Chapter 2

Concepts

In this chapter

The details of the SELinux access control mechanism and policy language are extensive and fully described in later chapters. However, the basic concepts and goals of SELinux are fairly simple. In this chapter, we examine the security concepts of SELinux and the motivations behind these concepts. Gaining a conceptual understanding is necessary to effectively use and apply SELinux access controls. This chapter focuses on the primary access control feature of SELinux, *type enforcement* (TE), although we also briefly discuss the optional multilevel security mechanism.

2.1 Security Contexts for Type Enforcement

All operating system access control is based on some type of access control attribute associated with objects and subjects. In SELinux, the access control attribute is called a *security context*. All objects (files, interprocess communication channels, sockets, network hosts, and so on) and subjects (processes) have a single security context associated with them. A security context has three elements: user, role, and type identifiers. The usual format for specifying or displaying a security context is as follows:

```
user:role:type
```

The string identifiers for each element are defined in the SELinux policy language, which we discuss in greater detail later. For now, just understand that a valid security context must have one valid user, role, and type identifier, and that the identifiers are defined by the policy writer. The namespaces for each identifier are orthogonal. (So, for example, it is possible, but not usually advisable, to have the same string identifier for a user, a role, and a type.)

Examining Security Contexts

SELinux modifies many system commands by adding the `-Z` option to display the security contexts of objects and subjects. For example, `ls -Z` shows the security contexts of file system objects and `ps -Z` shows the security contexts of processes. Another useful command is `id`, which shows the security context of your shell (that is, your current user, role, and type). The following, for example, shows the security context of a shell on a running SELinux system:

```
$ id -z
joe:user_r:user_t
```

You can use these commands to explore your own SELinux system as we walk through the examples in this chapter.

2.1.1 Comparing SELinux with Standard Linux

At this point, it is useful to compare the access control attributes on standard Linux with those of SELinux. For simplicity, we stick to common filesystem objects such as files and directories. In standard Linux, the access control attributes of subjects are the real and effective user and group IDs associated with all processes via the process structure in the kernel. These attributes are protected by the kernel and set via a number of controlled means, including the login process and setuid programs. For objects (for example, files), the inode of the file contains a set of access mode bits and file user and group IDs. The former controls access based on three sets of read/write/execute bits, one each for file owner, file group, and everyone else. The latter determines the file owner and group to decide which set of bits to use on a given access attempt.

As noted, in SELinux, the access control attributes are always the security context triple. All objects and subjects have an associated security context. Where standard Linux uses the process user/group IDs, the file's access mode, and the file user/group IDs to grant or deny access, SELinux uses the security contexts of a process and the object the process accesses. More specifically, because the primary access control feature of SELinux is type enforcement, the type identifier from the security context is used to determine access.

NOTE SELinux *adds* type enforcement to standard Linux. This means that both the standard Linux and enhanced SELinux access controls must be satisfied to access an object. So, for example, if we have SELinux write access to a file but we do not have `w` permission on the file, we cannot write the file.

Table 2-1 summarizes the comparison of standard Linux and the added SELinux security attributes and access control.

TABLE 2-1
Comparison of Standard Linux and Security-Enhanced Linux Access Control

	Standard Linux	SELinux Added
Process security attributes	Real and effective user and group IDs	Security context
Object security attributes	Access modes and file user and group IDs	Security context
Basis for access control	Process user/group ID and file's access modes based on file's user/group ID	Permissions allowed between process type and file type

2.1.2 More on Security Contexts

The security context is a simple, consistent access control attribute. In SELinux, the type identifier is the primary part of the security context that determines access. For historical reasons, the type of a process is often called a *domain.* The use of "domain" and "domain type" to mean the type of a process is so common and pervasive that we do not attempt to avoid using the term *domain.* In general, consider domain, domain type, subject type, and process type to be synonymous.

The user and role identifiers in a security context have little impact in the access control policy for type enforcement except for constraint enforcement, which we discuss in Chapter 7, "Constraints." For processes, user and role identifiers are more interesting because they are used to control the association of types with user identifiers and thus with Linux user accounts (more on this later). For objects, however, user and role identifiers have nearly no use. As a convention, the role of an object

is usually `object_r`, and the user of an object is usually the user identifier of the process that created the object. They have no effect on access control.

Finally, be aware of the differences between the user ID in standard Linux security and the user identifier in a security context. Technically, these are completely orthogonal identifiers, used separately by the standard and security-enhanced access control mechanisms, respectively. Any relationship between these two is strictly provided via the login process according to conventions not directly enforced by the SELinux policy.

2.2 Type Enforcement Access Control

In SELinux, all access must be explicitly granted. SELinux allows *no access by default,* regardless of the Linux user/group IDs. Yes, this means that there is no default superuser in SELinux, unlike root in standard Linux. The way access is granted is by specifying access from a subject type (that is, a domain) and an object type using an `allow` rule. An `allow` rule has four elements:

- *Source type(s)* Usually the domain type of a process attempting access
- *Target type(s)* The type of an object being accessed by the process
- *Object class(es)* The class of object that the specified access is permitted
- *Permission(s)* The kind of access that the source type is allowed to the target type for the indicated object classes

As an example, take the following rule:

```
allow user_t bin_t : file {read execute getattr};
```

This example shows the basic syntax of a TE `allow` rule. This rule has two type identifiers: the *source* (or subject or domain) type, `user_t`; and the *target* (or object) type, `bin_t`. The identifier `file` is the name of an *object class* defined in the policy (in this case, representing an ordinary file). The *permissions* contained within the braces are a subset of the permissions valid for an instance of the `file` object class. The translation of this rule would be as follows:

> A process with a domain type of `user_t` can read, execute, or get attributes for a file object with a type of `bin_t`.

As we discuss later, permissions in SELinux are substantially more granular than in standard Linux, where there are only three (`rwx`). In this case, `read` and `execute` are fairly conventional; `getattr` is less obvious. Essentially, `getattr` permission to a file allows a caller to view (not change) attributes such as date, time, and *discretionary access control* (DAC) access modes. In a standard Linux system, a caller may view such information on a file with only search permission to the file's directory even if the caller does not have read access to the file.

Assuming that `user_t` is the domain type of an ordinary, unprivileged user process such as a login shell process, and `bin_t` is the type associated with executable files that users run with the typical security privileges (for example, `/bin/bash`), the rule might be in a policy to allow users to execute shell programs such as the bash shell.

> NOTE There is no significance to the `_t` in the type identifier name. This is just a naming convention used in most SELinux policies; a policy writer can define a type identifier using any convenient convention allowed by the policy language syntax.

Throughout this chapter, we often depict allowed access using symbols: circles for processes, boxes for objects, and arrows representing allowed access. For example, Figure 2-1 depicts the access allowed by the previous `allow` rule.

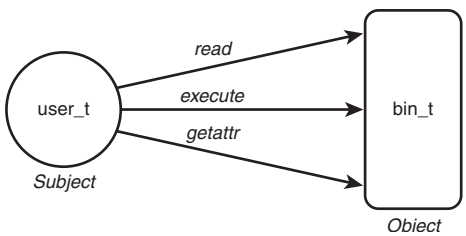

FIGURE 2-1
A depiction of an allow rule

2.2.1 Type Enforcement by Example

SELinux `allow` rules such as the preceding example are really all there is to granting access in SELinux. The challenge is determining the many thousands of accesses one must create to permit the system to work while ensuring that only the necessary permissions are granted, to make it as secure as possible.

To further explore type enforcement, let's use the example of the password management program (that is, `passwd`). In Linux, the password program is trusted to read and modify the shadow password file (`/etc/shadow`) where encrypted passwords are stored. The password program implements its own internal security policy that allows ordinary users to change only their own password while allowing root to change any password. To perform this trusted job, the password program needs the ability to move and re-create the shadow file. In standard Linux, it has this privilege because the password program executable file has the *setuid bit* set so that when it is executed by anyone, it runs as `root` user (which has all access to all files). However, many, many programs can run as `root` (in reality, all programs can potentially run as `root`). This means, any program (when running as `root`) has the potential to modify the shadow password file. What type enforcement enables us to do is to ensure that only the password program (or similar trusted programs) can access the shadow file, regardless of the user running the program.

Figure 2-2 depicts how the password program might work in an SELinux system using type enforcement.

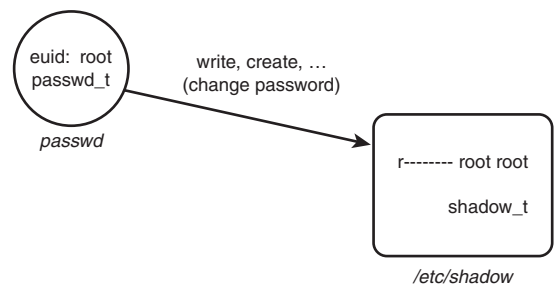

FIGURE 2-2
Type enforcement example: passwd program

In this example, we defined two types. The `passwd_t` type is a domain type intended for use by the password program. The `shadow_t` type is the type for the shadow password file. If we examine such a file on disk, we would see something like this:

```
# ls -Z /etc/shadow
-r———— root   root  system_u:object_r:shadow_t  shadow
```

Likewise, examining a process running the password program under this policy would yield this:

```
# ps -aZ
joe:user_r:passwd_t    16532 pts/0    00:00:00 passwd
```

For now, you can ignore the user and role elements of the security context and just note the types.

Examine the `allow` rule in Figure 2-2 The purpose of this rule is to give the `passwd` process' domain type (`passwd_t`) the access to the shadow's file type (`shadow_t`) needed to allow the process to move and create a new shadow password file. So, in reexamining Figure 2-2, we see that the depicted process running the password program (`passwd`) can successfully manage the shadow password file because it has an effective user ID of `root` (standard Linux access control) *and* because a TE `allow` rule permits it adequate access to the shadow password file's type (SELinux access control). Both are necessary, neither is sufficient.

2.2.2 The Problem of Domain Transitions

If all we had to do was provide allowed access for processes to objects such as files, writing a TE policy would be straightforward. However, we have to figure out a way to securely run the right programs in a process with the right domain type. For example, we do not want programs not trusted to access the shadow file to somehow execute in a process with the `passwd_t` domain type. This could be disastrous. This problem brings us to the issue of *domain transitions.*

To illustrate, examine Figure 2-3, in which we expand upon the previous password program example. In a typical system, a user (say Joe) logs in, and through the magic of the login process, a shell process is created (for example, running `bash`). In standard Linux security, the real and effective user IDs (that is, `joe`) are the same.[1] In our example SELinux policy, we see that the process type is `user_t`,

which is intended to be the domain type of ordinary, untrusted user processes. As Joe's shell runs other programs, the type of the new processes created on Joe's behalf will keep the `user_t` domain type unless some other action is taken. So how does Joe change passwords?

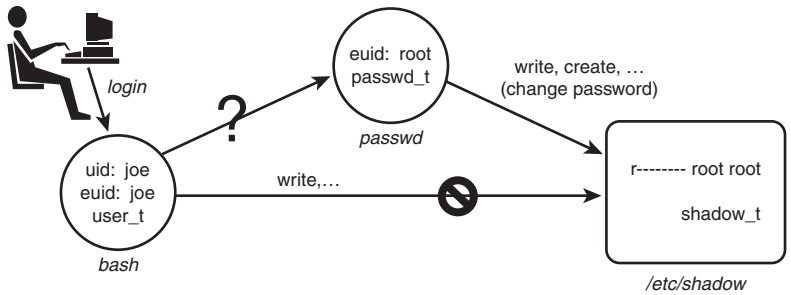

FIGURE 2-3
The problem of domain transitions

We would not want Joe's untrusted domain type `user_t` to have the capability to read and write the shadow password file directly because this would allow any program (including Joe's shell) to see and change the contents of this critical file. As discussed previously, we want only the password program to have this access, and then only when running with the `passwd_t` domain type. So, the question is how to provide a safe, secure, and unobtrusive method for transitioning from Joe's shell running with the `user_t` type to a process running the password program with the `passwd_t` type.

2.2.3 Review of SetUID Programs in Standard Linux Security

Before we discuss how to deal with the problem of domain transitions, let's first review how a similar problem is handled in standard Linux where the same problem of providing Joe a means to securely change his password exists. The way Linux solves this problem is by making `passwd` a setuid to the `root` program. If you list the password program file on a typical Linux system, you see something like this:

```
# ls -l /usr/bin/passwd
-r-s—x—x  1 root root 19336 Sep  7 04:11 /usr/bin/passwd
```

1 To be precise, Joe would not be a user ID. Rather, the string joe is used to determine the user ID (which is an integer number) from the password file (`/etc/passwd`). For ease of explanation, we skip that intermediate step and just use the string identifiers in our examples.

Notice two things about this listing. First the `s` in the `x` spot for the owner permission. This is the so-called *setuid bit* and means that for any process that executes this file, its effective UID (that is, the user ID used for access control decisions) will be changed to that of the file owner. In this case, `root` is the file owner, and therefore when executed the password program will always run with the effective user ID of `root`. Figure 2-4 shows these steps.

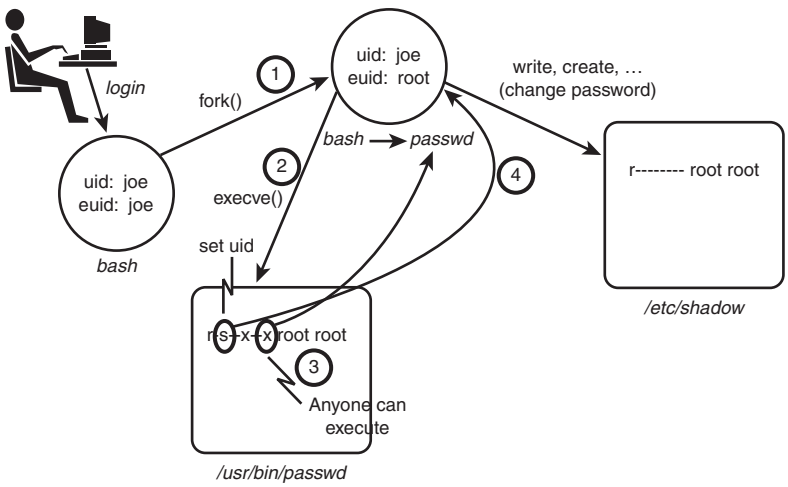

FIGURE 2-4
Password program security in standard Linux (setuid)

What actually happens when Joe runs the password program is that his shell will make a `fork()` system call to create a near duplicate of itself. This duplicate process still has the same real and effective user IDs (`joe`) and is still running the shell program (`bash`). However, immediately after forking, the new process will make an `execve()` system call to execute the password program. Standard Linux security requires that the calling user ID (still `joe`) have `x` access, which in this case is true because of the `x` access to everyone. Two key things happen as a result of the successful `execve()` call. First, the shell program running in the new process is replaced by the password program (`passwd`). Second, because the setuid bit is set for owner, the effective user ID is changed from the process' original ID to the file owner ID (`root` in this case). Because `root` can access all files, the password program can now access the shadow password file and handle the request from Joe to change his password.

Use of the setuid bit is well established in UNIX-like operating systems and is a simple and powerful feature. However, it also illustrates the primary weakness of standard Linux security. The password program needs to run as root to access the shadow file. However, when running as root, the password program can effectively access any system resource. This is a violation of the central security engineering principal of *least privilege.* As a result, we must trust the password program to be benign with respect to all other possible actions on the system. For truly secure applications, the password program requires an extensive code audit to ensure it does not abuse its extra privilege. Further, when the inevitable unforeseen error makes its way into the password program, it presents a possible opportunity to introduce vulnerabilities beyond accessing the shadow password file. Although the password program is fairly simple and highly trusted, think of the other programs (including login shells) that may and do run as root with that power.

What we would really like is a way to ensure least privilege for the password program and any other program that must have some privilege. In simple terms, we want the password program to be able to access only the shadow and other password-related files plus those bare-minimum system resources necessary to run; and we would like to ensure that no other program but the password (and similar) programs can access the shadow password file. In this way, we need only evaluate the password (and similar) programs with respect to its role in managing user accounts and need not concern ourselves with other programs when evaluating security concerns for user account management.

This is where type enforcement comes in.

2.2.4 Domain Transitions

As previously shown in Figure 2-2, the `allow` rule that would ensure that `passwd` process domain type (`passwd_t`) can access the shadow password file. However, we still have the problem of domain transitions described earlier. Providing for secure domain transition is analogous to the concept of setuid programs, but with the strength of type enforcement. To illustrate, let's take the setuid example and add type enforcement (see Figure 2-5).

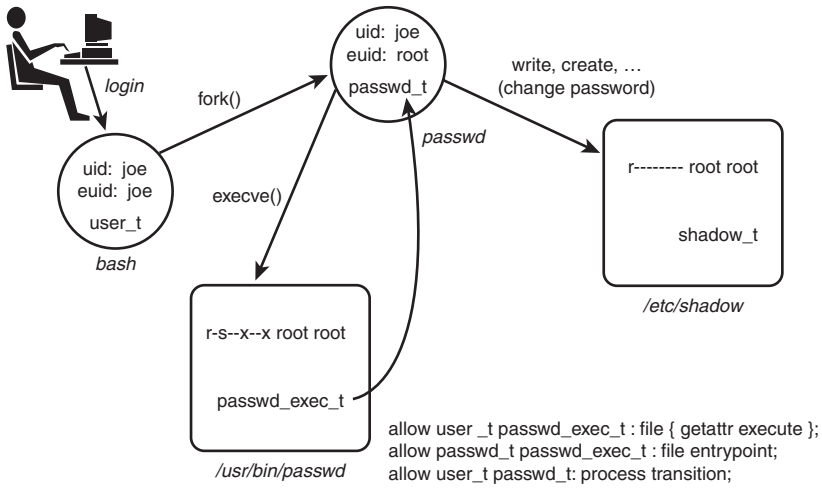

FIGURE 2-5
Passwd program security in SELinux (domain transitions)

Now our example is more complicated. Let's examine this figure in detail. First notice that we have added the three types we showed previously, namely Joe's shell domain (`user_t`), the password program's domain type (`passwd_t`), and the shadow password file type (`shadow_t`). In addition, we have added the file type for the `passwd` executable file (`passwd_exec_t`). For example, listing the security context for the password program on-disk executable would yield a result something like this:

```
# ls -Z /usr/bin/passwd
-r-s--x--x  root  root  system_u:object_r:passwd_exec_t  /usr/bin/passwd
```

Now we have enough information to create the TE policy rules that allow the password program (and presumably only the password program) to run with the `passwd_t` domain type. Let's look at the rules from Figure 2-5. The first rule is as follows:

```
allow user_t passwd_exec_t : file {getattr execute};
```

What this rule does is allow Joe's shell (`user_t`) to initiate an `execve()` system call on the `passwd` executable file (`passwd_exec_t`). The SELinux `execute` file permission is essentially the same permission as `x` access for files in standard Linux.

(The shell "stats" the file before trying to execute, hence the need for `getattr` permission, too.) Recall our description of how a shell program actually works. First it forks a copy of itself, including identical security attributes. This copy still retains Joe's shell original domain type (`user_t`). Therefore, the execute permission must be for the original domain (that is, the shell's domain type). That is why `user_t` is the source type for this rule.

Let's now look at the next allow rules from Figure 2-5:

```
allow passwd_t passwd_exec_t : file entrypoint;
```

This rule provides *entrypoint access* to the `passwd_t` domain. The `entrypoint` permission is a rather valuable permission in SELinux. What this permission does is define which executable files (and therefore which programs) may "enter" a domain. For a domain transition, the new or "to-be-entered" domain (in this case, `passwd_t`) must have `entrypoint` access to the executable file used to transition to the new domain type. In this case, assuming that only the `passwd` executable file is labeled with `passwd_exec_t`, and that only type `passwd_t` has `entrypoint` permission to `passwd_exec_t`, we have the situation that only the password program can run in the `passwd_t` domain type. This is a powerful security control.

> WARNING The concept of `entrypoint` permission is extremely important. If you did not fully understand the preceding example, please re-read it again before proceeding.

Let's now look at the final rule:

```
allow user_t passwd_t : process transition;
```

This is the first `allow` rule we have seen that did not provide access to file objects. In this case, the object class is `process`, meaning the object class representing processes. Recall that all system resources are encapsulated in an object class. This concept holds for processes, too. In this final rule, the permission is `transition` access. This permission is needed to allow the type of a process' security context to change. The original type (`user_t`) must have `transition` permission to the new type (`passwd_t`) for the domain transition to be allowed.

These three rules together provide the necessary access for a domain transition to occur. For a domain transition to succeed, all three rules are necessary; alone, none

is sufficient. Therefore, a domain transition is allowed only when the following three conditions are true:

1. The process' new domain type has `entrypoint` access to an executable file type.
2. The process' current (or old) domain type has `execute` access to the entry point file type.
3. The process' current domain type has `transition` access to the new domain type.

When all three of these permissions are permitted in a TE policy, a domain transition may occur. Further, with the use of the `entrypoint` permission on executable files, we have the power to strictly control which programs can run with a given domain type. The `execve()` system call is the only way to change a domain type,[2] giving the policy writer great control over an individual program's access to privilege, regardless of the user who may be invoking the program.

Now the issue is how does Joe indicate that he wants a domain transition to occur. The above rules allow only the domain transition; they do not require it. There are ways that a programmer or user can explicitly request a domain transition (if allowed), but in general we do not want users to have to make these requests explicitly. All Joe wants to do is run the password program, and he expects the system to ensure that he can. We need a way to have the system initiate a domain transition by default.

2.2.5 Default Domain Transitions: type_transition Statement

To support domain transitions occurring by default (as we want in the case of the password program), we need to introduce a new rule, the *type transition rule* (`type_transition`). This rule provides a means for the SELinux policy to specify default transitions that should be attempted if an explicit transition was not requested. Let's add the following `type transition` rule to the `allow` rules:

```
type_transition user_t passwd_exec_t : process passwd_t;
```

2 To be precise, a recent change to SELinux provides a means for a process, with necessary privilege, to change its security context without an `execve()` call. In general, without strong justification, this mechanism, described in Chapter 5, "Type Enforcement," should not be used because it greatly weakens the strength of type enforcement.

The syntax of this rule differs from the `allow` rule. There are still source and target types (`user_t` and `passwd_exec_t`, respectively) and an object class (`process`). However, instead of permissions, we have a third type, the *default type* (`passwd_t`).

`Type_transition` rules are used for multiple different purposes relating to default type changes. For now, we are concerned with a `type_transition` rule that has `process` as its object class. Such rules cause a default domain transition to be attempted. The `type_transition` rule indicates that, by default on an `execve()` system call, if the calling process' domain type is `user_t` and the executable file's type is `passwd_exec_t` (as is the case in our example in Figure 2-5), a domain transition to a new domain type (`passwd_t`) will be attempted.

The `type_transition` rule allows the policy writer to cause default domain transitions to be initiated without explicit user input. This makes type enforcement less obtrusive to the user. In our example, Joe does not want to know anything about access control or types; he wants only to change his password. The system and policy designer can use `type_transition` rules to make these transitions transparent to the user.

> NOTE Remember that a `type_transition` rule causes a domain transition to be attempted by default, but it *does not allow it*. You must still provide the three types of access required for a domain transition to successfully occur, whether it was initiated by default or as a result of the user's explicit request.

2.3 The Role of Roles

SELinux also provides a form of *role-based access control* (RBAC). The RBAC feature of SELinux is built upon type enforcement; access control in SELinux is primarily via type enforcement. Roles limit the types to which a process may transition based on the role identifier in the process' security context. In this manner, a policy writer can create a role that is allowed to transition into a set of domain types (assuming the type enforcement rules allow the transition), thereby defining the limits of the role. Take our password program example in Figure 2-5. Although according to the type enforcement rules, the password program can be executed by the `user_t` domain type to enter the new `passwd_t` domain, Joe's role must also be allowed to be associated with the new domain type for the transition to occur. To illustrate, we extend the password program example in Figure 2-6.

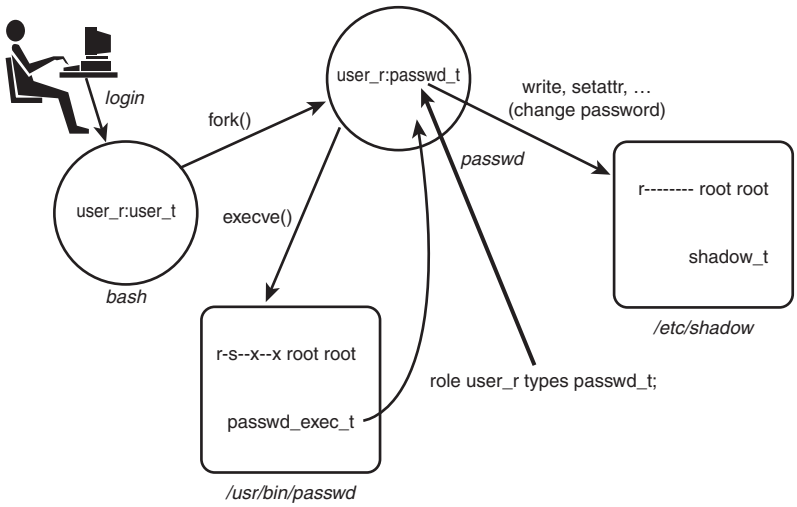

FIGURE 2-6
Roles in domain transitions

We have added the role portion (`user_r`) of the security contexts for the processes depicted. We also added a new rule, specifically the *role statement:*

```
role user_r type passwd_t;
```

The `role` statement declares role identifiers and associates types with the declared role. The previous statement declares the role `user_r` (if it has not already been declared in the policy) and associates the type `passwd_t` with the role. What this association means is that the `passwd_t` type is allowed to coexist in a security context with the role `user_r`. Without this `role` statement, the new context `joe:user_r:passwd_t` could not be created, and the `execve()` system call would fail, even though the TE policy allows Joe's type (`user_t`) all the necessary access.

A policy writer can define roles that are further constrained and then associate these roles to specific users. For example, imagine that in our policy we also create a role called `restricted_user_r`, identical to `user_r` in all regards except that it is not associated with the `passwd_t` type. Thus, if Joe's role is `restricted_user_r` instead of `user_r`, Joe would not be authorized to run the password program even though the TE rules would allow his domain type the access.

Chapter 6, "Roles and Users," discusses in detail the purposes of roles in SELinux and in particular how they are created and associated with users.

2.4 Multilevel Security in SELinux

Type enforcement is far and away the most important *mandatory access control* (MAC) mechanism that SELinux introduces. However, in some situations, primarily for a subset of classified government applications, traditional *multilevel security* (MLS) MAC coupled with type enforcement is valuable. In recognition of these situations, SELinux has always had some form of MLS capability included. The MLS features are optional and generally the less important of the two MAC mechanisms in SELinux. For the vast majority of security applications, including many if not most classified data applications, type enforcement is the best-suited mechanism for enhanced security. Nonetheless, the addition of MLS enhances security for some applications.

The basic concept of MLS was introduced in Chapter 1, "Background;" actual implementations of MLS are more involved. The *security level* used by MLS systems is a combination of a hierarchical *sensitivity* and a set (including the null set) of non-hierarchical *categories.* These sensitivities and categories are used to reflect real information confidentiality or user clearances. In most SELinux policies, the sensitivities (`s0`, `s1`, …) and categories (`c0`, `c1`, …) are given generic names, leaving it to user-space programs and libraries to assign user-meaningful names. (For example, `s0` might be associated with UNCLASSIFIED and `s1` with SECRET.)

To support MLS, the security context is extended to include security levels as such these:

```
user:role:type:sensitivity[:category,...][-sensitivity[:category,...]]
```

Notice that the MLS security context must have at least one security level (which is composed of a single sensitivity and zero or more categories), but can include two security levels. These two security levels are called *low* (or *current* for processes) and *high* (or *clearance* for processes), respectively. If the high security level is missing, it is considered to be the same value as the low (the most common situation). In practice, the low and high security levels are usually the same for most objects and processes. A range of levels is typically used for processes that are considered *trusted subjects* (that is, a process trusted with the ability to downgrade information) or multilevel objects such as directories that might contain objects of differing security levels. For purposes of this overview, assume that all processes and objects have a single security level.

The MLS rules for accessing objects are much the same as discussed in Chapter 1, except that security levels are not hierarchical but rather governed by a *dominance*

relationship. Unlike equality where a level is either higher than, equal to, or lower than another level, in a dominance relationship, there is a fourth state called *incomparable* (also known as *noncomparable;* see the definition of *incomp* in the following list). What causes security levels to be related via dominance rather than equality are the categories, which have no hierarchical relationship to one another. As a result, the four dominance operators that can relate two MLS security levels are as follows:

dom: (*dominates*) SL1 *dom* SL2 if the sensitivity of SL1 is *higher or equal to* the sensitivity of SL2, *and* the categories of SL1 are a *superset* of the categories of SL2.

domby: (*dominated by*) SL1 *domby* SL2 if the sensitivity of SL1 is *lower than or equal to* the sensitivity of SL2, *and* the categories of SL1 are a *subset* of the categories of SL2.

eq: (*equals*) SL1 *eq* SL2 if the sensitivity of SL1 and SL2 are *equal, and* the categories of SL1 and SL2 are the *same set.*

incomp: (*incomparable* or *noncomparable*) SL1 *incomp* SL2 if the categories of SL1 and SL2 cannot be compared (that is, neither is a subset of the other).

Given the domain relationship, a variation of the Bell-La Padula model is implemented in SELinux where a process can "read" an object if its current security level *dominates* the security level of the object, and "write" an object if its current security level *is dominated by* the security level of the object (and therefore read and write the object only if the two security levels are *equal*).

The MLS constraints in SELinux are in addition to the TE rules. If MLS is enabled, both checks must pass (in addition to standard Linux access control) for access to be granted. Chapter 8, "Multilevel Security," discusses the SELinux optional MLS features.

2.5 SELinux Features Familiarization

At this time, it is worthwhile to play with an SELinux system a little. For our examples, we use a *Fedora Core 4* (FC4) distribution with the strict policy. Most of these examples should also work on *Red Hat Enterprise Linux version 4* (RHEL4) or *Fedora Core 5* (FC5). You might also be able to work with other distributions, although there may be differences. Appendix A, "Obtaining SELinux Sample

Policies," describes how to obtain the policy files and other materials we use as examples throughout this book and how to configure your system accordingly.

Running in Permissive Mode

SELinux can run in permissive mode, where the access checks occur; but instead of denying unallowed access, it simply audits them. This mode is useful when first learning about SELinux, and you may want to start exploring the system in this mode. Of course, permissive mode should not be used in operational systems if you want the enhanced access security of SELinux. Note that some utilities are found in `/usr/sbin`, which is not normally in a regular user's path.

The simplest way to check the current mode of SELinux is to run the `getenforce` command. To set the system in permissive mode, run the command `setenforce 0`. (You must be logged in as `root` in the `sysadm_t` domain to change the system to permissive mode.) To return it to enforcing mode, run the command `setenforce 1`. (Because you are in permissive mode, you just need to be logged in as root to change the system to enforcing mode.)

We have already mentioned the `-Z` option added to some system commands. Commands such as `ls` and `ps` display the security contexts of files and processes. As an exercise, run the commands `ps xZ` and `ls -Z /bin` and examine the various security contexts for running processes and executable files.

2.5.1 Revisiting the Passwd Example

Throughout this chapter, we used the example of the shadow password file and the password program. If you examine the security context of these two files, their types should be `shadow_t` and `passwd_exec_t`, respectively. As discussed previously, `passwd_exec_t` is the entrypoint type for the `passwd_t` domain. To witness how the process of domain transitions work, walk through the following set of commands. You need two terminal windows or virtual consoles to do this walkthrough.

In the first window, run the `passwd` command:

```
$ passwd
Changing password for user joe.
Changing password for joe
(current) UNIX password:
```

This starts the password program and prompts for the user's current password. Do not enter the password, but instead switch to the second terminal. In the second terminal, `su` to `root` and then run the `ps` command:

```
$ su
Password:
Your default context is root:sysadm_r:sysadm_t.

Do you want to choose a different one? [n]
# ps axZ|grep passwd
user_u:user_r:passwd_t           4299 pts/1    S+     0:00 passwd
```

As you can see, the type of the running password program is `passwd_t`, as we would expect given the rules described in the examples earlier in this chapter.

> NOTE In a strict policy such as the one we use for our examples, a normal user (that is, a user running a shell in the `user_t` domain) does not have permission to read many `/proc/pid` entries, and as such the `passwd` process would not show up in the `ps axZ` output. That is why you need to `su` to `root` first.

2.5.2 Perusing the Policy File

In FC4 systems, the binary file containing the kernel policy is located in the well-known directory `/etc/selinux/`. The configuration file (`config`) in that directory indicates the policy to be used and loaded on boot. You can also configure the system to boot in permissive mode in this file. For our exercises, we are using FC4's strict policy, which (if installed according to Appendix A) should be here:

```
/etc/selinux/strict/policy/policy.[ver]
```

The version of the policy reflects the version of the SELinux policy compiler (`checkpolicy`). In our example, the version is 19. Configuring an SELinux system and creating a kernel policy file from policy sources are discussed in greater detail in Part III, "Creating and Writing SELinux Security Policies." For now, we want to look around inside the policy to see what is there.

A useful tool for examining the contents of a policy is the policy analysis tool *apol* created by Tresys Technology and distributed in a package of SELinux tools called SeTools (see Appendix D, "SELinux Commands and Utilities"). The SeTools package is included on most SELinux distributions. Run the command `apol` to determine whether the tool is present on your system. If not, Appendix D provides information on how to obtain the SeTools package.

The `apol` (for "analyze policy") tool is a sophisticated SELinux policy analysis tool that we use throughout the book to examine SELinux policies. For now, we want to use some of its basic features to examine aspects of the policy file. Run `apol` and open the strict policy file. Under the menu **Query > Policy Summary**, you can view a summary of the policy statistics (see Figure 2-7).

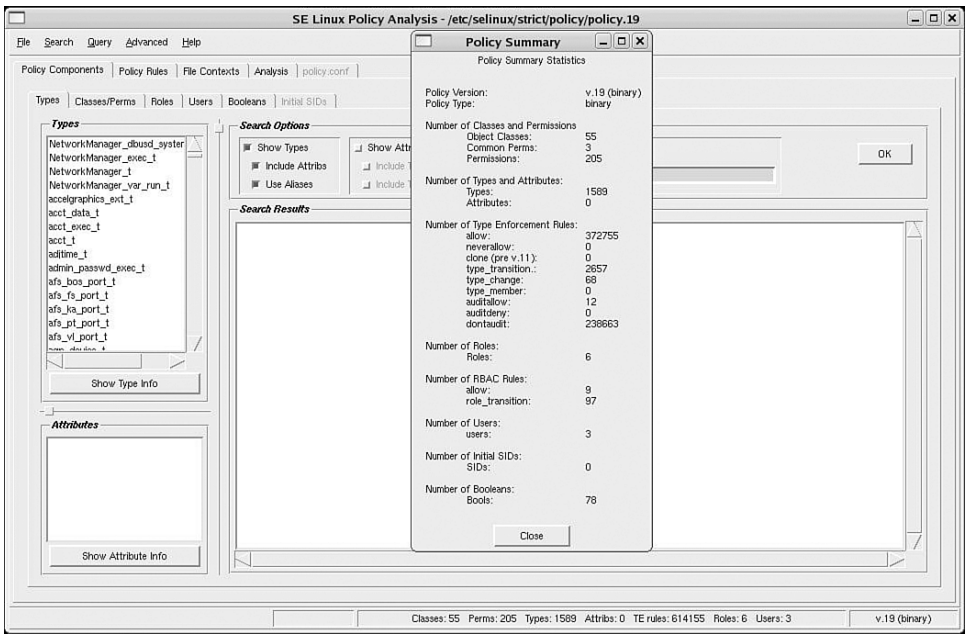

FIGURE 2-7
Policy summary using apol

`Apol` has a series of major tabs (Policy Components, Policy Rules, Analysis, and so on) that enable you to search and analyze a policy in various ways. Take some time to explore the Policy Components and Policy Rules tabs and become familiar with both portions of the policy we discussed in this chapter and the `apol` tool itself. You will find it useful throughout Part II, "SELinux Policy Language," to use `apol` to examine your policy and follow along with the examples.

2.6 Summary

- SELinux access control is based on a security context associated with all system resources including processes. The security context contains three elements: user, role, and type identifiers. The type identifier is the primary basis for access control.

  In SELinux, type enforcement is the primary access control feature. Access is granted between subjects (that is, processes) and objects by specifying `allow` rules that have the subject's type (also called a domain type) as the source and the object's type as the target. Access is granted for specified object classes using a fine-grained set of permissions defined for each object class.

  One of the key benefits of type enforcement is the ability to control which programs may run with a given domain type, thereby allowing access control down to individual programs (rather than the less-secure level of a user). The capability for a program to enter into a domain (that is, run with a given process type) is called domain transition and is tightly controlled by SELinux `allow` rules. SELinux also allows domain transitions to occur automatically through the `type_transition` rule.
- SELinux does not directly use the role identifiers in a security context for access control. Instead, all access is controlled based on types. Roles are used to associate the allowed domain types into which a process running on behalf of a user may transition. This allows sets of type enforcement allowed capabilities to be grouped together and authorized for a user as a role.
- SELinux provides an optional MLS access control mechanism that provides further access restrictions for a certain class of data sensitivity applications. The MLS features are built upon the TE mechanism. MLS also extends the security context to include a current (or low) security level and an optional high (or clearance) security level.

Exercises

1. What is a "domain" and how is it related to or different from a type?
2. What are the access control attributes used by SELinux type enforcement security to control access? What portion of the attribute is used by type enforcement for access control?
3. Let's assume that we have a file named `datafile` with the following security attributes:

   ```
   -r-xr-xr-x root root system_u:object_r:data_t datafile
   ```

   Let's also assume that your shell process type is `user_t` and that type has all access permissions for file objects of type `data_t`. Can you read and/or write this file? Why or why not?
4. For SELinux to allow a domain transition, a number of access permissions must be allowed among three types. What are the access permissions required and between what types? What do the types represent?
5. In answering Question 4, was a `type_transition` rule required? Why or why not?
6. In SELinux, a role is not used as a basis for access control, but it can prevent a domain transition from succeeding. How and why?

Extra credit: Examine the SELinux configuration file `/etc/selinux/config`. What are the possible states in which SELinux can run and what do each mean? How do the settings in this file differ from using the `setenforce` command?