Reference Policy for Security Enhanced Linux

Christopher J. PeBenito, Frank Mayer, Karl MacMillan
Tresys Technology

Abstract

The Reference Policy project is an effort to restructure the NSA example policy for SELinux, which has evolved through many years of community involvement and is the basis for nearly all sample SELinux Policy in use today. The Reference Policy is rigorously structured using modularity, layering, encapsulation, and abstraction, making it simpler to maintain, modify, and use. The goal of this restructuring is to allow greater adaptation and adoption of SELinux while maintaining the knowledge gained through the years of policy evolution, while increasing our ability to validate the security properties of a given SELinux policy.

1. Motivations

When expressing security goals for an application in an Security Enhanced Linux (SELinux) policy, it is important to have a strong foundation to build upon. The problem with the sample policy that evolved from the original NSA policy [1] is that the policy is difficult to understand, develop, and maintain unless you are intimately familiar with the SELinux enforcement mechanism and the policy language. In the example policy, source files have loose structure and policy modules are closely coupled. A policy writer must have detailed knowledge of most if not the entire policy in order to use it as a basis for new application policy modules. Creating third-party modules is difficult, requiring detailed understanding of type, role, user, attributes, etc. definitions throughout the entire policy. While most policies in use today, e.g., [2] [3], are based on the original NSA sample policy, the single largest complaint about SELinux is that writing policy is too difficult and complex.

Tresys started the Reference Policy [4] project to refactor the community knowledge gained through evolution of the NSA example policy, into a form that exhibits many of the strengths and features of modern software engineering, thereby making the policy more maintainable, verifiable, and useable.

2. Reference Policy Goals

The Reference Policy project has two primary goals:

- Security: Provide a better foundation for ensuring that security goals for SELinux systems and applications are achieved and verifiable; and
- Functionality: Provide better software engineering discipline to the structure of policy source files such that maintaining the policy is easier, adding application policy modules is simpler, and building high-level tools such as graphical policy editors is feasible.

2.1. Security Goals

Security is the primary reason for using the SELinux enhancements to Linux, and as such, it must also be the first priority for the Reference Policy. Commonly, when looking at the security of a system, it as viewed in a true or false state; the system is secure or it is not. In reality, this is not sufficient, as different systems have different purposes, and correspondingly, a different meaning of what is secure. One of the great features of SELinux, is its use of type enforcement as its mandatory access control mechanism, which allows policy writers to specify the security properties they need.

Reference Policy has four primary security goals, which are generally goals of any SELinux policy:

- Self-protection: for the system itself, the SELinux policy, and for each application (to be protect itself from other applications);
- Assurance: confidence in the correctness and completeness of all SELinux policy components, throughout the policy development lifecycle, including extensive use of least privilege and limitation of error propagation [5], as well as increased understandability;
- Secure extensibility: allowing specific applications and services to be added to the SELinux

policy while maintaining other security goals and assurance; and

- Improved role separation: role to be defined via least privilege and fine-grained role definitions to improve a current weakness of SELinux policies (i.e., overly powerful admin domains).

In our results to date, we believe we have made significant progress in all of these security goals except the last, improved role separation, which is in our future plans.

2.2. Functional Goals

There are several issues facing SELinux policy development that Reference Policy aims to address including:

- Managed complexity: reduce the amount of details that a policy writer must manage, making it simpler to create new policy modules without requiring intimate knowledge of the underlying policy implementation;

- Loadable modules support: support both existing, so-called *monolithic policies* as well as policies compatible with the emerging loadable policy modules infrastructure [6] to be built from the same policy source files;

- Better support for tools: structure the policy such that sophisticated policy development and analysis tools can work with policy sources more easily;

- Third-party support: enable third parties, such as OEMs, to create policy modules using stable, well defined interfaces;

- Improved comprehension: allow policy writers to more easily understand the policy by aggressively adding documentation, enabling them to make security relevant decisions; and

- Policy configurations: supports the strict and targeted policies, MLS, and the recently added MCS configuration [7] as build options, without requiring destructive changes to the policy or multiple policy source trees.

3. Concepts, Rules, and Conventions

The Reference Policy introduces or formalizes several new concepts in order to achieve our goals. It also uses strong rules for policy module constructions that help ensure these concepts are enforced.

3.1 Design Concepts

We defined several design concepts borrowed from modern software engineering to guide the Reference Policy project.

3.1.1. Layering

Layering in Reference Policy is a fairly weak principle, and is primarily used to help organize modules and ease user understandability. The deciding factor for which layer a module belongs is sometimes ambiguous. In general, modules are grouped by function in the system. Lower layer modules are generally included in most system policies, for example, to protect kernel resources, or startup or shutdown the system. Higher layer modules tend to be more optional, and are included as needed, such as for user applications or network-facing services. The layers currently defined are kernel, system, administration, services, and application layers.

3.1.2. Modularity

Modules are the smallest components in Reference Policy, grouping related policy statements for a particular domain and its resources. Groupings are mainly based on the similarity of function. Each module includes the policy resource (e.g., type, attributes) declarations, associated policy rules, and object labeling requirements. This form of a module forms the basis for encapsulation and abstraction of the policy statements, which we discuss further below. Each module generally corresponds to RPMs in a Red Hat system, which is a practical decision to ease system configuration.

The existing example policy has evolved a form of modularity that helps manage complexity, but in general still leaves modules tightly coupled. For the Reference Policy we are more strict in the organization and definition of modules, enforcing several strong rules by convention.

Each module has three component source files: 1) a private module policy (.te), 2) a set of external interfaces (.if), and 3) a file labeling policy (.fc). The private policy file contains the declarations and rules that are local to the module. The external interface file provides abstract access to types and attributes that are private to the module, for other modules' use. These interfaces are designed to be intuitive, reflecting access to some set of system capabilities. A policy developer should be able to use an interface, understanding its purpose, without having to understand how the purpose is implemented within the module. The file labeling policy files consists of file contexts statements, if any, associated with the module.

3.1.3. Encapsulation and Abstraction

The encapsulation rules for a module ensures that all of the module's implementation details are private to the module itself. This allows changes to the module's implementation without affecting other policy modules, reducing the coupling of policy modules.

The Reference Policy has several important policy writing rules which capture the important concepts of encapsulation and abstraction. Perhaps the most important rule is *type and attribute identifiers are private to a module*; thus, types and attributes may not be directly referenced by name outside of the module in which it is declared. This also means that *there are no global types or attributes*.

Abstraction is used to create higher-level concepts of access. This enables policy writers to make security relevant decisions, rather than being impeded with understanding all of the implementation details. M4 macros are used to create the abstractions, though in general we enforce strict conventions on the type of macros allowed, unlike the current practice of allow any form of macro that M4 can support[1]. Each macro contains only one concept, and rules are not added for convenience.

There are three types of macros supported by Reference Policy: interface, template, and support. Interface macros are those that export access to a module's internal types and/or transform a type, e.g., making a caller's type a "domain." Interfaces are one of the primary innovations of Reference Policy, providing most of the de-coupling of module implementations. Template macros are a special form of interfaces that create and/or manipulate derived types on behalf of the callers. While the derived types are conceptually private to the caller, their implementation details are contained wholly within the module that creates the template macros. These macros are used , for example, to create the types and rules for ssh domains and keys for each role. Support macros parallel many of the "core macros" in the NSA example policy, providing common policy patterns. Examples include domain transitions and object class permission sets. Support macros are essentially "helper functions" that are used solely to help simplify the private, internal implementation of a module (i.e., they are never interfaces).

[1] In the future, we may remove M4 all together in favor of a custom-made parser, primarily because M4 is too flexible and invites abuse of our modularity rules.

3.1.4. Module Interfaces

The most fundamental change to the current structure is the use of macros for gaining access to a type outside of the module in which the type is defined. Interfaces provide access to a module's policy resources (i.e., to its privately declared types and attributes). All domains needing a particular access will use the same interface; therefore, the policy rules required for the access will be consistent across all users of the interface. Thus policy changes for access to a type require only a change in one place, rather than requiring changes to all the modules that use the type as is common in the sample policy.

In the future it may make sense to add interface and templates macro semantics to the base language. As macros, any time an interface is changed, all modules that use this interface must be recompiled. With interfaces in the language, modules will become decoupled on the loadable module level, in addition to the source level.

For improved clarity, interfaces follow clear naming conventions. In particular, the module name, or abbreviation, is prefixed to the interface name. This allows a policy writer to look a policy and easily see where all of the interface calls are. In addition, consistent verbs are used to describe the access, such as read, write, and delete.

Each interface contains two parts, the dependencies and the access. The dependencies are contained in a gen_require() macro. This macro contains the statements that would be placed in a require block for loadable modules. It lists all of the types and attributes used by the interface. If an object class for a user space object manager is used, such as DBUS or NSCD, the object class and required permissions must also be listed.

3.1.4. Module Template Interfaces

Template macros are the only macros that can declare types and attributes as part of an interface, i.e., the so-called derived types. The module defining the template macro is the only places where explicit access to/from the derived types can be defined. The only real difference between interface and template macros is that template macros use and/or create derived types on behalf of the caller. Both macros are "interfaces" to the defining module and can access the module's private types.

4. Module Example

Below is an excerpt from the BIND DNS server, from the Reference Policy.

4.2 bind.te

Example 1 shows the complete private policy for the named domain. The policy_module() macro contains the name and version of this module, which are used in loadable policy modules. The next nine lines contain the type declarations and transformation macro calls. The type for the domain and its entry point program are declared and transformed using the init_daemon_domain() macro, which is an interface from the init module. The remainder of the private policy contains the rules for the BIND service. For access to types not in this module, there are nine calls to interfaces of other modules complete the policy, for example logging_send_syslog_msg() is an interface from the logging module that allows bind to send messages to the system log daemon.

4.3. bind.if

Interfaces are defined in the module .if file. In Example 2, we see a use of the interface() macro to define access to entry the named domain (i.e., a domain transition interface). The caller would provide the type it wants to have the access to enter the named domain.

In addition to the actual interfaces, the interfaces file has its inline documentation encapsulated in XML, as also shown in Example 2. This provides documentation not only for policy writers to see when reading the file, but the XML can also be extracted from the policy and read by development and analysis tools.

4.4. bind.fc

The structure of a module's file contexts file has not been changed, as illustrated in Example 3. To facilitate the use of MLS and MCS policies, a gen_context() macro has been added. The regular type enforcement file context is specified as the first parameter. The second parameter provides the level of the file when a MLS policy is being used. An optional third parameter specifies the categories a file has when using a MCS policy. This allows a change between a MLS and non-MLS policy, for example, without modifying the policy.

5. Summary

The Reference Policy project has many goals, both security and functional goals, but the most important one is to make the policy simpler to understand and maintain. It uses layering, modularity, encapsulation, and abstraction to structure the policy, in addition to extensive documentation, to better meet these goals.

Interfaces, the most fundamental change to the policy, abstract the details creating higher-level concepts of access for policy writers. Modules encapsulate all of the implementation details for a particular subset of policy since all types and attributes are only directly usable from a particular module, and there are no global types or attributes. Finally, layers organize the modules into more meaningful groups.

6. References

[1] http://selinux.sourceforge.net and http://www.nsa.gov/selinux

[2] http://fedora.redhat.com/download

[3] http://hardened.gentoo.org/selinux

[4] http://serefpolicy.sourceforge.net

[5] Peter A. Loscocco, Stephen D. Smalley, Patrick A. Muckelbauer, and Ruth C. Taylor. "The Inevitability of Failure: The Flawed Assumption of Security in Modern Computing Environments." In Proceedings of the 21st National Information Systems Security Conference, pages 303-314, October 1998.

[6] Karl MacMillan. "Core Policy Management Infrastructure for SELinux." 2005 SELinux Symposium.

[7] James Morris. "A Brief Introduction to Multi-Category Security (MCS)." http://www.livejournal.com/users/james_morris/5583.html

```
policy_module(bind,1.1.0)

type named_t;
type named_exec_t;
init_daemon_domain(named_t,named_exec_t)

type named_cache_t;
files_type(named_cache_t)

type named_conf_t;
files_type(named_conf_t)

type named_zone_t;
files_type(named_zone_t)

allow named_t named_cache_t:file manage_file_perms;
allow named_t named_conf_t:file r_file_perms;
allow named_t named_zone_t:file r_file_perms;

kernel_read_system_state(named_t)
kernel_read_network_state(named_t)

corenet_non_ipsec_sendrecv(named_t)
corenet_udp_sendrecv_all_if(named_t)
corenet_udp_sendrecv_all_nodes(named_t)
corenet_udp_sendrecv_all_ports(named_t)
corenet_udp_bind_all_nodes(named_t)
corenet_udp_bind_dns_port(named_t)

logging_send_syslog_msg(named_t)
```

Example 1. Excerpt of BIND private policy file (bind.te)

```
## <summary>Berkeley internet name domain (DNS) server.</summary>

########################################
## <summary>
##     Execute bind in the named domain.
## </summary>
## <param name="domain">
##     Domain allowed access.
## </param>
#
interface(`bind_domtrans',`
       gen_require(`
              type named_t, named_exec_t;
       ')

       domain_auto_trans($1,named_exec_t,named_t)

       allow $1 named_t:fd use;
       allow named_t $1:fd use;
       allow named_t $1:fifo_file rw_file_perms;
       allow named_t $1:process sigchld;
')
```

Example 2. Excerpt of BIND interface file (bind.if)

```
/etc/rndc.*                --     gen_context(system_u:object_r:named_conf_t,s0)

/usr/sbin/named            --     gen_context(system_u:object_r:named_exec_t,s0)

/var/named(/.*)?                  gen_context(system_u:object_r:named_zone_t,s0)
/var/named/slaves(/.*)?           gen_context(system_u:object_r:named_cache_t,s0)
/var/named/data(/.*)?             gen_context(system_u:object_r:named_cache_t,s0)
```

Example 3. Excerpt of BIND file contexts file (bind.fc)