# SENG: An Enhanced Policy Language for SELinux

Paul Kuliniewicz
*Purdue University*

## Abstract

Most of the statements in the current SELinux policy language operate directly on features of the underlying access control model. Because the policy for a typical Linux system contains a large number of distinct types, a realistic policy will be large and unwieldy. Current practice is to manage this complexity through preprocessor macros, using them to encapsulate portions of the policy and allow for their reuse. However, such macros inhibit the use of tools to analyze the policy, as these macros must first undergo expansion. As a result, the policy being analyzed is not in the same form as the one originally written, complicating the task of using the analysis results to improve the policy.

This paper introduces SENG, an experimental alternative language for writing SELinux policies. SENG builds upon the existing policy language by adding abstractions with well-defined semantics, with the goal of eliminating the need for macros to write a maintainable policy. To facilitate testing and adoption, a compiler has been implemented to transform SENG policies into the existing policy language, allowing SENG to be used without requiring changes to the existing policy framework.

## 1 Motivation

One of the major factors preventing widespread adoption of SELinux is the preceived difficulty of writing policies. A typical Linux environment has hundreds of resources and applications that need both access to and protection from each other, necessitating a correspondingly large policy. For example, when including all optional components, version 1.26 of the monolithic example policy [3] specifies 2,024 types, 66,676 access vector rules, and 2,095 type transitions. If written solely in the currently available language, the sheer size of the policy would render it unmaintainable.

To sidestep this issue, the example policy relies upon the m4 macro processor. Macros are used to express the intended policy more succinctly, hiding implementation details and providing higher-level abstractions over the rules in the underlying language.

However, the pervasive use of m4 inhibits the ability of automated tools to analyze a policy. Because m4 is merely a text-based processor, macros are unconstrained by the semantics of the underlying policy language. The use of additional m4 features such as conditionals, regular expressions, and recursion makes it infeasible for a tool to understand a policy without first expanding all macros. Macro expansion strips away all higher-level abstractions, leaving behind only a series of individual statements. Because there is no clear connection between the original policy and the result of expansion, it can be difficult for the policy writer to apply an analysis tool's results when modifying the policy.

SENG is an experimental language that takes a different approach to managing the complexity of a large SELinux policy. Instead of relying on macros, SENG extends the language with native support for certain higher-level abstractions, each designed to obviate the need for macros. Since, unlike macros, SENG's abstractions have well-defined semantics, they are readily amenable to automated analysis, thus permitting analysis tools to work directly upon and output results in terms of the policy as originally written.

## 2 New Features in SENG

SENG builds upon the existing policy language by adding new higher-level abstractions on top of the current set of statements, each designed to eliminate a shortcoming in the language currently addressed through m4.

### 2.1 Type, class, and permission sets

The existing language uses attributes to group together related types, but relies on macros to do the same for

```
permset ra_dir_perms { read getattr lock search ioctl add_name write };
permset ra_file_perms { ioctl read getattr lock append };

resource var_log { append append_dir };
type ANYTYPE.log_t { file_type sysadmfile logfile };

permission var_log append ($dom) {
    allow $dom var_log_t:dir ra_dir_perms;
    allow $dom $dom.log_t:file { create ra_file_perms };
    type_transition $dom var_log_t $dom.log_t:file;
};

allow daemon_t var_log append;
```

Figure 1: Example of permission sets, abstract resources, and type templates in SENG.

classes and permissions. SENG introduces *class sets* and *permission sets* to replace the need for such macros. For the sake of regularity, SENG also refers to attributes as *type sets*. A set can be used in any context where its members could explicitly be listed individually.

Figure 1 presents an example of defining and using permission sets. Type and class sets are used similarly.

## 2.2 Abstract resources

Frequently, a single conceptual resource is composed of multiple types, classes, and permissions, such that a single access vector rule does not suffice to grant access. SENG addresses this though *abstract resources* definable by the policy writer. Like classes, abstract resources offer one or more permissions. A domain can be granted access to a permission over an abstract resource through an `allow` rule, similarly to how it can be granted a permission over a type:class pair.

Each permission on an abstract resource has one or more rules associated with it, each parameterized over a single variable. Whenever a domain is granted the permission, each rule is evaluated with the variable replaced with the name of the domain. Thus, granting access to an abstract resource's permission is equivalent to specifying a series of rules involving that domain.

Figure 1 presents an example of defining and using abstract resources.

## 2.3 Abstract permissions

Granting a domain multiple permissions over a type cannot be accomplished with a single access vector rule if the permissions are defined for different classes. SENG provides *abstract permissions* to handle this case without resorting to macros. A domain can be granted an abstract permission over a type using an `allow` rule by omitting the class list.

Like abstract resources, each has a series of associated rules that are evaluated when it is used in an `allow` rule. Unlike abstract resources, these rules are parameterized over both the domain and type specified in the `allow` rule.

Figure 2 presents an example of defining and using abstract permissions.

## 2.4 Templates

It is often necessary to create a new type associated with an existing type or role. For example, domains having write access to a shared directory (such as `/tmp` or `/var/log`) often need a unique type for the files they place there, distinct from types used by other domains. In the existing language, macros declare new types with names created by prepending or appending strings to the macro's arguments.

SENG uses a template system to eliminate the need for such name mangling. Templates can generate new types and booleans, though for simplicity, the remainder of this section will focus on type templates.

A *type template* is defined similarly to an ordinary type, except that the name must begin with `ANYROLE` or `ANYTYPE`. A template generates a new name by combining the name of an existing role or type (the prefix), respectively, with the remainder of the name used in the template (the suffix). Templates are automatically instantiated at compile time whenever a type derivable from it is used. Recursive template instantiation, in which the prefix is itself a type generated by the template, is forbidden, to prevent an unbounded number of types from being created.

The prefix resolution operator, when applied to the name of a type generated by a template, returns the pre-

```
permission create_dir_file ($dom, $typ) {
    allow $dom $typ:dir create_dir_perms;
    allow $dom $typ:file create_file_perms;
    allow $dom $typ:lnk_file create_lnk_perms;
};

allow domain_t type_t create_dir_file;
```

Figure 2: Example of abstract permissions in SENG.

```
resource home { use };
type ANYROLE.home_t { file_type sysadmfile polymember };
type ANYROLE.app_t { domain };

permission home use ($dom) {
    allow $dom prefix($dom).home_t create_dir_file;
};

allow user_r.app_t home use;
```

Figure 3: Example of the prefix resolution operator in SENG.

fix used to instantiate it. The result of the operator can be used directly in a rule or used to build another name. This allows rules to refer to the role or type used to create an instantiated type, or a type created by another template and that prefix, without relying upon naming conventions.

Figure 1 presents an example of using a type template. Figure 3 demonstrates the use of the prefix resolution operator.

## 2.5 Custom type transitions

In practice, a type transition needs several additional permissions to be granted to allow the transition to take place. For example, a domain transition is useless unless the original domain can execute the related executable file. As a result, the example policy rarely uses `type_transition` rules directly, but instead invokes macros that also provide the supporting grants.

SENG introduces *custom type transitions* to allow the policy writer to specify rules that should be evaluated along with a type transition. These rules are parameterized over the original domain, related type, target type, and optionally target class. The target class can also be a particular class, if the rules should only be applied when the transition is for that class. The name of a custom type transition to use can be specified at the end of the `type_transition` rule.

Figure 4 presents an example of using custom type transitions.

## 2.6 Tunables

The monolithic example policy uses m4 conditionals to optionally include portions of policy, depending on factors such as the target Linux distribution and which optional components are being included. SENG provides *tunables* to provide this compile-time configurability of policy. The presence of a tunable can be checked for in an `ifdef` statement. If the tunable has been defined, the rules in the then block are included; otherwise the rules in the optional else block are used.

Tunables are not to be confused with booleans. Tunables operate solely at compile time, whereas booleans can be changed on a live system.

Figure 5 illustrates an example of using tunables.

## 3 Examples

The following examples demonstrate the usage of many of SENG's features, and are based on a translation of the example monolithic policy into SENG. Note that the line numbers are not part of the language.

## 3.1 Example 1

Figure 1 is an example of permission sets, abstract resources, and type templates.

Lines 1 and 2 define two permission sets, which are then used in `allow` rules in lines 8 and 9. As seen in line 9, once defined, permissions and permission sets can be used interchangeably.

```
 1: trans file_auto ($dom, $via_typ, $to_typ:$to_class) {
 2:     allow $dom $via_typ:dir rw_dir_perms;
 3:     type_transition $dom $via_typ $to_typ:$to_class;
 4: };
 5: trans file_auto ($dom, $via_typ, $to_typ:file) {
 6:     allow $dom $to_typ:file create_file_perms;
 7: };
 8: trans file_auto ($dom, $via_typ, $to_typ:lnk_file) {
 9:     allow $dom $to_typ:lnk_file create_lnk_perms;
10: };
11:
12: type_transition domain_t type_t foo_t:file file_auto;
```

Figure 4: Example of custom transitions in SENG.

```
tunable distro_debian;

ifdef (distro_redhat) {
    allow redhat_domain_t redhat_type_t:file r_file_perms;
};
ifdef (distro_debian) {
    allow debian_domain_t debian_type_t:file r_file_perms;
};
```

Figure 5: Example of tunables in SENG.

Line 4 defines a new abstract resource with two permissions, and the block on lines 7 through 11 associates rules with one of its permissions. Each rule is parameterized over the variable `$dom`.

The `allow` rule on line 13 grants a domain one of the abstract resource's privileges; note how the resource name is used in place of a type:class pair. Lines 8 through 10 are evaluated with `daemon_t` substituted in for `$dom`.

Line 5 defines a type template. When lines 9 and 10 are evaluated for `daemon_t`, the template is used to create the new type `daemon_t.log_t`. Note that this type never appears explicitly in the example, but rather is formed by the result of variable substitution.

## 3.2 Example 2

Figure 2 presents an example of using abstract permissions in SENG.

The block on lines 1 through 5 defines a new abstract permission and associates rules with it.

The `allow` rule on line 7 uses the abstract permission. Note that the class name is omitted in the rule. Lines 2 through 4 are evaluated with `domain_t` substituted for `$dom` and `type_t` substituted for `$typ`.

## 3.3 Example 3

Figure 3 presents an example of using the prefix resolution operator.

The `allow` rule in line 9 causes line 6 to be evaluated with `user_r.app_t`, derived from the template in line 2, substituted in for `$dom`. Because the prefix resolution operator returns `user_r`, the rule in line 6 has the effect of granting `user_r.app_t` permissions over `user_r.home_t`.

## 3.4 Example 4

Figure 4 presents an example of using custom transitions in SENG.

The block on lines 1 through 4 defines a custom transition that applies whenever a `file_auto` transition is specified. The block on lines 5 through 7 specifies additional rules for a `file_auto` transition when the target class is `file`, as does the block on lines 8 through 10 when the target class is `lnk_file`.

The `type_transition` rule on line 12 requests a `file_auto` transition. The rules on line 2 and 3 are evaluated (with the appropriate variable substitutions) because they apply for all such transitions. The rule on line 6 is similarly evaluated because the transition is for a `file`, but the rule on line 9 is not.

Note that in SENG syntax, the target class is paired with the target type and not the related type in a `type_transition` rule.

## 3.5 Example 5

Figure 5 presents an example of using tunables in SENG.

At compile time, line 4 is ignored because no tunable named `distro_redhat` has been defined. However, line 7 is included because the tunable `distro_debian` was defined on line 1.

## 4 Future Work

Much of SENG's design has been motivated by shortcomings in the language used by versions 1.26 and earlier of the example monolithic SELinux policy. As a result, SENG does not consider the implications of current work on the reference policy [4], which also involves extensive use of m4, or advanced features such as MLS support. Work is needed to determine to what degree SENG is appropriate for these efforts and to identify any refinements to SENG that may be necessary.

Currently, the semantics of SENG's features is defined solely by the proof-of-concept implementation of a compiler that transforms SENG policies into the existing language. A formalized definition of SENG semantics is needed to clearly express the language's behavior.

No analysis tools yet exist for SENG, even though SENG's primary motivation is to facilitate their use. Proof-of-concept analysis tools are needed to demonstrate the practical advantages of SENG over the existing language.

## 5 Related Work

Jaeger et. al. [1] and Zanin and Mancini [7] have done work on analyzing SELinux policies to evaluate whether they satisfy particular goals. Tresys Technology's SETools suite [5] and MITRE's SLAT [2] are analysis tools that use the existing policy language. The common drawback to all these tools and methods is that they only operate after macro expansion has taken place, and so can only state results in terms of the expanded form, rather than that created by the policy writer.

The SELinux reference policy [4] splits a policy into individual modules. Each module hides its implementation details from other modules and exposes only an abstract interface defined by m4 macros. It may be possible for SENG features such as abstract resources and abstract permissions to replace these interfaces, providing modularization without the need for m4.

Another effort at designing a new policy language is Tresys's SEFramework project [6]. Its goal is to provide a simplified language usable by application developers, whereas SENG targets policy writers. SEFramework is an entirely new language for policies, whereas SENG is primarily an extension of the existing language, although both can be compiled into the existing language. SEFramework's fobjects and resources play similar roles to SENG's abstract permissions and resources: both simplify granting permissions to domains by grouping related rules into a single interface, although the two language's approaches have different semantics. Finally, although the structure of names used in SENG and SEFramework are superficially similar, in SEFramework they establish a hierarchy wherein a child's permissions are constrained by those held by the parent, whereas SENG only uses the segmentation for name substitution.

## 6 Conclusion

SENG is an experimental language for writing SELinux policies that replaces the need for m4 macros by adding support for well-defined, higher-level abstractions. By doing so, SENG aims to be much more suitable for analysis tools than the existing policy language, thus making policies easier to write and maintain. Although SENG is currently in an early stage of development, it offers significant promise of making SELinux a more usable option for securing a Linux system.

## 7 Acknowledgements

The author would like to thank Ninghui Li and Elisa Bertino for their guidance in the semester projects this work grew out of, and the anonymous reviewers for their comments and suggestions.

## References

[1] JAEGER, T., SAILER, R., AND ZHANG, X. Analyzing Integrity Protection in the SELinux Example Policy. *Proceedings of the 12th USENIX Security Symposium* (Aug 2003).

[2] MITRE. http://simp.mitre.org/selinux/.

[3] NSA. http://www.nsa.gov/selinux/code/download5.cfm.

[4] TRESYS TECHNOLOGY. http://serefpolicy.sourceforget.net/.

[5] TRESYS TECHNOLOGY. http://www.tresys.com/selinux/selinux_policy_tools.html.

[6] WILSON, A. SEFramework: A New Policy Development Framework and Tool to Support Security Engineering. *SELinux Symposium* (March 2005).

[7] ZANIN, G., AND MANCINI, L. V. Towards a Formal Model for Security Policies Specification and Validation in the SELinux System. *SACMAT* (Jun 2004).