8] Matunda Nyanchama and Sylvia Osborn. Access rights administration in role-based security systems. In *Database Security VIII: Status and Prospects*, pages 37–56. North-Holland, 1994.

[9] Matunda Nyanchama and Sylvia Osborn. *Modeling Mandatory Access Control in Role-Based Security Systems*, pages 129–144. Springer US, Boston, MA, 1996.

[10] Matunda Nyanchama and Sylvia Osborn. The role graph model and conflict of interest. *ACM Trans. Inf. Syst. Secur.*, 2(1):3–33, February 1999.

[11] James L. Peterson. Petri nets. *ACM Comput. Surv.*, 9(3):223–252, September 1977.

[12] The OpenStack Project. Appendix a. the policy.json file. https://docs.openstack.org/kilo/config-reference/content/policy-json-file.html, March 2017.

[13] The OpenStack Project. Openstack. https://www.openstack.org/, March 2017.

[14] Jerome H. Saltzer and Michael D. Schroeder. The protection of information in computer systems, 1975.

[15] Ravi S. Sandhu, Edward J. Coyne, Hal L. Feinstein, and Charles E. Youman. Role-based access control models. *Computer*, 29(2):38–47, February 1996.

[16] Luay A Wahsheh, Conte de Leon Daniel, and Jim Alves-Foss. Formal verification and visualization of security policies. *Journal of Computers*, 3(6):22–31, 2008.

[17] Jared Zook. *High-level modeling and verification of mandatory access control policies across multiple security-enhanced linux devices.* Master's Thesis, 2016.