Information Flow in Operating Systems: Eager Formal Methods*

Joshua D. Guttman, Amy L. Herzog, and John D. Ramsdell

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

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 [6]. Second, the prime security model [2] was oriented toward preventing disclosure in multi-level secure systems [1], and this required 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 [10].

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 [4, 5].

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. The programs manipulating the resources directly must be trustworthy; direct manipulation by Trojan horses is not our concern. The core goals are protecting the confidentiality and integrity of these resources.

To preserve integrity, each causal chain of interactions leading from untrusted sources to sensitive destinations must traverse a program considered trusted to filter transactions. 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. In both cases, the security goal is an *information flow goal*. Each says that information flowing between particular

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

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

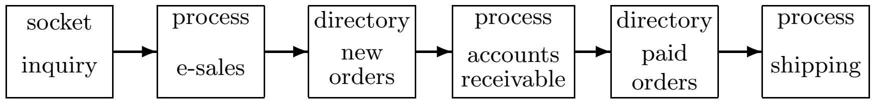

Fig. 1. An E-Commerce Processing System

network socket at the left. An e-sales program ensures that the 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 meanings to these diagrams 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 an example of the "eager formal methods" approach (Section 5, cf. [3]).

2 An SELinux Model

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 be allowed. 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*, 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 specify that collections of users should be permitted to execute corresponding collections of programs. The main purpose of the *user* component is to specify what roles that user is permitted; the main purpose of the *role* is to specify what types of processes those users are permitted to execute.

The main component is the type, accounting for at least 22,000 out of the 22,500 access control statements in the example policy file contained in the distribution. The type is used to specify the detailed interactions permitted between processes and other resources. Each type specification determines some actions that are allowed; 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 specifies a set of process types δ, a set of resource types τ, and a set of class-permission pairs γ. If a process whose type is in δ requests an action with class-permission pair in γ against a resource with type in τ, then that request is authorized.

Since we abstract from auditing and other issues that do not affect information-flow security goals, the configuration file defines five relations of interest. Each relation is built up by statements contained in the same configuration file.

– $\alpha(t_1, t_2, c, p)$ is the relation built up by the `allow` statements. It holds if $t_1 \in \delta$, $t_2 \in \tau$, and $\langle c, p \rangle \in \gamma$ 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.

A type t is called a *domain* if t is the type of any process. There is a distinguished role r_o such that $\rho(r_o, t)$ and $\mu(u, r_o)$ whenever t is not a domain.

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

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:

$$\begin{array}{llll} & \alpha(t_1, t_2, c, p) & & \\ \wedge & \rho(r_1, t_1) & \wedge & \rho(r_2, t_2) \\ \wedge & \mu(u_1, r_1) & \wedge & \mu(u_2, r_2) \\ \wedge & \chi_{c,p}(t_1, r_1, u_1;\ t_2, r_2, u_2) & & \\ \multicolumn{4}{l}{\wedge \text{ if } c = \texttt{process} \wedge p = \texttt{transition} \text{ then } \alpha_\rho(r_1, r_2).} \end{array}$$

This relation $\mathit{\Delta}_{c,p}(t_1, r_1, u_1;\ t_2, r_2, u_2)$ is the SELinux authorization predicate.

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

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

or else

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

$\mathit{\Phi}_{c,p}(t, r, u;\ t', r', u')$ 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 $\mathit{\Phi}_{c,p}(t, r, u;\ t', r', u')$, 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 $\mathit{\Phi}_{c,p}(t, r, u;\ t', r', u')$; 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 iterates 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

Core goals in the SELinux system are protecting the confidentiality and integrity of sensitive resources on the system. 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 [8].

Consider the case of raw disk access. Since this bypasses other access controls, a system administrator would want only specific administrative programs to access it directly. 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`.

Information flow goals can be used to ensure more complicated causal chains as well. Returning to the e-commerce example in Figure 1, we have 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 `sales_socket_t`; after being checked they are written to a file with type `new_order_type`. The accounts receivable program has type `acct_recv_t`, and writes out paid orders to files with type `paid_orders_t`, which are readable by a process having type `shipping_t`.

3.1 Visualizing Causal Chains: Diagrams

While it is easy to reason about the simple causal chain protecting raw disk access, our e-commerce example highlights the need for an alternate way of expressing more complicated 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. 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 pattern 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; we refer to these sets by symbols such as σ_i. Second, one may characterize what actions or events may transfer information from one context to the next; we refer to these sets by symbols such as γ_i. Members of γ_i are SELinux class-permission pairs.

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 (as in our raw disk access example). However, we may also be interested in potentially longer paths between contexts. In the e-commerce example, 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 indicate the two types of arrows by decorating γ_i with a superscript 1 (single event) or + (iterated events). Let λ_i be a label of one of the forms γ_i^1 or γ_i^+. Finally, we may specify exceptions, although we will return later to fill in this detail. Ignoring exceptions, we can write an information flow policy goal in the following visual form:

$$\sigma_0 \xrightarrow{\lambda_0} \sigma_1 \xrightarrow{\lambda_1} \cdots \xrightarrow{\lambda_{n-2}} \sigma_{n-1} \xrightarrow{\lambda_{n-1}} \sigma_n \tag{1}$$

Note the similarity between this form and our e-commerce example in Figure 1. In that example, σ_1 would be `sales_socket_t`, and σ_6 would be `shipping_t`, and so forth.

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, and that there must be just one such event if the decoration is 1 and may be more events if the decoration is +.

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).

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:

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

The operator $\mathcal{R}$ ("releases") asserts that its right hand operand is true and remains true until its left hand operator 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.

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 the decorations are all 1 rather than +, this leads to the formula

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

$\mathcal{X}\,\phi$ 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.... If the decorations are all +, then we want to say that a γ_i occurs, and then γ_is continue until a σ_{i+1} is reached, and so on:

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

$\phi\, \mathcal{U}\, \psi$ 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

$$H \Rightarrow \gamma_0\; \mathcal{O}_0\; (\sigma_1 \wedge (\gamma_1\; \mathcal{O}_1\; (\sigma_2 \ldots))). \tag{3}$$

When the label λ_i is of the form γ_i^1, then $\phi\, \mathcal{O}_i\, \psi$ is defined to be $\phi \wedge \mathcal{X}\, \psi$. When the label λ_i is of the form γ_i^+, then $\phi\, \mathcal{O}_i\, \psi$ is defined to be $\phi \wedge \mathcal{X}(\phi\, \mathcal{U}\, \psi)$.

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.

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. [9]) 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.

4 Goal Enforcement and Implementation

We have written software that reads and analyzes an SELinux configuration file. In effect, it constructs the information flow relation $\Phi_{c,p}$, and defines the initial states where $\rho(r,t) \wedge \mu(u,r)$. It may then either construct Binary Decision Diagrams (BDDs) directly to represent these values, or alternatively emit a NuSMV specification defining them [7]. In addition, given values σ_i and λ_i defining a diagram of the form of Formula 1, the software emits a specification for NuSMV to check. We find (as of this writing) that NuSMV can successfully answer a preliminary set of queries of this form, although we have primarily checked never-allow assertions in which the operator is and there are only two states of interest. Execution typically requires about 150MB of store, and answers a list of a couple of dozen never-allow queries in about 10 minutes of CPU time on a 500MHz Intel Linux machine.

A direct translation to BDDs has also been implemented. Specially designed algorithms to resolve security goals of the form (1) will be preferable to NuSMV for several reasons. First, time-consuming parts of the construction can be performed once, and maintained in memory to quickly resolve a variety of interactively posed security goals. Second, information can be returned to the user in a form more adapted to this application domain. And third, the algorithms can return more informative results, for instance defining what subset of the states in a set σ_i are in fact traversed in Formula (1).

5 Conclusion: Eager Formal Methods

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 techniques 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, under the name *Eager Formal Methods* [3]. Eager formal methods 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. Eager formal methods 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 briefly discussed two potential implementations; based on NuSMV and 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 use of the eager formal methods approach to be an important step toward increasing the use 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 and Amy L. Herzog. Eager formal methods for security management. In *Proceedings, VERIFY '02*, pages 91–101, July 2002.
4. 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.
5. P. Loscocco and S. Smalley. Meeting critical security objectives with security-enhanced Linux. In *Proceedings of the 2001 Ottawa Linux Symposium*, 2001.
6. 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.
7. NuSMV: a new symbolic model checker. URL `http://sra.itc.it/tools/nusmv`, 2001.
8. 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.
9. Andrei Sabelfeld and Andrew C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communication*, 21(1):5–19, January 2003.
10. 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.