Open Source Software Development Series

SELinux by Example

Using Security Enhanced Linux

FRANK MAYER
KARL MACMILLAN
DAVID CAPLAN

SELinux by Example

`bin_t`. For example, `user_t` would not be able to write files of type `bin_t`.

`Allow` rules, like all AV rules, are cumulative and the actual access allowed for a given subject-target-class key is the union of all the `allow` rules that refer to that key. For example, these two sets of rules are equivalent:

```
# These two rules...
allow user_t bin_t : file read;
allow user_t bin_t : file write;

# are equivalent to (and redundant with) this single rule.
allow user_t bin_t : file { read write };
```

5.3.3 Audit Rules

SELinux has extensive facilities for logging, or auditing, access attempts that are either allowed or denied by the policy. The audit messages, often called "AVC messages," give detailed information about an access attempt, including whether it was allowed or denied, the security context of the source and target, and other details about the resources involved in the access attempt. The messages, which are similar to other kernel messages and are usually stored in log files under `/var/log`, are an indispensable tool for policy development, system administration, and system monitoring. In this chapter, we examine the policy features that enable us to configure which access attempts will generate audit messages. Part III provides more information about how to use audit messages to debug and understand policies.

By default, SELinux does *not* record any access checks that are allowed but records *all* access checks that are denied. These defaults are not surprising; on most systems, thousands of accesses per second are allowed, but few accesses are denied. The allowed accesses are, by the fact that they were allowed, expected and usually do not require auditing. The denied accesses are usually, but not always, unexpected, and auditing them helps an administrator to monitor for policy bugs and/or possible intrusion attempts. The policy language allows us to override portions of these defaults to suppress audit messages for expected access denials and to generate audit messages for access attempts that were allowed.

SELinux provides two AV rules that allow us to control which access attempts are audited: `dontaudit` and `auditallow`. These two rules are the policy mechanism that enable us to change these auditing defaults. The `dontaudit` rule is the most commonly used. It specifies which access denials should `not` be audited, overriding the SELinux default behavior to audit all access denials.

> WARNING Access denials are audited only if the denial was made by SELinux. Recall from Chapter 3 that LSM module hook functions are usually called only if the access passes the standard Linux discretionary access control checks. This means that if an access was denied because of the standard Linux access checks, SELinux is not even aware of the access attempt and cannot generate an audit message. If you need to audit all denied accesses regardless of why the access is denied, you must directly use the kernel audit system included in the 2.6.x series of kernels. See the man pages for *auditd*(8) and *auditctl*(8).

For example, consider this:

```
dontaudit httpd_t etc_t : dir search;
```

This rule specifies that when processes of type `httpd_t` are denied `search` permission on directories of type `etc_t`, the denial should not be audited, overriding the default behavior. We might write this rule if processes with type `httpd_t` attempt to search directories of type `etc_t` (presumably `/etc/`) but function properly when this access is not granted. You will find Linux/UNIX applications often exhibit this type of behavior; that is, they attempt access they do not need yet work fine when the access is denied.

The `dontaudit` rule is useful when we want to mask audit denial messages that are expected, usually due to expected behavior of an application. The `dontaudit` rule allows us to avoid granting unnecessary access (because the application works without the access, it is unnecessary by any definition) without a large number of expected audit messages filling the system logs. As we said, this type of behavior is all too common.

Auditdeny Rule

Earlier versions of SELinux supported an `auditdeny` rule. These rules were used for a similar purpose to the dontaudit rules. Although still supported by the policy language, an `auditdeny` rule is seldom, if ever, seen in policies. The rule is deprecated, and we suggest you do not attempt to use it. The `dontaudit` rule, coupled with the default behavior of recording all access denials, is the desired method for controlling access denial auditing.

The other audit rule, `auditallow`, allows us to control the auditing of allowed access attempts. Unlike denied access, allowed access is not recorded by default. For example, let's look at the following rule:

```
auditallow domain shadow_t : file write;
```

This rules specifies that when a process with a type that has the `domain` attribute successfully obtains `write` access to files of type `shadow_t`, the allowed access is audited. The `auditallow` rule is useful to audit accesses that are an important security event. Examples of access that are likely to have an `auditallow` rule include writing to the shadow password file (as the above rule does) or reloading a new policy into the kernel.

Remember, audit rules let us override the default auditing settings. The `allow` rule specifies which access is allowed. The `auditallow` rule does not allow access; it enables only auditing of allowed permissions.

> NOTE Auditing is different in permissive and enforcing modes. When running in enforcing mode, audit messages are generated every time there is an allowed or denied access that the policy states should be audited up to a rate limit (this can be set with *auditctl*(8)). In permissive mode, only the first access attempt is logged until the next policy load or toggle of the enforcing mode. Permissive mode is most often used for policy development, and this auditing mode helps reduce the size of the log.

5.3.4 Neverallow Rules

The final AV rule is the `neverallow` rule. We use this rule to state invariant properties specifying certain accesses that may never be permitted by an `allow` rule. You might wonder why this rule exists, because access is denied by default. The reason is to aid policy writing by noting certain undesired permissions, thereby preventing the accidental inclusion of these permissions in our policy. Recall that an SELinux policy is likely to contain tens of thousands of rules. It is quite possible to accidentally grant an access we did not want to grant. The `neverallow` rule helps prevent this situation. For example, consider this rule:

```
neverallow user_t shadow_t : file write;
```

This `neverallow` rule would prevent us from adding a rule to the policy that allows `user_t` to write to files of type `shadow_t` by generating a compile error. This rule does not remove access, it just generates compile errors. The `neverallow` rule is to state important properties about our policy before we start writing `allow` rules. The `neverallow` rules prevent us from inadvertently including permissions that we did not intend.

The `neverallow` rule supports some additional syntax that the other AV rules do not. In particular, the source and target type lists in `neverallow` rules can contain the wildcard (`*`) and complement (`~`) operators. These operators work just as they do for permission lists in the rest of the AV rules (see the section "Special Permission Operators for AV Rules," earlier in this chapter).

For example, look at the following rule:

```
neverallow * domain : dir ~{ read getattr };
```

This rule states that no `allow` rule may grant any type any access except `read` and `getattr` access (that is, "read access") to directories labeled with one of the types associated with the `domain` attributes. The wildcard operator in this rule means all types. A `neverallow` rule similar to this is commonly found in policies and is used to prevent inappropriate access to directories in `/proc/` that store process information (which will be labeled with the same type as processes).

We can see from the preceding example that the wildcard operator is needed in the source type lists for `neverallow` rules because we are referring to any and all types, including those not yet created. The wildcard operator allows us to prevent future mistakes.

Another common `neverallow` rule is this:

```
neverallow domain ~domain : process transition;
```

This `neverallow` rule reinforces the concept of the `domain` attribute described earlier in this chapter. This rule states that a process cannot transition to a type that does not have the `domain` attribute. This makes it impossible to create a valid policy with a type intended for a process that does not have the `domain` attribute.

Loadable Module Dependency Handling

Loadable policy modules, which are a new feature in *Fedora Core 5* (FC5), contain language features for handling dependencies between modules. The dependency handling features ensure that the policy components (that is, identifiers) a module expects are present at module installation time. See Chapter 13, "Managing an SELinux System," for more information about how loadable policy modules are installed and managed. Possible policy component dependencies include object classes, permissions, users, roles, types or aliases, attributes, and Boolean identifiers.

The `require` statement states the policy components required for a loadable module. All policy components that are not declared in the module must be required in some form. For example, consider the following require statement:

```
require { type etc_t; }
```

The example above states that the loadable module in which it appears requires the type `etc_t` to be declared elsewhere in the policy (that is, in the base module or other loadable modules). This `require` statement allows the type `etc_t` to appear in policy rules within the module without being explicitly declared. Following is a more complete example showing a more `require` statement, type declaration, and an example `allow` rule:

```
require {
       attribute domain;
       type etc_t;
       class file { read getattr };
}
type httpd_t, domain;
allow httpd_t etc_t : file { read getattr };
```

As you can see, every policy component used in the example `allow` rule was either declared or required before it was used. For example, the `domain` attribute was required before it was used in the `httpd_t` type declaration. Obviously, many `require` statements would be needed for a loadable module of any complexity. In Chapter 12, "Reference Policy," we discuss how the reference policy automates the generation of `require` statements.

We use the `require` statement to state unconditional requirements that must be present in the policy for the loadable module to be installed. The `optional` statement is used to state requirements that may or may not be present. This allows the policy author to add rules based on whether policy components are present. For example, consider the following `optional` statement:

```
optional {
          require { type user_home_t; }
          allow httpd_t user_home_t : file read;
}
```

This statement allows processes with the type `httpd_t` to read files with the type `user_home_t` *if* that type is present. As you can see, the `optional` statement wraps standard policy statements, including `require` statements. Whenever modules are added or removed from the system, all the optional dependencies are checked and enabled or disabled as appropriate.

The full syntax of the `require` statement is as follows:

`require {` *`require_list`* `}`

`require_list` One or more semicolon-separated require declarations. A require declaration consists of an identifier for the variety of policy component followed by the name of the policy component. Valid policy component variety identifiers are `class`, `user`, `role`, `type`, `attribute`, and `bool`. For users, roles, types, attributes, and Booleans, only a single name may be listed (for example, `type httpd_t;`). For object classes, both the object classes and one or more permissions is listed (for example, `class file { read write };`).

`Require` statements not a part of an optional statement are valid only in non-base loadable modules. They are not valid in a base module or in any conditional statements.

The full syntax for the optional statement is as follows:

`optional {` *`rule_list`* `}`

`rule_list` One or more policy statements that are enabled if all the required statements in the `optional` statement are satisfied. Valid policy statements are `user`, `role`, `type`, `attribute`, and `alias` declarations and TE and RBAC rules (including conditional statements).

`Optional` statements are valid only in base and non-base loadable policy modules. They are not valid in conditional statements.