

Verifying Information Flow Goals in Security-Enhanced Linux*

Joshua D. Guttman Amy L. Herzog John D. Ramsdell
Clement W. Skorupka

The MITRE Corporation
guttman, aherzog, ramsdell, ragnor@mitre.org

July 3, 2003

Abstract

In this paper, we present a systematic way to determine the information flow security goals achieved by systems running a secure O/S, specifically systems running Security-Enhanced Linux. A formalization of the access control mechanism of the SELinux security server, together with a labeled transition system representing an SELinux configuration, provides our framework. Information flow security goal statements expressed in linear temporal logic provide a clear description of the objectives that SELinux is intended to achieve. We use model checking to determine whether security goals hold in a given system. These formal models combined with appropriate algorithms have led to automated tools for the verification of security properties in an SELinux system. Our approach has been used in other security management contexts over the past decade, under the name *rigorous automated security management*.

1 Introduction

In the 1980s, most of the rigorous work in information security was focused on operating systems, but the 1990s saw a strong trend toward network and distributed system security. The difficulty of having an impact in securing operating systems was part of the motivation for this trend.

There were two major obstacles. First, the only operating systems with significant deployment were large proprietary systems. Superimposing a security model and gaining assurance that the implementation enforced the model seemed intractable [9]. Second, the prime security model [1] was oriented toward preventing disclosure in multi-level secure systems [2], and this required

*This work was funded by the United States National Security Agency.

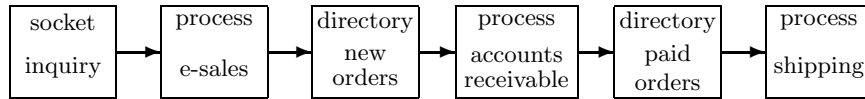


Figure 1: An E-Commerce Processing System

ensuring that even Trojan horse software exploiting covert channels in the system’s implementation could compromise information only at a negligible rate. This was ultimately found to be unachievable [14].

These obstacles seem more tractable now. Open-source secure operating systems are now available, which are compatible with existing applications software, and hence attractive for organizations wanting more secure platforms for publicly accessible servers. Security-Enhanced Linux (SELinux) in particular offers well thought out security services [7, 8, 10].

Moreover, a less stringent model of security, not focused on covert channels, is now relevant. Commonly, a network server must service unauthenticated clients (as in retail electronic commerce), or must provide its own authentication and access control for its clients (as in a database server). Sensitive resources must reside on the same server so that transactions can complete. In this situation, the programs manipulating the resources directly must be trustworthy; if they can be replaced with uncontrolled software such as Trojan horses, then security for the system is hopeless. Thus, we may focus our attention on protecting the confidentiality and integrity of these resources in a context where programs interacting directly with sensitive resources have a degree of trust, and in particular there is no advantage to considering covert channels.

To preserve integrity, each causal chain of interactions leading from untrusted sources to sensitive destinations must traverse a program considered trusted to engage only in legitimate transactions with that destination. Dually, to preserve confidentiality, causal chains leading from sensitive sources to untrusted destinations must traverse a program trusted to filter outbound data. The trustworthy program determines what data can be released to the untrusted destination. For both integrity and confidentiality, the security goal is an *information flow goal*. Each says that information flowing between particular endpoints must traverse specific programs along its path.

As an example focused on integrity, consider the e-commerce processing system described in Figure 1. In this scenario, orders are submitted by customers through an SSL-protected network socket at the left. An e-sales program ensures that a customer order is properly formatted, and if so, that the purchase prices for the different items are correct. For simplicity, we let the program write accepted orders to a file in a directory meant for new orders. Files in this directory will be read by an accounts receivable program, which after some on-line interaction with a credit card clearing house, causes the company’s account to be credited. The order may now be written to the directory for paid orders. The shipping department program then checks inventory and causes the order to be

shipped as soon as the goods are available. Thus, the company wants to ensure that orders with erroneously low prices cannot arrive at accounts receivable, and that unpaid orders cannot arrive at the shipping department.

In this paper, we aim at three goals. First, we develop a highly abstract model of the SELinux operating system access control mechanism. In this model, the system configuration determines a labeled transition system representing possible information flows (Section 2). Second, we propose a diagram-like way to state security goals, and give meaning to diagrams like Figure 1 using temporal logic (Section 3). Third, we describe briefly how to determine, using model checking, whether a goal is enforced by a particular configuration (Section 4). We regard this as a kind of rigorous, automated security management (Section 5, cf. [5]).

2 An SELinux Model

In this section, we will introduce the core ideas SELinux uses for access control (Section 2.1), after which we will introduce five relations that will summarize the contents of an SELinux system access control configuration file (Section 2.2). Section 2.3 describes how these relations are defined from the contents of a configuration file. The authorization relation (Section 2.4) synthesizes information from the five relations to determine which actions are allowed and which are prohibited. The flow relation then expresses what entities are affected by permissible actions (Section 2.5). If an action is permitted, does the state of the process requesting the action change to reflect the state of a resource accessed, as is the case with a file read; or does the state of the resource change to reflect that state of the process, as is the case with a file write?

2.1 Underlying Ideas of SELinux Access Control

The SELinux security server makes decisions about system calls, for instance whether a process should be allowed to write to a particular file, or whether a process should be allowed to overlay its memory with the binary image contained at a particular pathname, and continue executing the result. For each system call, SELinux specifies one or more checks that must be satisfied in order for the call to complete. Each check is labeled by a pair consisting of a *class* and a *permission*. The class describes a kind of resource that the access involves, such as `file`, `process`, or `filesystem`. The permission describes the action itself, such as `read`, `write`, `mount`, or `execute`. By a *resource*, we mean any object in an SELinux system; processes, files, sockets, etc. are all regarded as resources. Each resource has a *security context* which summarizes its security relevant status.

In making a check, the security server receives as input two facts, the security contexts of the process and of another resource involved in the system call. A security context is a tuple consisting of three components,¹ called a *type*, a *role*,

¹or four components, if the system is compiled with support for multi-level security as it

Call	Action		Source	Target
	Class	Permission	(Process)	(Resource)
execve	dir	search	Current	Path
	file	execute	Current	File
	process	transition	Current	New
	process	entrypoint	New	File
	process	ptrace	Parent	New
	fd	inherit	New	FD

Table 1: Actions in an `exec` System Call

and a *user*. The user is similar in intent to the normal Unix notion of user, and represents the person on behalf of whom the system is executing a process or maintaining a resource. The role, derived from the literature on role-based access control, is an intermediate notion intended to connect a collection of users with a corresponding collection of programs that they should be permitted to execute. Associated with the user component is a specification of the roles that user is permitted; the *role* then in turn specifies what types of processes those users are permitted to execute.

The most important component is the type, accounting for at least 22,000 out of the 22,500 access control statements in the sample policy file contained in the distribution. The type is used to specify the detailed interactions permitted between processes and other resources. For each system call, zero or more actions must be authorized; for each of these actions, the `SELinux` system will check that the type of the process is allowed to engage in this action against the type of some other resource involved in the system call. If any of these checks fails, then the system call will terminate before the kernel causes the corresponding change in system state.

For instance, in order to read a file, a process must be permitted to engage in the `file read` action against it. However, the read system call also causes an update to the attributes associated with the file descriptor, indicating the current time as the last time of file access. Thus, it must also be permitted to engage in the `fd setattr` action. The `exec` system call has the most complex control requirements; they are summarized in Table 1. Several different resources are involved in these control requirements. The process type is the type of the current process in some; the type of the process after the `exec`, which may be different, in others; and the type of the parent process for `ptrace`. The path to the executable file, the file itself, the new process, and the file descriptors to be inherited furnish the target types considered for the different actions.

It will be convenient to refer to the set of all types as T , the set of all roles as R , the set of all users (i.e. user names) as U , the set of all classes as C , and

can be, but normally is not. For definiteness, we will assume MLS support is not compiled into the kernel in the remainder of this paper, although the approach we describe is equally applicable if it is.

the set of all permissions as P . Because not all permissions make sense with all classes, we write $\Gamma \subset C \times P$ for the set of class-permission pairs that are used in SELinux.

A type $t \in T$ is called a *domain* if t is the type of any process; we write $D \subset T$ for the set of domains. All roles but one are used to constrain the association of users with the types of processes. The dummy role r_o (`object_r`) is used in security contexts where the type is not a domain.

2.2 SELinux Access Control Relations

The configuration file defines several relations, five of which are of interest to us. The others concern auditing and other issues that not related to information-flow security goals. Each relation is built up by statements contained in the same configuration file.

- $\alpha(t_1, t_2, c, p)$ is the type permission relation. It holds if $t_1 \in D$, $t_2 \in T$, and $(c, p) \in \Gamma$ for some `allow` statement in the configuration.
- $\alpha_\rho(r_1, r_2)$ is the role transition relation. When a process changes security context, the role may change, but the old and new roles must satisfy α_ρ .
- $\rho(r, t)$ is the role-type relation. Each process in the system must have a security context such that $\rho(r, t)$ holds.
- $\mu(u, r)$ is the user-role relation. Each process in the system must have a security context such that $\mu(u, r)$ holds.
- $\chi_{c,p}(t_1, r_1, u_1; t_2, r_2, u_2)$ is the constraint relation. Whenever c, p is requested, the system checks that the constraint $\chi_{c,p}(t_1, r_1, u_1; t_2, r_2, u_2)$ holds between the process security context and the resource security context. Constraints may be used to ensure that only privileged types of process change the user or role of existing resources, for instance.

We express the special status of the dummy object role by stipulating that $\rho(r_o, t)$ and $\mu(u, r_o)$ hold whenever t is not a domain. We also assume that $\chi_{c,p}$ is empty when $(c, p) \notin \Gamma$.

2.3 Syntax and Semantics for the Configuration File

Types are declared with statements such as

```
type esales_sock_t;
type esales_t, domain;
type esales_exec_t, file_type, exec_type;
type new_orders_dir_t, file_type;
```

Any names after the first comma are attribute declarations, which declare that the new type belongs to a set named by the attributes. For instance, the `domain`

```

allow esales_t esales_sock_t:      tcp_socket
    { ioctl read getattr write setattr append bind connect
      getopt setopt shutdown listen accept };
allow esales_t new_orders_dir_t:   file { create write };
allow acct_rcv_t new_orders_dir_t: file { read };
allow acct_rcv_t paid_orders_dir_t: file { create write };
allow shipping_t paid_orders_dir_t: file { read };

allow sysadm_t esales_t :          process transition;
allow sysadm_t esales_exec_t:      file entrypoint;

```

Figure 2: Electronic Commerce Example Allow Statements

attribute of `esales_t` stipulates that it belongs to the set of domains D . Attributes may be used in other configuration statements to refer to the set of all types having that attribute.

For each type, some actions are specified that processes executing with that type are allowed to perform. In the SELinux configuration file they are introduced by the keyword `allow`. For a request to succeed, some `allow` statement in the configuration file must authorize it. Each `allow` statement

$$\text{allow } T_s T_t : C_a P_a;$$

specifies a set of process (source) types T_s , a set of resource (target) types T_t , a set of classes C_a , and a set of permissions P_a . We will not be specific about the syntax with which the sets of symbols such as T_s , T_t etc. are presented. If a process whose type is in T_s requests an action with a class-permission pair in $C_a \times P_a$ against a resource with type in T_t , then that request is authorized.

The `allow` statements for the electronic commerce example are gathered in Figure 2. The last two lines in the figure allow a process executing with type `sysadm_t` to initiate the `esales` program, transitioning to the new type `esales_t`, assuming that the `esales` executable is a binary image contained in some disk file of type `esales_exec_t`.

If the keyword `self` occurs as a target type, then it is treated specially, as indicating that each source type is allowed to engage in the specified actions with itself. In this case, the statement implies that the tuple $(t, t, c, p) \in \alpha$ for each $t \in T_s, c \in C_a, p \in P_a$. Letting $T'_t = T_t \setminus \{\text{self}\}$, then the statement always implies $T_s \times T'_t \times C_a, P_a \subset \alpha$. The relation α is the smallest relation compatible with the `allow` statements and these two rules.

The role `allow` statement takes the form

$$\text{allow } R_c R_n;$$

It controls change of role when a process has a transition from one security context to another; if the current role is in R_c and the new role is in R_n , then this change of role will be permitted. More precisely, the statement specifies

```

role ecomm_r types esales_t;
role ecomm_r types acct_rcv_t;
role ecomm_t types shipping_t;

allow sysadm_r ecomm_r;
allow system_r ecomm_r;

```

Figure 3: Electronic Commerce Example Role and Role Allow Statements

that $R_c \times R_n \subset \alpha_\rho$. The relation α_ρ is the smallest relation compatible with the role allow statements and this rule.

A role r is declared with the following syntax.

```
role r types Tr;
```

In addition to declaring the role name r , the statement stipulates a set of types with which r is permitted to be associated. The same role may be declared repeatedly to allow a large set of types to be declared conveniently. The statement specifies that $\{r\} \times T_r \subset \rho$. The relation ρ is the smallest relation compatible with the role declarations, the convention about r_o mentioned at the end of Section 2.2, and this rule. The role declarations and role allow statements for the electronic commerce example are contained in Figure 3.

A user u is declared with the following syntax.

```
user u types Ru;
```

In addition to declaring the user name u , the statement stipulates a set of roles with which u is permitted to be associated. The same user may be declared repeatedly to allow a large set of roles to be declared conveniently. The statement specifies that $\{u\} \times R_u \subset \mu$. The relation μ is the smallest relation compatible with the role declarations and this rule. No users need be added in the electronic commerce example.

Constraint definitions specify additional limits on transitions with the following syntax.

```
constraint Cc Pc  $\delta$ ;
```

The constraint expression δ has a natural translation as a set of pairs of security contexts $\bar{\delta}$. Since a security context is a triple consisting of a type, a role, and a user, $\bar{\delta} \subset T \times R \times U \times T \times R \times U$, where T, R, U are the sets of all types, roles, and users respectively. The value $\chi_{c,p}$ is the intersection of all $\bar{\delta}$ where δ is declared as a constraint on C_c, P_c where $c \in C_c$ and $p \in P_c$. In effect, this means that all relevant constraints must hold true for an action to occur. No constraints need be added in the electronic commerce example.

2.4 The Authorization Relation

Our formal model of the SELinux authorization mechanism puts these five relations together in a specific way. The class-permission pair c, p is authorized for

a process with security context t_1, r_1, u_1 against a resource t_2, r_2, u_2 if:

$$\begin{aligned}
& \alpha(t_1, t_2, c, p) \\
& \wedge \quad \rho(r_1, t_1) \quad \wedge \quad \rho(r_2, t_2) \\
& \wedge \quad \mu(u_1, r_1) \quad \wedge \quad \mu(u_2, r_2) \\
& \wedge \quad \chi_{c,p}(t_1, r_1, u_1; t_2, r_2, u_2) \\
& \wedge \quad \text{if } c = \mathbf{process} \wedge p = \mathbf{transition} \quad \text{then } \alpha_\rho(r_1, r_2).
\end{aligned}$$

This relation $\Delta_{c,p}(t_1, r_1, u_1; t_2, r_2, u_2)$ is the SELinux authorization predicate. The role allow relation α_ρ is relevant only in a single special case, namely when a process is making a transition to a new security context.

2.5 The Information Flow Relation

Some events (file write, for instance) transfer information from process to resource, while others (file read, for instance) transfer it from resource to process. SELinux has a file that describes how each c, p transfers information, whether like a read, like a write, in both directions, or in neither. Information flows from an entity with security context t, r, u to an entity with security context t', r', u' if for some event c, p either

$$c, p \text{ has write-like flow and } \Delta_{c,p}(t, r, u; t', r', u')$$

or else

$$c, p \text{ has read-like flow and } \Delta_{c,p}(t', r', u'; t, r, u).$$

$\Phi_{c,p}(t, r, u; t', r', u')$ means that at least one of these conditions holds, hence that there is flow from context t, r, u to context t', r', u' through event c, p .

The file defining the direction of flow for each class-permission pair contains only a simple approximation. It does not take into account indirect flows caused by error conditions or variations in timing, and it does not consider flow into other system resources besides the process requesting the event and the resource against which the event is requested. This is why our analysis avoids the subtleties of covert channels.

Having defined the information flow relation $\Phi_{c,p}(t, r, u; t', r', u')$, we regard it as a transition relation and consider what can be expressed in standard temporal logic in terms of this transition relation.

We regard the state as a sextuple $\langle t, r, u, c, p, k \rangle$ consisting of a type, user, and role, as well as a class and permission signifying the transition about to occur; the last component k is a Boolean flag used to make the transition relation total. When true, it indicates that all transitions so far have been legitimate.

The transition relation is highly non-deterministic. If k is false, then in the next state k must remain false, although the remaining components can take any value. If k is true, then k' is true in the next state only when the type, user, and role are values t', r', u' such that $\Phi_{c,p}(t, r, u; t', r', u')$; the next c', p' is unconstrained. Otherwise, k' is false.

The initial states $\langle t, r, u, c, p, k \rangle$ for this model are the ones that are compatible with ρ and μ in that

$$\rho(r, t) \wedge \mu(u, r).$$

Φ says that there is a causal effect of one state on the next, and iterated applications of the relation say that there is some sequence of events (possibly involving many different processes and resources) creating a causal chain from the first state of the sequence to the last.

The model we have just developed, and encoded in the information flow predicate $\Phi_{c,p}$, is an enormous simplification of the SELinux system. It concentrates on the information flow consequences of individual events, and abstracts from all aspects of system resources apart from their security contexts. The benefit of this approach is to provide a minimal representation still allowing us to analyze core security goals achieved by an SELinux configuration.

3 Security Goals

The purpose of securing a system is to protect the confidentiality and integrity of sensitive resources. As described above, protecting these resources entails ensuring that information flowing from one place to another must traverse specific points along its path. Security goals of this sort are examples of intransitive noninterference: information flow from one security context to another is only acceptable if it happens through another, trustworthy, program [12]. These intransitive noninterference chains cover realistic security desires, ranging from the simple to the complex.

Consider the case of raw disk access. Since accessing the disk directly bypasses traditional access controls, it is likely that a system administrator would want only specific administrative programs to have the ability. In the sample SELinux policy, raw disk data has the type `fixed_disk_device_t`, and the type `fsadm_t` is used for administrative programs requiring direct disk access. The system administrator aims to ensure that all one-step paths ending in `fixed_disk_device_t` begin with `fsadm_t`. This is the simplest form of causal chain.

Web servers provide slightly more complex security needs. When running a web server, it is very important to keep user capabilities separate from server administrator capabilities. Since frequently the administrator is also a user on the system, the core desire is to prevent users from performing sensitive operations without first supplying (for example) the administrator password. A concrete example from the sample SELinux policy involves the modification and execution of web server system scripts. A special domain, `httpd_admin_t`, is defined for web server administration functions. Thus in this case, we wish to ensure that any path of information flow starting at `user_t` and ending in a write to scripts of the type `httpd_sys_script_t` passes through `httpd_admin_t`.

A still more complex security goal is pictured in the e-commerce example in Figure 1. This pictures an integrity goal: although untrusted users may connect to the server, only paid orders should be shipped. A process with SELinux type

`esales.t` checks new orders read from a resource with type `esales_sock.t`; after being checked they are written to a file with type `new_orders_dir_type`. The accounts receivable program has type `acct_rcv.t`, and writes out paid orders to files with type `paid_orders_dir.t`, which are readable by a process having type `shipping.t`.

3.1 Visualizing Causal Chains: Diagrams

We will now formalize a particular way to express information flow goals. We wish to ensure that all paths through a system from a starting security context to a final security context go through a series of intermediate steps. These intermediate steps can be viewed, as in Figure 1, as security contexts or sets of them. We may also sometimes wish to specify the *means* by which one security context can affect another: in other words, we may wish to label the arrows in Figure 1 with class-permission pairs such as `socket read`, `file create`, `file write`, etc. Information flow security goals are expressed in this alternating chain of security contexts and actions.

When constructing a chain, one has four degrees of freedom. First, one can define what security contexts appear at a stage in the process; formulas defining these sets use symbols such as σ_i . We adopt the convention of writing “ σ_i is ϕ ” where ϕ involves only the variables t, r, u free, to mean that σ_i is defined to be the set $\{(t, r, u) \mid \phi\}$. For instance, for the web server integrity goal described above,

$$\begin{aligned}\sigma_0 &\text{ is } t = \text{user.t} \\ \sigma_1 &\text{ is } t = \text{httpd.admin.t} \\ \sigma_2 &\text{ is } t = \text{httpd.sys.script.t}.\end{aligned}$$

Second, one may characterize what actions or events may transfer information from one context to the next; formulas defining these sets use symbols such as γ_i . Again, we adopt the convention of writing “ γ_i is ϕ ” where ϕ involves the variables c, p , to mean that γ_i is defined to be the set $\{(c, p) \mid \phi\}$. Members of γ_i are SELinux class-permission pairs. Following our web server example goal,

$$\begin{aligned}\gamma_0 &\text{ is } \text{true} \\ \gamma_1 &\text{ is } c = \text{file} \wedge p \in \{\text{write}, \text{append}\}.\end{aligned}$$

The third kind of freedom captures the intuitive notion of the *length* of the arrows. Between two security contexts in our causal path, we may be interested in constraining the paths to a single event. However, we may also be interested in potentially longer paths between contexts. Our raw disk access example concerns only direct disk accesses, rather than longer chains. Our web server security goal is concerned with longer paths, since it would be wise to consider all information flows from a user ending in the writing of HTTP system scripts.

The e-commerce example provides a more concrete example of longer paths which may be perfectly acceptable. Perhaps some customers receive a special discount on their order, which may need to be checked between the e-sales program and the new orders file. Thus, flow from `esales.t` to `new_order_type`

may go by way of a process with a different type that validates the discount, and possibly other intermediate types.

We distinguish iterated events by surrounding them with square brackets and a superscript $+$. Let λ_i be a label of one of the forms γ_i or $[\gamma_i]^+$. The final action formulas for our web server example are the following:

$$\begin{aligned}\lambda_0 &\text{ is } [true]^+ \\ \lambda_1 &\text{ is } [c = \text{file} \wedge p \in \{\text{write}, \text{append}\}]^+.\end{aligned}$$

The fourth degree of freedom is that of exceptions, although we will return later to fill in this detail (see Section 3.2.3). Ignoring exceptions, we can write an information flow policy goal in the following visual form:

$$\sigma_0 \xrightarrow{\lambda_0} \sigma_1 \xrightarrow{\lambda_1} \dots \xrightarrow{\lambda_{n-2}} \sigma_{n-1} \xrightarrow{\lambda_{n-1}} \sigma_n \quad (1)$$

Note the similarity between this form and our e-commerce example in Figure 1. Translating the picture in Figure 1 to our information flow formulas, we obtain the diagram shown in Figure 4. The σ_i correspond directly to the boxes pictured in Figure 1. The labels λ_i correspond to the arrows pictured there. However, when a process reads from a socket, information flows forward from the socket to the process. When it needs to set socket options, information also flows backward to the socket. For this reason, we must allow for longer, cyclic paths between `esales_sock_t` and `esales_t`. The $+$ sign permits longer paths in this case.

$$\sigma_0 \xrightarrow{\lambda_0} \sigma_1 \xrightarrow{\lambda_1} \sigma_2 \xrightarrow{\lambda_2} \sigma_3 \xrightarrow{\lambda_3} \sigma_3 \xrightarrow{\lambda_4} \sigma_5 \quad \text{where}$$

σ_0 is $t = \text{esales_sock_t}$
 λ_0 is $c = \text{tcp_socket}$
 σ_1 is $t = \text{esales_t}$
 λ_1 is $[c = \text{tcp_socket} \vee (c = \text{file} \wedge p \in \{\text{create}, \text{write}\})]^+$
 σ_2 is $t = \text{new_orders_dir_t}$
 λ_2 is $c = \text{file} \wedge p = \text{read}$
 σ_3 is $t = \text{acct_rcv_t}$
 λ_3 is $c = \text{file} \wedge p \in \{\text{create}, \text{write}\}$
 σ_4 is $t = \text{paid_orders_dir_t}$
 λ_4 is $c = \text{file} \wedge p = \text{read}$
 σ_5 is $t = \text{shipping_t}$

Figure 4: Security Diagram for the Electronic Commerce Example

3.2 Formalizing Diagrams in Linear Temporal Logic

We interpret an information flow policy as an assertion about all sequences of state transitions leading from a state in σ_0 to a state in σ_n . It asserts that this

path must encounter the σ_i in the order given, executing events from λ_i in each stage.

To formalize these assertions, we first represent the fact that they concern only state transitions leading from states in σ_0 to states in σ_n . We may express this as the hypothesis $H = \sigma_0 \wedge \Diamond(\sigma_n \wedge k)$, stating that σ_0 currently holds and σ_n will eventually hold, and where k is true because all transitions so far have satisfied Φ . We interpret an information flow diagram (1) by two formulas of Linear Temporal Logic (LTL).

3.2.1 Order Assertions

The first formula asserts that states are encountered in the right order, subject to the hypothesis H that we are passing from σ_0 to σ_n :

$$H \Rightarrow \bigwedge_{0 < i < n} \sigma_i \mathcal{R} \neg \sigma_{i+1}. \quad (2)$$

The operator \mathcal{R} (“releases”) asserts that its right hand operand is true and remains true until its left hand operand has been true at least once. Thus, this formula asserts that each set σ_{i+1} is not encountered until after σ_i has been encountered, along paths from σ_0 to σ_n .

3.2.2 Event Assertions

The other formula asserts that the transitions along a path from σ_0 to σ_n proceed using the right class-permission pairs. From the time that σ_i has been encountered but σ_{i+1} has not yet been reached, all of the transitions should be in the set γ_i . In the case where none of the γ_i are decorated with a $+$, this leads to the formula

$$H \Rightarrow \gamma_0 \wedge \mathcal{X}(\sigma_1 \wedge \gamma_1 \wedge \mathcal{X}(\dots)).$$

$\mathcal{X}\phi$ asserts of a state that ϕ is true in the next state immediately after it. Thus, we start with a γ_0 which brings us to σ_1 and then continue with a γ_1 which brings us to σ_2 and so on. If all the γ_i are surrounded by square brackets with a $+$, then we want to say that a γ_i occurs, and then γ_i s continue until a σ_{i+1} is reached, and so on:

$$H \Rightarrow \gamma_0 \wedge \mathcal{X}(\gamma_0 \mathcal{U} (\sigma_1 \wedge \gamma_1 \wedge \mathcal{X}(\gamma_1 \mathcal{U} \dots))).$$

$\phi \mathcal{U} \psi$ is true in a state if ψ eventually becomes true, and ϕ remains true until the first such occasion. We combine the two forms into a formula

$$H \Rightarrow \gamma_0 \mathcal{O}_0 (\sigma_1 \wedge (\gamma_1 \mathcal{O}_1 (\sigma_2 \dots))). \quad (3)$$

When the label λ_i is of the form γ_i , then $\phi \mathcal{O}_i \psi$ is defined to be $\phi \wedge \mathcal{X} \psi$. When the label λ_i is of the form $[\gamma_i]^+$, then $\phi \mathcal{O}_i \psi$ is defined to be $\phi \wedge \mathcal{X}(\phi \mathcal{U} \psi)$.

Formulas 2 and 3 do not need a leading “always” \Box , because, for states such that k is true, all accessible states are also initial states.

3.2.3 Exceptions

In some cases, we want to make an assertion subject to some exceptions. If the exception occurs, we do not care what the information flow is; if the exception does not occur, then we want the information flow diagram to hold true as before. The exceptional condition may be either a state or a transition.

For instance, in the e-commerce case, an example of an exceptional state could arise from queries about order status. There may be a directory for status queries, with type `query_t`, such that flow from a network socket to the shipping department program is permitted if it comes by way of `query_t`. There is then a corresponding security requirement on the shipping department program, stating that input from files of this type never cause products to be shipped, but verifying that requirement is a matter for programming language security analysis (see e.g. [13]) rather than operating system security analysis.

The exceptional condition may also be a transition, that is, a class-permission pair. For instance, perhaps the accounts receivable program can send a signal to the shipping program to tell it when to stat the shared directory. This signal is a flow of information to shipping that does not traverse the type `paid_orders_t`. However, it is merely advisory, and we know it causes nothing to be shipped unless the program succeeds in reading a new paid order. Thus, there is no need to prohibit this flow.

We incorporate exceptions without changing the form of Equations 2 and 3. Instead, let σ_e be the set of exceptional states, and let γ_e be the set of exceptional transitions; we redefine H to take the form:

$$\sigma_0 \wedge ((\neg\sigma_e \wedge \neg\gamma_e) \mathcal{U} (\sigma_n \wedge k))$$

Thus, we concern ourselves with a path only if it started at σ_0 and avoided σ_e and γ_e until reaching a state in which $\sigma_n \wedge k$. If a path is of this form, then we require that the bodies of Equations 2 and 3 hold. Taking into account exceptions, we can now write information flow policy goals in the following visible form:

$$\sigma_0 \xrightarrow{\lambda_0} \sigma_1 \xrightarrow{\lambda_1} \dots \xrightarrow{\lambda_{n-2}} \sigma_{n-1} \xrightarrow{\lambda_{n-1}} \sigma_n \quad [\sigma_e; \gamma_e] \quad (4)$$

As we will see in Section 4, exceptions become important very quickly when analyzing complex policies such as the policy included in the sample SELinux distribution.

4 Goal Enforcement and Implementation

We have written software that reads and analyzes SELinux configuration files. It constructs a labeled transition system with security contexts as states, and actions as transition labels. The transition relation represents the information flow relation $\Phi_{c,p}$, and the initial states are the security contexts that satisfy $\rho(r, t) \wedge \mu(u, r)$. The labeled transition system is written to disk in an easily read format.

We have tools that use the labeled transition system to construct a Binary Decision Diagram (BDD) for specialized processing, but the tool most often used takes the labeled transition system along with a set of diagrams, and produces input for the model checker NuSMV [11].

Consider the web server diagram introduced in Section 3.1 and the policy configuration files in the NSA SELinux release.² In this case, we use the diagram

$$\sigma_0 \xrightarrow{\lambda_0} \sigma_1 \xrightarrow{\lambda_1} \sigma_2 \quad [\sigma_e; \lambda_e] \quad \text{where}$$

$$\begin{aligned} \sigma_0 &\text{ is } t = \text{user_t} \\ \lambda_0 &\text{ is } [true]^+ \\ \sigma_1 &\text{ is } t = \text{httpd_admin_t} \\ \lambda_1 &\text{ is } [c = \text{file} \wedge p \in \{\text{write}, \text{append}\}]^+ \\ \sigma_2 &\text{ is } t = \text{httpd_sys_script_t} \\ \sigma_e &\text{ is } false \\ \lambda_e &\text{ is } false \end{aligned} \tag{5}$$

To see if the default policy meets the security goal expressed by the diagram, we created the labeled transition system from the policy configuration files. We then encoded the diagram in a textual syntax, and used it and the labeled transition system to create input for NuSMV.

Figure 5 contains an excerpt of the output from NuSMV. It shows that the security goal is not met by the policy files. Our analysis shows that information can flow from the user type `user_t` to the HTTPD system scripting type `httpd_sys_script_t` via a path that includes using the action `netif udp_recv`.

The security goal is met by the sample policy files in the SELinux release if the diagram in Equation 5 has the following exception for security contexts σ_e :

$$r = \text{sysadm_r} \vee r = \text{system_r}$$

The exception σ_e means that any path that contains a type associated with the roles `sysadm_r` or `system_r` is ignored, and not considered a violation of the security goal. These roles are associated with system processes manipulating low-level resources. Such processes can alter many aspects of the system, but are run only by purportedly trustworthy system administrators.

The diagram in Figure 4 can be used to specify a security goal for the e-commerce processing system in Figure 1, however, once again, the security context exception above must be added. The modified diagram expresses a security goal that is satisfied by policy files modified to include the additional types and permissions described in Section 2.3.

For policy files based on the ones in the release, NuSMV execution typically requires about 150MB of store, and about 10 minutes of CPU time on a 1GHz Intel GNU/Linux laptop.

²We have used release 2003040709 of April 2003 throughout this section.

```

-- specification
!(t = user_t
  & E[t != httpd_admin_t U t = httpd_sys_script_t
    & EF (k = TRUE & t = httpd_sys_script_t)])
is false
-- as demonstrated by the following execution sequence
-> State 1.1 <-
t = user_t
r = system_r
u = system_u
c = netif_c
p = rawip_send_p
k = 1
-> State 1.2 <-
t = netif_ipsec2_t
r = object_r
u = jdoe_u
p = udp_recv_p
-> State 1.3 <-
t = dpkg_t
r = system_r
u = system_u
c = fifo_file_c
p = append_p
-> State 1.4 <-
t = httpd_sys_script_t
r = object_r
u = jdoe_u
c = netif_c
p = accept_p

```

Figure 5: HTTPD Security Goal Failure

5 Rigorous Automated Security Management

In this paper, we have presented a systematic way to analyze the information flow goals achieved by an SELinux system. A formalization of the access control mechanism of the SELinux security server together with a labeled transition system representing an SELinux configuration provides our framework. Security goal statements in linear temporal logic provide a clear description of the objectives that SELinux is intended to achieve. We use model checking to determine whether security goals hold in a given system.

The approach used in developing these formalizations and analysis methods has been used in other security management contexts over the past decade, most recently under the name *rigorous automated security management* [3, 4, 5, 6]. This method front-loads the contribution of formal methods to problem-solving. The focus is on modeling devices, their behavior as a function of configurations, and the consequences of their interactions. A class of practically important security goals must also be expressible in terms of these models.

These models suggest algorithms taking as input information about system configuration, and returning the security goals satisfied in that system. In some cases, although not as yet in the case of SELinux, we can also derive algorithms to generate configurations to satisfy given security goals. The formal models provide a rigorous justification of soundness. By contrast, algorithms are implemented as ordinary computer programs requiring no logical expertise to use. Resolving practical problems then requires little time, and no formal methods specialists. Rigorous automated security management consists of four steps.

Modeling Construct a simple formal model of the problem domain. In this paper, we have seen the formalization of the access control mechanism of the SELinux security server, and the transition relation of an SELinux security policy.

Security Goals SELinux is intended to achieve *information flow* security goals. These take the forms given in Equations 2 and 3.

Goal Enforcement The security goals and underlying model must be chosen so that there is an algorithm that, given a system as represented in the model, and a particular goal statement of one of the selected logical forms, determines whether the system satisfies that goal. In the SELinux system, model checking provides our assurance.

Implementation Having defined and verified one or several goal enforcement algorithms, one writes a program to check goal enforcement. The inputs to this program consist of goal statements that should be enforced, and system configuration information. In this paper, we have discussed an implementation based on NuSMV and mentioned our own specially adapted BDD software.

For systems such as SELinux, formal models of access control configuration and checking reasonable security goals are tractable. A combination of this

formal model and an appropriate algorithm has led to automatic tools for the verification of security properties in an SELinux system. While much future work remains, we believe this approach to be an important step toward increasing the usefulness of secure operating systems.

References

- [1] David E. Bell and Leonard J. LaPadula. Computer security model: Unified exposition and Multics interpretation. Technical Report 75-306, ESD, June 1975.
- [2] Department of Defense trusted computer system evaluation criteria. DOD 5200.28-STD, December 1985.
- [3] Joshua D. Guttman. Filtering postures: Local enforcement for global policies. In *Proceedings, 1997 IEEE Symposium on Security and Privacy*, pages 120–29. IEEE Computer Society Press, May 1997.
- [4] Joshua D. Guttman. Security goals: Packet trajectories and strand spaces. In Roberto Gorrieri and Riccardo Focardi, editors, *Foundations of Security Analysis and Design*, volume 2171 of *LNCS*. Springer Verlag, 2001.
- [5] Joshua D. Guttman and Amy L. Herzog. Rigorous automated network security management. Submitted for publication, June 2003.
- [6] Joshua D. Guttman, Amy L. Herzog, and F. Javier Thayer. Authentication and confidentiality via IPsec. In D. Gollman, editor, *ESORICS 2000: European Symposium on Research in Computer Security*, number 1895 in *LNCS*. Springer Verlag, 2000.
- [7] P. Loscocco and S. Smalley. Integrating flexible support for security policies into the Linux operating system. In *Proceedings of the FREENIX Track of the 2001 USENIX Annual Technical Conference*, 2001.
- [8] P. Loscocco and S. Smalley. Meeting critical security objectives with security-enhanced Linux. In *Proceedings of the 2001 Ottawa Linux Symposium*, 2001.
- [9] P.A. Loscocco, S.D. Smalley, P.A. Muckelbauer, R.C. Taylor, S.J. Turner, and J.F. Farrell. The inevitability of failure: The flawed assumption of security in modern computing environments. In *Proceedings of the 21st National Information Systems Security Conference*, pages 303–314, October 1998.
- [10] National Security Agency. Security-enhanced Linux. At URL <http://www.nsa.gov/selinux/index.html>, April 2003.
- [11] NuSMV: a new symbolic model checker. URL <http://sra.itc.it/tools/nusmv>, 2001.

- [12] A. W. Roscoe and M. H. Goldsmith. What is intransitive noninterference? In *12th IEEE Computer Security Foundations Workshop*, pages 228–238. IEEE CS Press, June 1999.
- [13] Andrei Sabelfeld and Andrew C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communication*, 21(1):5–19, January 2003.
- [14] John C. Wray. An analysis of covert timing channels. In *Proceedings, 1991 IEEE Symposium on Research in Security and Privacy*, pages 2–7. IEEE Computer Society, May 1991.

Contents

1	Introduction	1
2	An SELinux Model	3
2.1	Underlying Ideas of SELinux Access Control	3
2.2	SELinux Access Control Relations	5
2.3	Syntax and Semantics for the Configuration File	5
2.4	The Authorization Relation	7
2.5	The Information Flow Relation	8
3	Security Goals	9
3.1	Visualizing Causal Chains: Diagrams	10
3.2	Formalizing Diagrams in Linear Temporal Logic	11
3.2.1	Order Assertions	12
3.2.2	Event Assertions	12
3.2.3	Exceptions	13
4	Goal Enforcement and Implementation	13
5	Rigorous Automated Security Management	16