

Achieving Security Goals with Security-Enhanced Linux

Amy L. Herzog, Joshua D. Guttman

The MITRE Corporation
202 Burlington Rd.
Bedford, MA 01730 USA
{aherzog, guttman}@mitre.org

February 5, 2002

1 Introduction

Access control is a vital security mechanism in today's systems. It is used on nearly every computer system to prevent a variety of mishaps, including keeping users from misusing resources (maliciously or not), protecting information, and containing damage from attacks. Frequently used for basic security even when other measures are absent, access control mechanisms can be a very attractive target for attackers, and are of great concern to the security administrator.

Evidence of this can be seen by the proliferation of buffer overflow attacks in the security world today. Typically, these attacks are launched against software that requires some privilege to run. They take advantage of inadequate bounds checking in software to exploit the greater privilege of the target program, and usually aim to produce a root shell for the attacker.

As software is not reliably well-written, the importance of mandatory access control is becoming more apparent. [10] Systems with mandatory access control mechanisms grant full authority over the policy only to the system administrator. Many implementations also allow for a much more fine-grained access control policy than standard discretionary access control systems. One mandatory access control system, Security-Enhanced Linux [7], combines the power of type enforcement with the Linux operating system.

The power available via Security-Enhanced Linux (hereafter SE Linux) comes at a price, however. The security policy dictating the behavior of the system is lengthy and complex. It is extremely difficult both to generate and analyze the policy by hand. Policy generation and analysis are currently open and active research areas in the SE Linux community. However, before policies can be successfully analyzed, it must be clear what properties they should possess. This

is a large problem facing security administrators today, who frequently do not know what they are trying to protect (or how to protect it).

The work presented here is part of an ongoing effort to design and implement a rigorous SE Linux policy analysis toolkit. In this paper, we outline two main types of security goals that can be achieved via access control mechanisms. These goals are presented in the context of SE Linux systems, but would be applicable in any access control environment. These goals are in a form both easily understandable and easily translatable into a form ready for analysis.

In Section 2, we provide a brief introduction to Mandatory Access Control mechanisms and SE Linux in particular. We then describe our current work on policy analysis tools (Section 3) before moving on to a description of our general security goals in Section 4. We end with a look at future directions (Section 5).

2 Background Information

Many security efforts in the computer industry today focus on application-level tools aimed at providing a specific security service, e.g. confidentiality or authentication. However, these security mechanisms can only be as secure as the operating system underlying them. [10] Operating system security is critical to overall system security; if an application-level security mechanism can be tampered with or bypassed it does no good at all.

A distinguishing feature of a more secure operating system lies in *mandatory security*, defined in the sense of [10, 2] to be any security policy where the definition of the policy logic and assignment of security attributes is tightly controlled by a security policy administrator. A *mandatory access control* policy is the component of mandatory security which concerns us here. A mandatory access control (or MAC) policy specifies how subjects may access objects under the control of the operating system. There have been several implementations of MAC systems, including DTOS [9, 3] and Flask [11]. SE Linux, the system we will discuss here, is a Linux implementation of the Flask microkernel. To provide background for our current work, we will discuss the architecture of the SE Linux system, and then describe its policy.

2.1 SE Linux System Architecture

One major advantage of SE Linux is that security policy logic is cleanly separated from enforcement mechanisms. Thus, while a specific security policy model (a powerful combination of Type Enforcement and Role-Based Access Control) is implemented, a different model could be implemented with minimal change to the enforcement mechanisms.

In SE Linux, both security policy specification/decision and policy enforcement are implemented as kernel subsystems. The security policy decision logic is encapsulated in the *Security Server*. The Security Server makes labeling and access decisions in response to policy-independent requests. Policy enforcement is accomplished via a set of kernel subsystems called *Object Managers*.

The Security Server reads in and stores the security policy of the system at startup. It also reads in information about the system itself; every object in the system (file, process, socket, etc.) is assigned security relevant information. From the security policy and information about the system, the Security Server builds an access matrix where each element of the matrix determines the allowable accesses between a pair of system objects.

There is one Object Manager for each class of object in the system (files, processes, sockets, and so on). When an access control decision must be made, the Object Manager passes the relevant security label information to the Security Server and waits for a decision. The Security Server then examines its access matrix and passes a decision back to the Object Manager, which enforces the decision. In this way, the Object Manager can enforce policy decisions without requiring access to the details of the policy itself.

2.2 SE Linux Security Policy

The security policy currently included with SE Linux is a combination of *Type Enforcement* and *Role-Based Access Control* (RBAC). Type Enforcement (TE) is the main access control mechanism implemented. It focuses on the relationships between two types of system components—active ones (the *domains* of a system), and passive ones (generally termed *types*). When a domain attempts to affect a type, access is looked up in a matrix; the contents of a [domain *i*, type *j*] cell in the matrix determine exactly the sorts of operations permitted. The SE Linux implementation removes the explicit distinction between domains and types, and the Security Server simply constructs an access matrix of [type *i*, type *j*] access permissions (as mentioned above). Readers interested in a more thorough discussion of Type Enforcement are referred to [1].

Role-Based Access Control (RBAC) takes a broader approach; it involves defining the different *roles* relevant to a system, and then assigning privileges to the roles. After roles and privileges are defined, users are simply assigned roles. A longer discussion of RBAC can be found in [4].

SE Linux actually specifies its RBAC policy using Type Enforcement; roles are given access to types, which are then assigned certain permissions (in ways described below). The only major power given directly to roles is the ability to transition into other roles; no access permissions are ever granted directly to roles. For the remainder of the paper, therefore, we shall focus only on the Type Enforcement access mechanism.

All relevant security attributes of an object in the system are recorded via a *security context*, which consists of the object's *identity*, *role*, and *type*. Changes to the identity (usually a user identity) are tightly controlled (the policy configuration only allows certain programs, such as `login`, to change it). The role listed codifies the RBAC policy, and the type the TE policy.

As mentioned above, permissions are granted to types. There are a large number of permissions, including the familiar `read`, `write`, and `append` as well as many not-so-familiar, each specific to a certain class of object in the system. Permissions are granted in a number of different ways, the most obvious and

common being an `allow` statement. The syntax of the statement is: `allow type_1 type_2:class {permissions}`, giving objects of type `type_1` the ability to perform actions of type `{permissions}` on objects of type `type_2` and class `class`. An example follows:

```
allow syslogd_t devlog_t:sock_file { create };
```

(The above statement grants `syslogd` the ability to create the `/dev/log` socket file.) In addition to the `allow` statement, there are several other statements granting power to various types.¹

The access matrix of the Security Server is constructed from all of the rules in the policy files (including the `allow` statements). Choosing a simple example, when the security server is called upon to make a decision about security contexts c_1 and c_2 and the read permission r , it will examine the entry for the starting type $c_1(t)$, the target type $c_2(t)$, and determine if `read` is among the allowed permissions. It reports the allow or deny to the requesting Object Manager.

The flexibility inherent in these permission-granting statements is impressive; by increasing the number of types in the system (with various permissions granted to each via the `allow` statements) one can specify access control in the system arbitrarily narrowly. However, this flexibility comes at the cost of simplicity—the more types one has, the greater number of statements required. This causes SE Linux policy files to be quite confusing to read and understand.

3 Current Work

The complexity of a typical SE Linux security policy is daunting; the policy included with the distribution contains approximately 54,000 distinct rule statements, around 100 distinct permissions, and 700 defined types (that a file, process, etc. could be labeled). Policies this long are bound to be complex; hand analysis is impossible due to the large number of potential interactions between various rules. Even simple query-based tools which present raw data about the security policy are not completely helpful; interpreting data is a challenge even to the most aware security administrator. These issues highlight the need for more formal and rigorous policy analysis tools to help the administrator determine what policy is being enforced. Work on such a rigorous policy analysis tool is ongoing; we will provide a brief outline here in order to provide context for the security goal specification below.

We represent each SE Linux security policy as a graph, where nodes are the various security contexts in the system, and all permissions are represented by directed edges. The edges are directed since all permissions can be viewed in terms of information flow from one context to another: if a process with context c_1 can write to a file with context c_2 , information is flowing from c_1 to c_2 . If a process with context c_1 can read from a file with context c_2 , information is

¹A full discussion of the policy configuration can be found in [6].

flowing from c_2 to c_1 . If both of these permissions were allowed in a particular SE Linux policy, the graph representing the policy would have directed edges from c_1 to c_2 , and vice versa. (The edges are also labelled with the class and permission the edge represents—in the case of a process c_1 writing to a file c_2 , the edge would be labelled with the pair $\langle file, write \rangle$.)

This method of security policy representation is simple and powerful. Once the translation into a graph is complete, increasingly complex analysis questions can be answered with more powerful graph analysis tools—from simple traversal algorithms to regular expressions to model checking. Given a graph representing an SE Linux security policy, simple analysis queries (can type i write or append to type j ?) can be easily answered by graph traversal. Security goals can either be translated into regular expressions or statements in temporal logic as appropriate; standard analysis methods can then be employed to determine if the relevant subset of policy is a subset of the regular expression, or if the policy graph satisfies the temporal logic statements.

Even this powerful machinery, however, has limitations—before goals *can* be analyzed, they must be stated. The following section describes general security goals that are achievable with MAC mechanisms, and which are also easily expressed as regular expressions, making them convenient and appropriate for our policy analysis work.

4 Access Control Security Goals

As we have seen, SE Linux is a powerful and yet very complex tool for achieving computer system security. Its mechanisms allow an administrator complete control over the behavior of the system, from truly restricting processes to their least required privilege, to protecting data and keeping processes reliably separate. In fact, it is so powerful that a security administrator could easily be confused not only about what a particular policy enforces, but what it is even *possible* to enforce with Mandatory Access Control.

Many security administrators have a fairly hazy notion of ‘security’ in relation to their systems. When asked what they would like out of a secure system, administrators may desire containment of a break-in, or protection from buffer overflow attacks. In this section, we will approach that question from a different direction; we wish to define and describe the general security goals one can achieve with a Mandatory Access Control policy such as the one implemented in SE Linux. We will also provide insight into why they are likely to be achievable, and provide examples of how these goals fit easily onto the rigorous policy analysis framework described above. We discuss problems Access Control seems able to solve and present a list of specific security objectives before introducing our two general goal formats in detail.

4.1 Motivation

Access Control mechanisms are frequently employed to try and prevent or contain damage from common security attacks. Running daemons as `nobody` or another daemon-specific user identity (rather than the traditionally powerful `root` user) when possible, restricting traditional UNIX read and write permissions on sensitive files, etc. are all ways Discretionary Access Control mechanisms have been used to enhance the security of a system. However, as discussed in [10], such mechanisms are inherently flawed. Mandatory Access Control, with its absence of a `root` uid and tightly-controlled policy, helps significantly reduce the success of attacks. However, when analyzing their complex policies, detailed security objectives are required. In [8], six critical security objectives are outlined as appropriate to achievement via an SE Linux security policy:

1. Limiting raw access to data
2. Protecting kernel integrity
3. Protecting system file integrity
4. Confining privileged processes
5. Separating processes
6. Protecting the administrator domain

While these are certainly important security objectives, we note that this list, though broad, is not necessarily complete. It is conceivable that some specific system administrator goals may not fit into one of these six categories, and yet may be achievable via an SE Linux policy (trusted pipelines, for example).

Upon examining the solutions to these goals, patterns begin to emerge. In the case of the first three goals and the last, the strategy for achievement (listed in [8]) is to determine the resource requiring protection, enumerate the (small) list of ‘authorized’ users of the resource, and aggressively limit transitions into the security context of these authorized users. To confine privileged processes (Goal #4), one defines a separate domain for the process in question, and then restricts access permissions for that domain to the least-privilege required. For separating processes from one another (Goal #5), an instance of each of the above strategies is required. This leads us to our two access control security goals.

4.2 General Security Goals

Examining the six security objectives listed above, we see that there are actually two core types of security goal being discussed; namely, those of resource protection and process isolation. (Sometimes, as in the case of separating processes, these will be used in tandem.) We use the term *resource* to represent a security context in the system; processes, files, sockets, and the like all have

unique security contexts and can be viewed as resources in this context. Resources and system objects are the same in the sense that they both represent security contexts; we use different terms to highlight their different functions in the system.

The general form for resource protection goals is as follows:

Direct information flow into resource B only comes from system object(s) A_i .

Direct information flow in this form represents some regular expression of system permissions where the information ultimately flows from the contexts of the A_i into the context of B (such as a simple write, or a write followed by an append, or a more complicated expression). The term *direct* indicates that there is information flow in a direct sense, rather than through some kind of covert channel. (While information leaks through covert channels can be serious, the SE Linux policy analysis framework will at first focus on more direct methods of information flow.)

Referring back to the examples discussed in [8] and outlined above, specific examples of resource protection goals are:

- Direct information flow into the `/boot/` directory should only come from the administrator domain and `rc` scripts. (Goal #2)
- Write access into the file `/etc/shadow` should only come from the administrator and the `useradd` program domains. (Goal #3)
- Transitions into the administrator domain should be restricted to the three users authorized for system maintenance, and only through the `login` program or a special `newrole` program. (Goal #6)

Strategies listed in [8] for the use of SE Linux policy to achieve these aims indicates that this type of security goal is easily achievable via MAC mechanisms. Transformation from the general security goal statement form into a regular expression to be checked automatically is also quite straightforward.

Consider as a simple first example the second resource protection goal listed above—that write access into the file `/etc/shadow` should come from the administrator domain and the `useradd` program. In terms of paths through a policy graph, this means that the last step in any path resulting in a write to the file `/etc/shadow` must come from either the administrator domain (which has the type `sysadm_t`) or the domain held by the `useradd` program (which has the type `admintool_t`). This can be depicted visually as

$$((\text{sysadm_t} \cup \text{admintool_t})(w \cup a)(\text{shadow_t}))$$

where $w \cup a$ stands for the ‘write’ and ‘append’ operations, and the notation `n_t` actually represents any security context with type `n_t`. Resource protection goals are a useful security goal format; they are intuitive to specify for security administrators, and can easily be expressed in formats useful for formal policy analysis.

Rather than protecting a given resource, process isolation goals aim to keep processes from overstepping their bounds, and hence explicitly list all resources a given process should be allowed to affect. They have a similar form to resource isolation goals. The general form of a process isolation goal is:

Process A should have direct information flow only into resource(s) B_i .

The meaning of direct information flow and resources is exactly as with resource protection goals. A *process* in the system is also a security context, but represents the ‘active’ system objects.

Again referring back to goals discussed in [8] and outlined above, a specific process isolation goal follows here:

- The `sendmail` daemon should have direct information flow only into the mail queue directory, `/var/spool/mail`, and `/etc/mail`.² (Goal #4)

Again, strategies listed in [8] indicate that process isolation goals are easily achievable via MAC mechanisms. Process isolation will be useful and relevant whenever the administrator would like to confine a process to its least privilege, whether that program is `sendmail`, the `apache` daemon, an FTP server, or other potentially dangerous program. Again, transformation from the general statement into an analysis form is straightforward. Considering the example given above—that the `sendmail` daemon should have direct information flow only into the mail queue directory, `/var/spool/mail`, and `/etc/mail`:

$((\text{sendmail.t}(\dot{\rightarrow} (\text{mailspool.t} \cup \text{mailqueuedir.t} \cup \text{mailfile.t}))?)?)$

(Here, $\dot{\rightarrow}$ matches any string of length one with information flow in the direction indicated by the arrow.) Again, the notation `n.t` indicates any security context with type `n.t`. We see also that this goal format is useful both in the ease of specification and in the ease of transition to a formal analysis form.

It should be noted that some desires on the part of a security administrator will result in the composition of goals of these types. One example is the goal discussed in [8] and outlined above of separating processes from one another (Goal #5). It is natural to wish to protect processes in one domain from interfering with processes in another domain. However, to achieve such a desire two of the security goals listed above are required; one must limit transitions into security contexts able to affect `/proc`, which limits direct interference and is an example of resource protection; another must restrict access permissions for the two processes one wishes to separate, which prevents indirect interference and is an example of process isolation. Through a potential combination of security goals of the two types, any security desire of an administrator is easily specifiable and easily transformed for analysis purposes.

²Of course, `sendmail` can in fact do more than these simple examples: send outgoing traffic to any port it wants, accept information from the SMTP port, etc. We restricted our focus for illustration; the actual policy available at <http://www.nsa.gov/selinux/> and [5] have complete information

We present a slightly more complicated example, to illustrate the power of our analysis approach and goal formats—that of a trusted pipeline scenario.

4.3 A Trusted Pipeline Example

As a more complex example than the simple scenarios we have previously considered, consider the following: A security administrator would like to allow individual users to use programs such as `procmail` to perform mail filtering, but is concerned about viruses and would like to ensure that mail directories cannot be altered except by `procmail`, *after* the email has been filtered. Thus, one would like to ensure that for each mail message, the following sequence of trusted actions occurs:

- `sendmail` receives the incoming mail from port 25;
- the mail gets deposited for virus scanning;
- after successful scanning, the virus checker passes the mail to `procmail`;
- `procmail` can then deliver the mail to users' home directories.

The administrator would like to ensure, then, that the only way for information to get from incoming TCP port 25 to users' home directories is to follow this trusted pipeline. The security goal involved here is one of resource protection. In our discussion of this goal, we will render the our expression somewhat cleaner by assuming that programs output data directly to one another (rather than writing to and reading from input files). The security goal statement follows:

Any information flow into the `homedir.t` comes via the path
`<smtp_in.t, sendmail_exec.t, virusscan.t, procmail.t>`.

This goal results in a more involved expression than those previously discussed, and temporal logic is natural for its analysis. Since we are truly interested in any information flow, let \mapsto stand for the union of all file permissions with the appropriate information flow; that is, $\mapsto = write \cup append \cup create \cup lock \cup unlink \cup unlink \cup link \cup rename$. We give the security goal as a state formula for the state `smtp_in.t`. The notation A indicates that whatever follows is *Always* true, and applies to paths through the graph; G implies that a state property is *Globally* true, and U stands for *Until*.

$$\begin{aligned}
 &A(G(\neg \text{homedir.t}) \\
 &\quad \vee ((\mapsto \vee \text{smtp_in.t}) U \\
 &\quad\quad ((\text{sendmail_exec.t}) \wedge (\mapsto \vee \text{sendmail_exec.t}) U \\
 &\quad\quad\quad ((\text{virusscan.t}) \wedge (\mapsto \vee \text{virusscan.t}) U \\
 &\quad\quad\quad\quad ((\text{procmail.t}) \wedge (\mapsto \vee \text{procmail.t}) U \\
 &\quad\quad\quad\quad\quad (\text{homedir.t}))))))
 \end{aligned}$$

A model checker would be used to determine if a specific policy graph (provably) satisfied this state formula. This kind of automation represents a large improvement over simple query or hand-analysis of policies; a security goal as complicated as this would be nearly impossible to verify in those ways.

5 Conclusion

As we have seen, the SE Linux Mandatory Access Control System makes great strides toward improving the overall security of computer systems. It is not without its problems, however; the power inherent in the Type Enforcement-based access control policy comes at the cost of that policy being complex. Hence, policy analysis tools are extremely important, giving the security administrator the ability to know beyond a doubt exactly what security guarantees her system gives.

However, guarantees cannot be given until the security goals themselves are understood. In this paper, we have presented an outline of our SE Linux policy analysis methods, and given a detailed description of the two security goals suitable for achievement with a MAC system. These goals, resource protection and process isolation, are easy for administrators to specify, and equally easy to translate into our analysis framework. This provides an important step along the path of policy analysis in Mandatory Access Control systems.

Much interesting future work remains. The tool currently under development could be later extended, and used in conjunction with other tools to help automate part of the policy generation process (also currently somewhat tedious). One can envision a suite of policy generation and analysis tools, easing the administrative burden of MAC systems and bringing significantly stronger security to the average computer system.

References

- [1] W. E. Boebert and R. Y. Kain. A practical alternative to hierarchical integrity policies. In *Proceedings of the Eighth National Computer Security Conference*, 1985.
- [2] Secure Computing Corporation. Dtos general system security and assurance assessment report. Technical report, Technical Report MD A904-93-C-4209 CDRL A011, June 1997. (<http://www.securecomputing.com/randt/HTML/dtos.html>).
- [3] E. Schneider D. Olawsky, T. Fine and R. Spencer. Developing and using a policy neutral access control policy. In *Proceedings of the New Security Paradigms Workshop*, September 1996.
- [4] David Ferraiolo and Richard Kuhn. Role-based access control. In *Proceedings of the 15th National Computer Security Conference*, 1992.
- [5] Franklin Haskell. Confining sendmail using security-enhanced linux. October 2001.
- [6] P. Loscocco and S. Smalley. Integrating flexible support for security policies into the linux operating system. Technical report, NSA, NAI Labs.

- [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] Spencer E. Minear. Providing policy control over object operations in a mach based system. In *Proceedings of the Fifth USENIX UNIX Security Symposium*, pages 141–156, June 1995.
- [10] P.A. Muckelbauer R.C. Taylor S.J. Turner P.A. Loscocco, S.D. Smalley 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.
- [11] P. Loscocco M. Hibler D. Anderson R. Spencer, S. Smalley and J. Lepreau. The flask security architecture: System support for diverse security policies. In *Proceedings of the Eighth USENIX Security Symposium*, pages 123–139, August 1999.