SELinux Policy Semantics

Joe Hurd, Magnus Carlsson and Peter White
Galois, Inc.

{joe,magnus,peter}@galois.com

14 February 2008

1 Introduction

A raw SELinux policy is compiled to binary form using the `checkpolicy` program, and this is used by the SELinux security server to decide whether to permit or deny system actions. This document presents a semantics of policies in terms of permitted actions.

This is a security-centric view of SELinux policies, which examines the set of all possible permitted actions that can occur on an SELinux system. Such a conservative view is useful for analyzing a system that may be host to some malicious processes, fully aware of their SELinux permissions and exercising them in cooperation to achieve an attack on the system.

2 SELinux

SELinux [2, 3] *objects* are divided into a set C of *classes*; there is no built-in distinction between passive classes such as file and dir, and the active class $\mathsf{process}$. The term *subject* is used of an object when it is known to be of class $\mathsf{process}$. Write o_c for the class of object o.

In addition to its class, each object o is labelled with a *security context* o_s. A security context is a triple (u, r, t) (usually written $u{:}r{:}t$), consisting of a *user* $u \in U$, a *role* $r \in R$ and a *type* $t \in T$. There is a distinguished role $\mathsf{object\_r} \in R$, intended for passive classes of objects. For shorthand we define

$$S \equiv U \times R \times T \ ,$$

the set of all possible security contexts (although not all of them will be valid—see Section 3.2).

For each class c there is a set P_c of *permissions* representing actions that make sense to perform on objects of class c. For shorthand we define

$$CP \equiv \{(c, p) \mid c \in C \wedge p \in P_c\} ,$$

the set of all legal class permission pairs.

One part of an SELinux policy is a definition of a set B of boolean variables. For example, the policy statement

```
bool b true
```

declares the boolean variable b and initializes it to $\top$. Exactly which actions are permitted is parameterized by the subset of variables $A \subseteq B$ that have the value $\top$ at the time of the request. In effect an SELinux policy is actually a family of policies

$$\mathcal{P}(B) \to \text{Policy} .$$

In the remainder of this document a *policy* means an SELinux policy together with a particular setting of the boolean variables.

3 Semantics

At a high level, there are only two kinds of security relevant action that may be carried out on an SELinux system:

1. Subject s performs an action p on object o. This kind of action is called an *access*, and the set of accesses permitted by the policy is modelled by the Access relation:

   $$\begin{aligned} \mathsf{Access} : S \times S \times CP &\to \mathbb{B} \\ (s_s, o_s, (o_c, p)) &\mapsto \begin{cases} \top & \text{if permitted by the policy} \\ \bot & \text{otherwise} \end{cases} \end{aligned}$$

   The SELinux policy statement

   ```
   allow s_s o_s : o_c p
   ```

   is used to permit the above access (although other conditions must also be met—see Section 3.3 for details).

2. Subject s creates a new object n in the context of *related object* r. This kind of action is called a *create*, and the set of creates permitted by the policy is modelled by the Create relation:

$$\begin{aligned}\mathsf{Create} : S \times S \times S \times C &\to \mathbb{B} \\ (s_s, r_s, n_s, n_c) &\mapsto \begin{cases} \top & \text{if permitted by the policy} \\ \bot & \text{otherwise} \end{cases}\end{aligned}$$

   Section 3.4 examines the conditions that must be met for a create action to be permitted.

Note that this is a simplification of a real SELinux system, in at least two important ways:

- It does not take into account actions that have a side-effect of changing the policy in force, such as loading a new policy, (persistently) changing a boolean variable, or rebooting the system.
- In addition to specifying the permitted actions, SELinux policies also specify which attempted actions are audited. By default all denied actions are audited and no permitted actions are, but this is fully customizable in the policy.

3.1 Default Transitions

There are two kinds of *transition* that can occur on an SELinux system:[1]

1. A *domain transition* occurs when a subject s uses the `execve` system call on an executable file with security context r_s, resulting in a new process with security context n_s.
2. An *object transition* occurs when a subject s creates an object of class c in the context of related object r, when the newly created object will be labelled with security context n_s.

In addition to specifying the permitted actions, policies also specify *default transitions*, which use the security contexts of the subject and related object together with the class of the new object to determine its security context n_s. Both *default domain transitions* and *default object transitions* are specified in the policy using the statement

[1]From page 117 of *SELinux by Example* [2].

`type_transition` s_s r_s `:` c n_s

where the class c is process for a default domain transition.

Default transitions are modelled by the DefaultTransition function:

$$\begin{aligned}\mathsf{DefaultTransition} : S \times S \times C &\to S \\ (s_s, r_s, c) &\mapsto n_s\end{aligned}$$

Before any `type_transition` policy statements have been processed, the DefaultTransition function is initialized to

$$\mathsf{DefaultTransition}\ (s_s, r_s, c) = \begin{cases} s_s & \text{if } c = \mathsf{process} \\ r_s & \text{otherwise} \end{cases}$$

`type_transition` policy statements override the initial DefaultTransition function, and it is an error for two `type_transition` policy statements to conflict.

Default transitions allow the policy author to specify sensible defaults so that applications can operate without even being aware that they are running on SELinux. However, they explicitly do not grant permission to perform the domain or object transition: the default transition just specifies the new security context for the resulting object. Successfully performing the transition usually requires many permissions to be granted (see Section 3.4 for more details).

It is possible to give a process permission to change the default transition in the policy and specify the new security context. The setexec permission gives a process the ability to specify a new security context following a domain transition, and the setfscreate, setsockcreate and setkeycreate permissions are used for different classes of object transitions. However, it is important to note that even if a process has permission to specify a new security context, still all of the standard conditions must be satisfied for the action to succeed (in particular, the specified new security context must be valid).

If any of the above permissions are granted to a process, then the corresponding default transition in the policy becomes irrelevant from a security perspective. As motivated in the Introduction, this semantics looks solely at the set of possible actions, and if a process has permission to change the default transition then the initial value specified in the policy is irrelevant.

3.2 Valid Security Contexts

The policy specifies the subset of possible security contexts which are valid. This is modelled using the Valid predicate:

$$\begin{aligned}\mathsf{Valid} : S &\to \mathbb{B} \\ (u,r,t) &\mapsto \mathsf{UserRole}\ (u,r) \wedge \mathsf{RoleType}\ (r,t)\end{aligned}$$

The $\mathsf{UserRole}$ and $\mathsf{RoleType}$ relations come directly from the policy. For example, the policy statements

```
user u roles { r };
role r types { t };
```

add (u,r) to the $\mathsf{UserRole}$ relation, and (r,t) to the $\mathsf{RoleType}$ relation.

3.3 Permitted Accesses

Following Guttman et. al. [1], the Access relation is reduced to a conjunction of simpler relations:

$$\begin{aligned}&\mathsf{Access}\ ((u_s,r_s,t_s),(u_o,r_o,t_o),(c,p)) \equiv \\ &\quad \mathsf{Valid}\ (u_s,r_s,t_s) \wedge \\ &\quad \mathsf{Valid}\ (u_o,r_o,t_o) \wedge \\ &\quad \mathsf{AllowTypes}\ (t_s,t_o,c,p) \wedge \\ &\quad \mathsf{Constraints}\ ((u_s,r_s,t_s),(u_o,r_o,t_o),(c,p))\ .\end{aligned}$$

The Valid conjuncts ensure the subject and object have valid security contexts. In practice this check can be skipped, because every object on an SELinux system must have a valid security context.

The $\mathsf{AllowTypes}$ relation is the main mechanism of type enforcement. It is initialized to the empty relation, and allowed type accesses are explicitly added to it by `allow` statements in the policy.

The $\mathsf{Constraints}$ relation further cuts down the set of permitted actions. In contrast with the $\mathsf{AllowTypes}$ relation, the $\mathsf{Constraints}$ relation is initialized to the total relation, and is reduced by `constrain` statements in the policy.

To more closely match the SELinux implementation we also define a function to compute an *access vector*:

$$\begin{aligned}\mathsf{AccessVector} : S \times S \times (c : C) &\to \mathcal{P}(P_c) \\ (s_s, o_s, c) &\mapsto \{p \in P_c \mid \mathsf{Access}\ (s_s, o_s, (c,p))\}\ .\end{aligned}$$

3.4 Permitted Creates

For each class c of newly created object, there is a built-in class r for its related object. For example, a new process has as its related object the executable file, and a new file is related to its containing dir. This knowledge is modelled by the RelatedClass function:

$$\begin{array}{lll} \mathsf{RelatedClass} : C & \to & C \\ c & \mapsto & r \\ \mathsf{process} & \mapsto & \mathsf{file} \\ \mathsf{file} & \mapsto & \mathsf{dir} \end{array}$$

When a subject s makes a request to create a new object of class c in the context of related object r, the SELinux system processes this request in two parts. Firstly, the default transition rules are applied to determine the security context n of the new object.[2] Secondly, the SELinux system effectively transforms the create request into a number of access requests, all of which must succeed for the transition to succeed. For example, to create a new file the following accesses must be permitted:

$$\begin{array}{l} \mathsf{Create}\ (s_s, r_s, n_s, \mathsf{file}) \iff \\ \quad \mathsf{Access}\ (s_s, r_s, (\mathsf{dir}, \mathsf{search})) \wedge \\ \quad \mathsf{Access}\ (s_s, r_s, (\mathsf{dir}, \mathsf{add\_name})) \wedge \\ \quad \mathsf{Access}\ (s_s, r_s, (\mathsf{dir}, \mathsf{write})) \wedge \\ \quad \mathsf{Access}\ (s_s, n_s, (\mathsf{file}, \mathsf{create})) \end{array}$$

As a more complicated example, here are some of the necessary permissions for a process to make the system call `execve` to execute a file:

$$\begin{array}{l} \mathsf{Create}\ (s_s, r_s, n_s, \mathsf{process}) \Rightarrow \\ \quad \mathsf{Access}\ (s_s, r_s, (\mathsf{file}, \mathsf{execute})) \wedge \\ \quad \mathsf{if}\ n_s = s_s\ \mathsf{then} \\ \quad\quad \mathsf{Access}\ (s_s, r_s, (\mathsf{file}, \mathsf{execute\_no\_trans})) \\ \quad \mathsf{else} \\ \quad\quad \mathsf{Access}\ (n_s, r_s, (\mathsf{file}, \mathsf{entrypoint})) \wedge \\ \quad\quad \mathsf{Access}\ (s_s, n_s, (\mathsf{process}, \mathsf{transition})) \end{array}$$

In all situations, the process must have execute permission on the executable file. If the new process would have the same security context (as determined

[2]If the subject s has permission to override the default transition rules, then it may specify the new security context n.

by the default transition rules defined in Section 3.1), then the process must also have $\mathsf{execute\_no\_trans}$ permission on the file. If the new security context is different, then the process must have $\mathsf{entrypoint}$ permission on the file, and $\mathsf{transition}$ permission to the new process.

It is conjectured that $\mathsf{Create}\ (s_s, r_s, n_s, c)$ is completely determined by the access vectors

$$\begin{array}{l} \mathsf{AccessVector}\ (s_s, r_s, \mathsf{RelatedClass}(c)) \\ \mathsf{AccessVector}\ (n_s, r_s, \mathsf{RelatedClass}(c)) \\ \mathsf{AccessVector}\ (s_s, n_s, c) \end{array}$$

but discovering the precise transformation rules is postponed to a later phase of the project.

4 Haskell Implementation

The Haskell implementation of this semantics processes SELinux policy files in three phases:

1. Parsing the concrete syntax in the policy file into an abstract syntax tree (the `Lexer` and `Parser` modules).
2. Performing a scope analysis of the abstract syntax tree, resolving optional blocks and constructing the symbol table (the `Symbol` module).
3. Building the authorization relations in Section 3.3 by expanding attributes and converting special syntax for types and permissions into explicit sets (the `Authorize` module).

The only deviation from the semantics as presented is that the $\mathsf{AllowTypes}$ relation is conditional on the settings of the boolean variables. This allows boolean variables to be changed dynamically in the Haskell implementation of the semantics, just as in a running SELinux system.

Here is the result of the scope analysis and authorization relation building phases of the Haskell implementation, run on the SELinux Reference Policy (version 20070629):

```
Built the symbol table for policy.conf:
classes: 61
--permissions: 879
sids: 27
```

```
commons: 3
roles: 6
types/attributes: 2629
--types: 2380
--aliases: 98
--attributes: 151
users: 5
bools: 79

Built the authorization relation for policy.conf:
attributes: 124 (containing 2331 unique types)
role type relation: 5 roles <--> 490 types
user role relation: 5 users <--> 4 roles
boolean variables: 79
conditionals: 1061
constraints: 77
```

4.1 Validation

The principal way of validating the Haskell implementation is by automatically comparing access vectors with the `checkpolicy` program. Here is an example policy test file:

```
#ACCESS u:r:t u:r:t c
#BOOL b false
#ACCESS u:r:t u:r:t c
class c
sid policy_grammar_requires_at_least_one_sid
class c { p }
type t;
bool b true;
role r types { t };
if (not b) { allow t t : c p; }
user u roles { r };
sid policy_grammar_requires_at_least_one_sid u:r:t
```

This a legal SELinux policy file, with the test commands contained in the comments. In this test the Haskell implementation is asked to compute

$$\mathsf{AccessVector}\ ((u, r, t), (u, r, t), c)\ ,$$

and then to set the boolean variable b to $\perp$ and compute the same access vector again.

Here is the result of the test:

```
ACCESS ( u:r:t u:r:t c )... { }
BOOL ( b := False )... ok
ACCESS ( u:r:t u:r:t c )... { p }
```

The first access vector is the empty set of permissions, and the second is the set containing the permission p. This is confirmed by automatically running the same test using the `checkpolicy` program.

At the time of writing there are 38 policy test files, constructed by hand to gain understanding of corner cases, discover ambiguities and improve code coverage.

Another way of validating the semantics is to make use of `neverallow` statements in the policy. For example, loading the following policy into `checkpolicy` causes an error because the `allow` and `neverallow` statements conflict:

```
class c
sid policy_grammar_requires_at_least_one_sid
class c { p }
type t;
bool b true;
role r types { t };
allow t t : c p;
neverallow t t : c p;
user u roles { r };
constrain c p (t1 != t2);
sid policy_grammar_requires_at_least_one_sid u:r:t
```

This is an interesting example because it demonstrates that `neverallow` pays attention to the syntax of policies rather than their semantics. In fact the `allow` statement is redundant because the `constrain` statement completely counteracts its effect, but its presence is sufficient to trigger a conflict.

4.2 Potential Ambiguities

Here is another example policy test file, illustrating a potential ambiguity in the semantics of SELinux policies:

```
#ACCESS u:r:t u:r:t c
class c
sid policy_grammar_requires_at_least_one_sid
class c { p }
type t;
allow t { self t -t } : c p;
role r types { t };
user u roles { r };
sid policy_grammar_requires_at_least_one_sid u:r:t
```

The ambiguity stems from the complex allow command:

```
allow t { self t -t } : c p;
```

There are two plausible interpretations of the target types in this command that result in different access vectors:

1. Replace `self` by t (the source type) to make the target set

   ```
   { t t -t }
   ```

   which simplifies to the empty set $\{\}$. This makes the access vector $\{\}$.

2. Leave `self` untouched and simplify the types in the target set, which results in the target set

   ```
   { self }
   ```

   This makes the final access vector $\{p\}$.

Using the `checkpolicy` program it can be confirmed that interpretation 2 is the correct choice in this case. However, to ensure there are no ambiguities in the low-level semantics the Haskell implementation rejects the policy with the error message

```
UNSUPPORTED: self cannot occur with negative types
```

4.3 Supported Policy Language

As the previous section shows, there are some aspects of the SELinux policy language with confusing semantics. Here is the complete list of unsupported language statements:

1. `self` cannot occur with negative types.
2. `self` cannot occur negatively.
3. `self` cannot occur inside a tilde operator.
4. `auditdeny` rules are unsupported.
5. When subtracting types from a set using the $\{A \; - B\}$ notation, the types in B must all be in A.
6. `dominance` statements are unsupported.

In addition, the current tool does not support the SELinux MLS permissions. However, the entire SELinux Reference Policy lies within the supported policy language.

Acknowledgments

Comments from Tresys Technology greatly improved this paper.

References

[1] Joshua D. Guttman, Amy L. Herzog, John D. Ramsdell, and Clement W. Skorupka. Verifying information flow goals in Security-Enhanced Linux. *Journal of Computer Security*, 13(1):115–134, 2005.

[2] Frank Mayer, Karl MacMillan, and David Caplan. *SELinux by Example.* Open Source Software Development Series. Prentice Hall, 2007.

[3] Bill McCarty. *SELinux: NSA's Open Source Security Enhanced Linux.* O'Reilly, 2005.