



Design and Implementation of the SELinux Policy Management Server

March 1, 2006

Karl MacMillan, Joshua Brindle, Frank Mayer,
Dave Caplan, and Jason Tang

Copyright ©2006 Tresys Technology, LLC. All Rights Reserved.

Other names and brands may be claimed as the property of others. Information regarding third party products is provided solely for educational purposes. Tresys Technology, LLC is not responsible for the performance or support of third party products and does not make any representations or warranties whatsoever regarding quality, reliability, functionality, or compatibility of these devices or products.

Abstract

Policy management is an important capability for effectively using and deploying SELinux. There are several challenges to managing the SELinux policy in a production environment, which include properly integrating package management with policy management. It is also important to address the lack of granular security controls for policy updates. In the Policy Management Server we have designed an access control mechanism for policy updates that provides fine-grained access using SELinux's own security enhancements. In this paper we present an overview of the Policy Management Server, including its goals, design, and implementation status.

1. Introduction

The ability to change, update, and reload a configurable policy is one of the features that sets Security Enhanced Linux (SELinux) apart from other mandatory access control mechanisms. This flexibility introduces new challenges that we must address to in order for these features to reach their potential. Policy must be deployed, updated, and customized in a production environment, all closely integrated with other parts of the system, such as the package manager. The SELinux policy management server (PMS) is designed to address these challenges.

The PMS and associated infrastructure enhance the ability to deploy, update, and customize policy in a production environment. It also adds fine-grained access control over changes to the policy to allow the safe delegation of administrative function to one or more administrators or administrative tools. In the future, the policy management server will be extended to support the management of policy across several networked systems.

2. Policy Management

We define policy management as the set of tasks required to successfully use policy in a production environment. Development activities, like creating new policies for an application, are deliberately excluded. Policy development tools will likely interact with policy management infrastructure, but the main goal of policy management is the support of production environments. Policy management is composed of three basic activities:

1. Deployment – installing initial policy or policy updates onto a target machine. The policy may be vendor supplied or locally customized.
2. Customization – configuring the policy for the local system. Common examples of customizations include port labeling, boolean settings, and adding users.
3. Querying – obtaining information about the currently installed policy.

2.1 Previous Work

The loadable module work that we introduced last year [1] provides the basis for policy management. Loadable modules allow policies to be divided into loosely coupled modules and linked with a special base module on a running system. This ability to link at load time allows, for example, an applications policy to be bundled with the application and added to the system at application install time. Included in the loadable module work is a transactional module store that allowed the storage and management of one or more policy modules. The module store, which is a structured set of files and directories protected by policy, is managed by a program called semodule. Semodule allows the insertion, upgrading, or removal of policy modules in a manner that maintains a consistent and coherent policy at all times.

2.2 Policy Management Server

Loadable policy modules support our first policy management task, deployment, but do not directly support the other activities. To support these tasks we started the policy management server project. Figure 0 shows the basic architecture of the PMS. There are three main components:

1. **Semanage Store** – a generalized version of the loadable module store that contains policy modules and local policy customizations.
2. **Libsemanage** – a C library that allows clients to manage the semanage store. There are two modes of operation. The client can manipulate the semanage store directly or indirectly through the policy management server (using Unix domain sockets for IPC).
3. **Policy Management Server** – a daemon that encapsulates access to the policy and allows clients to request changes to the semanage store. The PMS adds additional features not possible by allowing clients direct access to the semanage store.

In this architecture, a client “connects” to the semanage store directly or indirectly through the PMS and changes or queries the policy. The libsemanage API is the same regardless of whether the client will be directly manipulating the semanage store or making requests to the PMS. The API has functions for adding, removing, and upgrading modules, changing locally configurable aspects of the policy (adding users, setting persistent boolean states, etc.), and querying the policy (listing the modules, users, booleans, etc.). An example session after the client connection is:

1. Query the policy for persistent boolean settings. A list of booleans and their values is returned.
2. Begin a transaction. Modifications to the policy are required to occur within a transaction boundary. Changes are made active only when the transaction is committed and the policy checked for consistency.
3. Add a new module.
4. Change the default value for a boolean added by the new module.
5. Commit the transaction. At this point the modules are linked together with the base module, the policy is expanded to the kernel format, and the local customizations are applied. If these steps are successful the new policy is made active; if an error occurs no changes are made to the policy. All queries are suspended while the policy switch is made.

In this example it is clear that local customizations are distinct from the addition or removal of modules. This is an important design choice to enable simpler interaction with package management systems. On current SELinux systems the package manager (e.g., rpm) installs and ‘owns’ the binary policy file. If any changes are made to the policy, rpm cannot resolve differences between the modified policy and a policy in an upgrade. This makes it difficult to support both local policy customization and policy upgrades. To partially address this problem, support for overriding default boolean states and adding users without changing the policy directly was added to `load_policy` for existing system. This allows common customizations to be made without changing the files managed by the package manager. The policy management infrastructure is expanding this idea to include additional local customization. Additionally, larger customizations can be added through custom modules. This allows the policy modules and base modules managed by the package manager to remain unchanged while still supporting common local customizations, allowing simpler upgrades of policy from packages.

2.3 Policy Access Control

The importance of the policy to the security of the system means that it is important to limit which users, roles, and applications can modify the policy. SELinux currently provides controls for the loading of policy and access to the saved policy files. The control over policy modification is very limited. A domain is able to either change all of the policy or none of the policy. A user management application would, for example, be able to associate all roles with users or change any allow rule for any type.

The inability to enforce least-privilege on policy changes makes it difficult to securely delegate policy administration and increases the risks from errors in policy management applications. Once an application or administrator is granted access to the policy there is no limit on what they may modify.

The PMS addresses these issues by introducing fine-grained access control over policy change. This was accomplished by:

1. Encapsulating the policy – the PMS mediates all access to the policy allowing it to enforce access control. This is one reason that a separate daemon is required.
2. Creating a policy object model – the policy components are represented by object classes that are

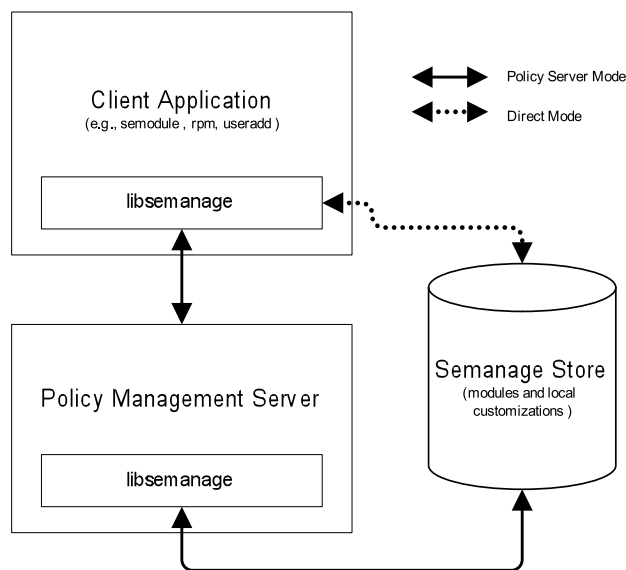


Figure 0 - Basic Policy Management Architecture

controlled by the PMS. Access to the policy objects can then be specified using standard policy rules.

3. Adding policy language extensions – policy language extensions were added for labeling policy objects and to ease specification of access to policy objects.

The goals of this design are to allow the concise specification of fine-grained access control over the policy and to leverage the existing SELinux policy concepts. Though conceptually counterintuitive initially, we feel that it is natural to use policy to specify access control over policy. It only makes sense to use SELinux fine-grained access control mechanism to protect the policy itself. The “policy for the policy” is loosely termed meta-policy.

The PMS access control system implements two important concepts: policy object module and namespace hierarchy.

2.3.1 Policy Object Model

We developed an object model for policy resources that abstracts elements of the policy (e.g., types, users, roles) into object classes. The object classes currently defined for the meta-policy are: `policy`, `policy.type`, `policy.attribute`, `policy.user`, `policy.role`, `policy.bool`, and `policy.class`.

Like all object classes in SELinux, each meta-policy object class has defined permissions representing access to policy elements. For example, `use.src.allow` is a permission defined for object class “`policy.type`.” A domain allowed this permission may use types labeled with the target type as the source in new allow rules. See Example 1. We understand that talking about allowing access to types based on the type of a type becomes confusing with the overloaded use of the word `type`. This is the conceptual challenge when working with meta-policy!

Treating policy elements as object classes allows regular type enforcement policy to be written for policy elements. These rules are no different from other SELinux rules except that they use object classes and permissions specific to the policy itself. The meta-policy objects are labeled based on names using the new `policycon` statement. Labeling uses the longest partial match, similar to `genfs`.

A full explanation of the policy object classes and permissions is beyond the scope of this paper. More information is available from <http://sepolicy-server.sf.net>.

To illustrate these concepts, consider the example policy in Example 2, which is a piece of a policy that gives `apache` the ability to bind on `http_port_t` and `http_cache_port_t` (tcp 80 and 8080), and also to connect to ports labeled `mysqld_port_t`. This piece of policy is just an ordinary example as would be used today, and not part of the meta-policy. To use `apache` with PostgreSQL instead of MySQL the policy needs to be changed to allow connection to `postgres_port_t`.

When the PMS is active, the domain adding the new policy rules would need access to policy object classes. Example 1 shows a portion of the required meta-policy required to give the `apache` administrator domain (`apache_admin_t`) the required access. First, the types are labeled using the `policycon` statement. Notice that the second and third statements label both the `mysqld_port_t` and `postgres_port_t` types with the same context. This preserves the important concept that types are an equivalence class. After labeling, the `apache_admin_t` domains is granted permission to use all types (i.e., the object class of the rule is `type`) labeled `apache_types_t` in the source or target of an allow rule (i.e., the permissions granted are `use.src.allow` and `use.tgt.allow`).

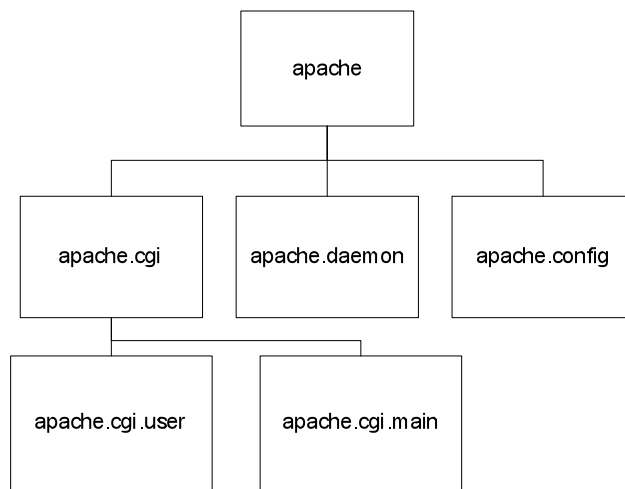
2.3.1 Namespace Hierarchy

This PMS object model can effectively control many policy changes, but the control is coarse grained in some important areas. For example, it is impossible to control what pairs of types can be used in allow rules; a domain can use any type that it is allowed to use as in the source of an allow rule with any type it is allowed to use in the target. This is somewhat similar to the way that `relabelfrom` and `relabelto` permissions work for file system objects. It is also difficult to express policy on previously unknown policy components (e.g., enforce access on a type added by the domain changing the policy). Extending the object model to address these issues would require significant changes to the policy language and could result in a very complex and verbose meta-policy. [2]

To address these challenges, we recognized the need for further additions to the SELinux policy language and an additional enforcement model. These additions add the concept of namespace hierarchy to the policy and enforce constraints on policy statements based on this hierarchy. These additions provide a

powerful semantic model that helps to preserve policy intent during policy change without requiring verbose meta-policy statements. There are two main components to this change: policy language syntax for declaring hierarchical policy elements and a set of additional constraints enforced on policy statements.

The SELinux policy contains separate namespaces for users, roles, types, and object classes. Currently, these namespaces are flat. The hierarchy syntax introduces the "." character as a delimiter into the role, type, and object class namespaces. Using this delimiter it is possible to declare an element in a namespace that is a child of another element. For example, declaring the type `apache.daemon` creates a new type that is a child of the type `apache`. See Figure 0 for a more complete example. Each of the child elements is completely separate from the parent; no attributes or allowed access is inherited from the parent. There is no limit on the number of children a type can have or the depth of the hierarchy. Also, the declaration of a



```

type apache, file_type, sysadmfile, domain;
type apache.cgi, domain, privlog;
type apache.cgi.user, domain;
type apache.cgi.main, domain, privlog;
type apache.daemon, domain;
type apache.config, file_type;
  
```

Figure 0 - An example namespace hierarchy.

child does not implicitly declare the parent, though it is possible to declare a parent after the child. Using this syntax, it is possible to create complex trees of roles, types, or object classes.

In addition to the policy syntax, the hierarchy feature creates additional policy semantics in the form of implicit constraints on roles and types. These constraints, which will be enforced by the compiler and the PMS, create a policy semantic that can be combined with the policy object model to enable efficient delegation of policy administration while ensuring that the policy is not changed in a way that changes the intent of the policy. The constraints limit the access of a child to no more access than that of the parent. For types, this means that a child type cannot have an attribute that the parent does not have and cannot have an allowed access relationship with another type and object class that the parent does not have. For roles, this means that a child role cannot be authorized for a type for which the parent is not authorized. Only the immediate parent is considered in these constraints.

The policy object model and namespace hierarchy together provide fine grained access control on the policy and allows for secure delegation of portions of the policy. For example, the system administrator

may define an apache policy that meets the needs of the system by confining and isolating apache from other important processes. The apache administrator may then want to further confine parts of apache, such as CGI scripts. This won't change the intent of the policy from the system perspective, but within apache a hierarchy has been made to limit the access of untrusted or unknown code that apache runs as part of its normal workload. Apache is then more protected while the rest of the system is unaffected. The apache administrator would only have access granted by the meta-policy to change policy for types that are children of the top-level apache type. This prevents him from changing other parts of the policy and from creating child domains for CGI scripts that exceed the access granted by the system administrator.

3. Future Work

The policy management server provides increased management capabilities, including access control on the policy, but it is still limited to the single host. What is needed in the future is the ability to manage and enforce a single overall policy across multiple hosts, creating what we will call a policy domain. The policy management server provides a part of the infrastructure required to create this capability, but it must be significantly extended with new functionality. Additionally, the creation of a policy domain also creates additional complications not seen when managing and enforcing policy on a single host that require research.

In the simplest model of a "policy network domain," a single policy is managed centrally and distributed to all of the hosts for local enforcement. The net effect of each host enforcing the same policy would be the enforcement of the overall security policy. In practice, this single policy model is not realistic because of local modification of policies required to support the variations in the hosts (e.g., local users or network settings). More complicated models might involve larger policy variations by distributing many, related policies across a group of hosts. This might arise in the presence of hosts with widely varying characteristics (e.g., a large database server and a cell phone). In this model, the overall security policy would be still be enforced by the individual policies on each host, but successfully creating a set of individual policies that achieve the desired net effect is more complex, especially when the hosts must cooperate to enforce the same access attempt, as would happen in remote file access or remote procedure calls. [3] [4]

In both of these models, the complications include:

1. Atomic policy change – policy changes should take effect simultaneously across all hosts including revoking previously granted access.
2. Sharing modifications – some policy modifications will need to be propagated across hosts while others will remain local, sometimes depending not on the type of change but the specifics of the situation (e.g., sometimes users will be global and sometimes local).
3. Secure policy distribution – the policy changes will need to be distributed securely. This will likely require both authentication and the capability to define trust relationships between hosts.
4. Disparate policy coherence – multiple related, but disparate policies will introduce complications, including managing identifier namespaces, when hosts communicate or cooperatively enforce access. New policy development and analysis techniques will likely be needed to ensure that the overall security policy is enforced through the enforcement of the disparate policies.

References

[1] Macmillan, Karl. Core Policy Management Infrastructure for SELinux. SELinux Symposium, 3 Mar. 2005. <<http://www.selinux-symposium.org/2005/presentations/session3/3-2-macmillan.pdf>>.

[2] SELinux Policy Management Infrastructure Design. Tresys Technology. <<http://sepolicy-server.sourceforge.net/index.php?page=policy-management-design>>.

[3] Spencer, Ray, Stephen Smalley, Mike Hibler, David Anderson, and Jay Lepreau. The Flask Security Architecture: System Support for Diverse Security Policies. USENIX Security Symposium, Aug. 1999, University of Utah. <<http://www.cs.utah.edu/flux/papers/flask-usenixsec99.pdf>>.

[4] Secure Computing Corporation. "Assurance in the Fluke Microkernel." Flask: Flux Advanced Security Kernel. <www.cs.utah.edu/flux/fluke/html/final.ps.gz>.

```
policycon type apache_t system_u:object_r:apache_types_t;
policycon type mysql_port_t system_u:object_r:port_types_t;
policycon type postgresql_port_t system_u:object_r:port_types_t;

allow apache_admin_t apache_types_t : policy.type { use.src.allow use.tgt.allow };
allow apache_admin_t port_types_t : policy.type { use.tgt.allow };
```

Example 1. Example apache admin policy.

```
# Create the apache_t domain for the webserver apache
daemon_domain(apache_t)
# Allow apache to bind the appropriate ports (80 and 8080)
allow apache { http_port_t http_cache_port_t }:tcp_socket name_bind;
# Allow apache to communicate with the MySQL database
allow apache { mysqld_port_t }:tcp_socket name_connect;

#... (rest of the apache policy) ...
```

Example 2. Apache Policy