Analyzing Integrity Protection in the SELinux Example Policy

Trent Jaeger Reiner Sailer Xiaolan Zhang
IBM T. J. Watson Research Center
Hawthorne, NY 10532 USA
Email: {jaegert,sailer,cxzhang}@us.ibm.com

Abstract

In this paper, we present an approach for analyzing the integrity protection in the SELinux example policy. The SELinux example policy is intended as an example from which administrators customize to create a policy for their site's security goals, but the complexity of the model and size of the policy make this quite complex. Our aim is to provide an access control model to express site security goals and resolve them against the SELinux policy. Ultimately, we aim to define a minimal trusted computing base (TCB) that satisfies Clark-Wilson integrity, by first testing for the more restrictive Biba integrity policy and resolving conflicts using Clark-Wilson semantics. Our policy analysis tool, Gokyo, implements the following approach: (1) it represents the SELinux example policy and our integrity goals; (2) it identifies conflicts between them; (3) it estimates the resolutions to these conflicts; and (4) provides information for deciding upon a resolution. Using Gokyo, we derive a proposal for a minimal TCB for SELinux includes 30 subject types, and we identify the work remaining to ensure that TCB is integrity-protected. Our analysis is performed on the SELinux example policy for Linux 2.4.19.

1 Introduction

A goal for many years has been effective mandatory access control (MAC) for UNIX systems. By an *effective* MAC system, we envision that system administrators can define access control policies that guarantee site security goals while enabling the convenient execution of applications. Early MAC policies, such as the Bell-LaPadula secrecy policy [2] and the Biba integrity policy [4], defined clear security goals, but were too restrictive for convenient use for UNIX applications. Commercial operating systems that were extended to meet Orange Book B1 (i.e., MAC plus other features) were not broadly applied (i.e., mainly aimed at government installations). Recent efforts at MAC systems use flexible access control models to achieve convenient use (e.g., DTOS, Flask [17, 21], etc.), but demonstrating that particular security goals have been met is more difficult (and these systems have not been widely used either). Flexible access control models typically result in more complex policies, so it is more difficult to determine if these policies have the desired effect.

The recent addition of the Linux Security Modules (LSM) framework [22] enables the MAC enforcement for the Linux kernel. The LSM framework is designed to be agnostic to the MAC approach, and it has been designed to support modules with flexible MAC models. The most comprehensive and flexible module for LSM is the SELinux module [18]. While SELinux supports a variety of policy models itself, an extended Type Enforcement (TE) model [5] is used for most policy development. An example policy is under development that consists of a set of UNIX service and application policies that each aim to ensure effective operation while preventing security vulnerabilities. The example policy does not define a secure system, but serves as a basis for developing a secure system once the security goals are defined. The extended TE model is rather complex (i.e., consists of a large number of concepts) and the SELinux example policy is large (e.g., 50,000+ policy statements in the `policy.conf` for Linux 2.4.19), so customization of the SELinux example policy to a policy that guarantees satisfaction of system security goals is an arduous and error-prone task.

While the use of a simpler access control model might make it easier to ensure that security goals are met, we believe that this would result in applications failing to run conveniently, and ultimately, the circumvention of these security goals. The comprehensive nature of the SELinux policy model enables flexible tradeoff between application and security goals. For example, the SELinux example policy itself is developed by proposing application policies and refining them based on the policy violations that may be generated. Thus,

the SELinux example policy itself is a direct result of making these trade-offs.

The question is whether a manageable set of effective security goals can be described and verified for SELinux policies. Obviously, it is highly unlikely that the SELinux example policy adheres to a simple high-level policy, such as the two-level integrity model of LOMAC [9]. However, the policy may be sufficiently close to such a policy that the conflicts can be managed (i.e., either a small number or a small number of equivalence classes). If so, then verification may be possible by verifying the general goals and using ad hoc techniques to resolve the conflicts. We have found that this approach holds some promise for application policies, in particular the Apache administrator [12], but do not know whether this can work for the trusted computing base (TCB) subjects in the SELinux policy. Obviously, if we cannot prove that the TCB is integrity-protected, its system cannot be considered secure.

In this paper, we propose a near-minimal TCB for SELinux systems and examine how to verify that this TCB is integrity-protected. First, we define integrity relationships between the TCB subject types and less trusted system and application subject types. Second, we input these constraints into our policy analysis tool, called Gokyo [12], and identify integrity conflicts between the TCB and the system. The Gokyo tool enables flexible expression of conflict sets and their resolution, so our next goal is to determine what resolutions appear feasible for TCB integrity conflicts. Using Gokyo, we classify conflicts into classes based on their likely resolution. Since most resolutions depend on ad hoc information, it is still a manual process to complete the analysis. Using Gokyo, we identify a minimal TCB for the SELinux example policy of 30 subject types, half of which are infrequently-used administration subjects. To use this TCB, 5 sanitization problems must be solved, but we believe that most can be addressed in practice, including the use of Gokyo itself to manage the broad file access rights currently granted to trusted subjects. Ultimately, Gokyo is useful in identifying problems in meeting security goals, classifying these problems, and providing information for resolving them.

The paper is structured as follows. In Section 2, we examine the SELinux extended Type Enforcement model and outline our site security goals for that model, integrity protection of a minimal trusted computing base. In Section 3 we describe our approach to resolving a policy against our integrity protection requirements. In Section 4, we detail the implementation of our analysis using Gokyo and present our analysis results. In Section 5, we present related work, and we conclude in Section 6.

2 SELinux Security Goals

2.1 SELinux Policy Model

While SELinux supports a variety of access control policy models [21], the main focus of SELinux policy development has been an extended Type Enforcement (TE) model [1, 5, 20]. In this section, we provide a brief overview of the SELinux policy model concepts, focusing only on the concepts that are relevant to the analysis that we perform. A number of other concepts are represented in the SELinux extended TE model, such as roles and identity descriptors, that we do not cover here. A detailed description of the SELinux policy model is given elsewhere [20].

The traditional TE model has subject types (e.g., processes) and object types (e.g., files, sockets, etc.), and access control is represented by the permissions of the subject types to the object types. In SELinux, the distinction between subject and object types has been dropped, so there is only one set of types that are object types and may also act as subject types.

The SELinux extended TE model is shown in Figure 1. All objects are labeled with a *type*. All objects are an instance of a particular *class* (i.e., data type) which has its own set of *operations*. A permission associates a type, a class, and an operation set (a subset of the class's operations). Thus, permissions associated with SELinux types can be applied independently to different classes. For example, different rights can be granted to a user's files than to their directories. In fact, since the objects are of different classes, they have different operations. Should the administrator want to give different access rights to two objects of the same class, then these objects must belong to different types.

Permission for a (subject) type to perform operations on a(n) (object) type are granted by the *allow* statement. Any element of the permission relationship can be expressed using this statement, so the expression of least privilege rights is possible. The *dontaudit* statement provides a variation on the basic permission assignment. A combination of allow statements result in a union of the rights specified, whereas a combination of dontaudit statements on the same type pair and class are intersected.

In addition, the extended TE model also has type at-

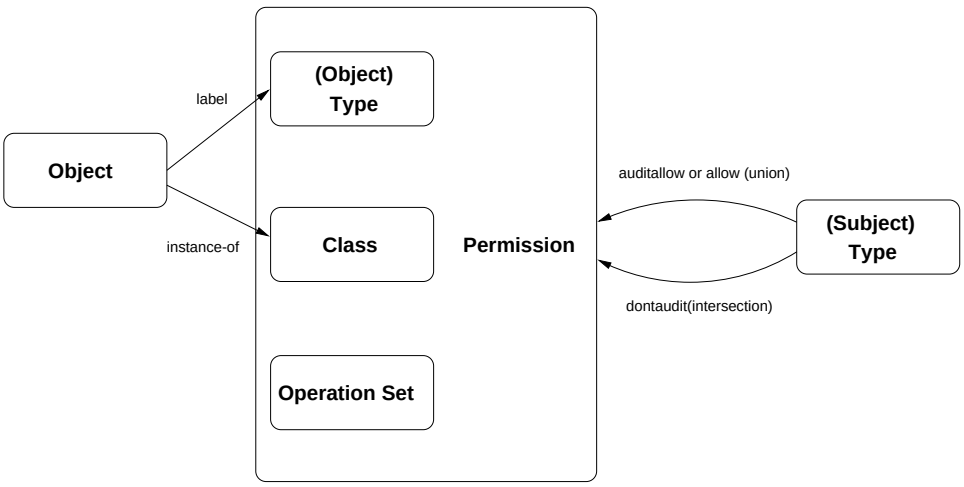

Figure 1: SELinux extended Type Enforcement (TE) policy model basics.

tributes that represent a set of types (i.e., all the types with that attribute assigned). Type attributes enable assignment to multiple types at a time. For example, a permission can be assigned to each subject type with that attribute or a subject can be assigned permission to each object type with that attribute.

Containment is enforced by limiting the permissions accessible to a subject type (as described above), limiting the relabeling of object types, and limiting the domain transitions that can be made by a subject type. Relabel rights are controlled in SELinux by limiting access to *relabelfrom* and *relabelto* operations. As the names indicate, *relabelto* enables objects to be relabeled to that type and *relabelfrom* enables objects of a particular type to be relabeled.

Domain transitions can occur when a subject type executes a new program. Again, SELinux defines an operation, called *transition*, to perform these transitions. A subject type must have a transition permission for the resultant subject type in order to affect a domain transition.

The SELinux model also has statements for *type transition* and *type change*. Type transition statements are used by SELinux to automatically compute transitions, but are not necessary for control (i.e., transition permissions are always necessary). Type change statements alter the type of an object upon access by the specified subject type. Such statements are useful when a system administrator logins in using a user's tty. Type change statements transition the object type of the tty to prevent users from altering input.

In order to simplify the task of expressing policies, the SELinux extended TE model also includes a large number of macros for expressing sets of policy statements that commonly occur together. We do not examine the policy macros in detail because policy analysis requires us to understand the policy at the level of the type enforcement model statements (i.e., which subject types can perform which operations on which object types).

2.2 SELinux Example Policy

The SELinux community is working jointly on the development of UNIX application policies whose composition is called the *SELinux example policy*. The SELinux example policy does not define a secure system, but is intended as input to the development of a custom policy for each site's security goals, commonly called a *security target*. Unfortunately, customization is not simply composition of the policies for the applications of interest. The application policies themselves are somewhat specialized to the environment in which they were developed, and interactions between the policies of multiple applications may lead to vulnerabilities. In general, the composition of policies that are proven secure may not result in a secure system.

The task of customization is further complicated by the size of the example policy and the complexity of the extended TE model described in Section 2.1. The SELinux example policy for Linux 2.4.19 consists of over 50,000 policy statements (i.e., the processed macro statements in `policy.conf`). According to our analysis, this specification represents over 700 subject types and 100,000 permission assignments. We believe that size and complexity of the SELinux example policy

make it impractical to expect that typical administrators can customize it to ensure protection of their trusted computing base (TCB) and to satisfy their site's security goals on this TCB. This may seem obvious to some and may seem insufficiently justified to others, but we will describe a more detailed argument on why we believe this in Section 2.3.

Despite this, we are convinced that the SELinux example policy is valuable to building secure systems, for these two reasons primarily: (1) it provides a flexible enough representation to capture the permissions necessary for UNIX applications to execute conveniently and (2) it provides a comprehensive definition of a reference monitor for UNIX. First, the SELinux example policy is developed per application in a manner that identifies a superset of the permissions required to run an application conveniently while possibly meeting a particular security target. What typically happens is that a proposal is made for an application policy, then this policy is tested by the community when they use the application. Since SELinux reports authorization failures (i.e., the lack of a permission requested), it is much easier to determine that insufficient permissions were assigned than whether a security vulnerability is created. Thus, a verified proposal for least privilege permissions for each application is represented by the SELinux policy. What we need is a better way to test whether our security goals are satisfied, such that conflicts can be identified and addressed.

Second, the SELinux example policy is a comprehensive representation of UNIX access control. The SELinux model aims to comprehensively control access to all classes (i.e., kernel data types) that may be operated upon by a user-level Linux process. There are 29 classes defined in the SELinux example policy. Each class has its own set of operations that are intended to capture all the relevant subtleties in accessing and modifying a class. Given the scope of the SELinux example policy at this granularity, the SELinux example policy provides as precise and comprehensive a repository of UNIX application access control information as exists today. We need to leverage this repository in the development and refinement of security goals, but provide such leverage through higher-level concepts that enable effective management.

2.3 SELinux Security

Unlike early MAC models like Bell-LaPadula [2] and Biba [4], a TE model does not explicitly indicate the security goals of the policy. Thus, the policy implies the security goals of the system. For a TE system, more like an access matrix, we only learn that certain subjects can only perform certain operations on certain objects. The security goals of the policy are not represented at a higher-level than this.

The SELinux model provides an approach by which secrecy and integrity properties may be achieved with least privilege permissions and containment of services [16]. The system administrators create a policy that is restrictive with respect to granting rights that violate secrecy and integrity properties and we use the notions of least privilege and containment to minimize the damage due to compromises where these occur.

From our perspective, the integrity of the TCB is the basis of security, so that is the focus of our analysis. In general, it is preferable to have a "minimal" TCB. The smaller the TCB, the easier it is to verify the components. However, if the minimal TCB subjects are dependent on other subjects, then these other subjects must be added to the TCB or dependencies must be removed. In this paper, we will identify dependencies and determine how to resolve them to keep our TCB as small as is feasible.

Since we are striving for a minimal TCB, we do not assume a two-level integrity system (system and user) as in LOMAC [9], but rather we start with the most fundamental system services and try to determine how the integrity of these can be enforced. Applications may further depend on other subjects. For example, Apache depends on other system services and the Apache administrator. We believe these are at two distinct integrity levels [13]. In this paper, we examine only explicitly examine the TCB and non-TCB boundary.

Further, we note that the benefits of least privilege permissions and containment are not relevant to the protection of the TCB. Since the TCB subject types can legitimately transition to any other subject type, containment is not possible for the TCB subjects. Therefore, the focus is on the integrity of these services.

Figure 2 shows the SELinux example policy's type transition hierarchy for our proposed TCB subject types [1]. `kernel_t` is the primordial subject type in the SELinux system. It transitions to `init_t` which then can start a variety of services. Key to our analysis are the administrative (e.g., `sysadm_t`, `load_policy`, `setfiles_t`, etc.) and authentication subject types (e.g., `sshd_t`, `local_login_t`, etc.) that determine the basis for security decisions in SELinux. We also in-

[1] This hierarchy is generated by the *transition* permissions held by each of these subject types.

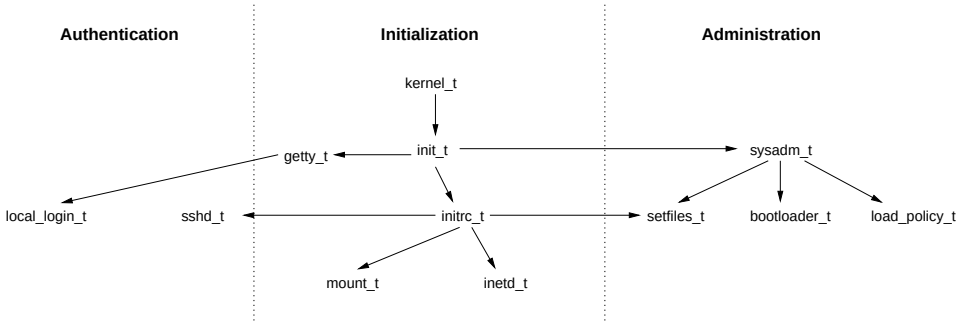

Figure 2: SELinux Example Policy's type transition hierarchy for our proposed TCB subject types.

clude `initrc_t` and `inetd_t` because these services initiate many of the services in a UNIX system.

Of course, there are lots of other services upon which the correct execution of applications is necessary (over 80 identified for the Apache administrator [13]), but we chose this proposal for a minimal TCB based primarily on the early appearance of these services in the type transition hierarchy and the amount of transition "fan-out." Both of these features indicate that vulnerabilities in that subject type will be difficult to contain.

While this TCB represents a small number of subject types, the complexity of their interactions with the rest of the system in the SELinux policy makes manual verification impractical. First, each subject type is included in around 500 to over 1000 policy statements in `policy.conf`. Manual examination of this many statements alone is impractical, but these statements must be compared to the other thousands to determine whether a significant conflict exists. Automated tools are necessary to represent the security goals, identify conflicts, and provide as much support as possible to conflict resolution.

2.4 Integrity Requirements

The goal of our analysis is to protect the integrity of our trusted computing base (TCB), so we need to define more precisely what we mean by this. Traditional policies for integrity protection include Biba [4] and Clark-Wilson [6]. The Biba integrity property is fulfilled if all the higher-integrity processes do not depend on lower-integrity processes in any manner. This implies that a high integrity process cannot read lower-integrity data, execute lower-integrity programs, or otherwise obtain lower-integrity data in any other manner. As a result, a process cannot write data above its integrity level. Therefore, if high and low integrity processes write to the same file, then it must be a low integrity file. Obviously, the high integrity process can no longer read from this file and maintain Biba integrity.

Of course, UNIX applications do not obey a strict Biba integrity. Often higher-integrity processes read input generated by lower integrity processes, but in many cases it is assumed that they can handle this input. When they cannot, we often identify this as a vulnerability in the supposedly high-integrity program [2]. The Clark-Wilson integrity model attempts to capture this notion. In the Clark-Wilson model, *constrained data items* (CDIs) are high-integrity data that are processed only by certified *transformation procedures* (TPs). However, TPs may also process *unconstrained data items* (UDIs). This is similar to high integrity programs processing low integrity data. The Clark-Wilson model also includes *integrity verification procedures* (IVPs) that can be used to verify the integrity of CDIs at particular times.

The Clark-Wilson model is based partly on certification of the components (IVPs and TPs) and partly on their enforcement of particular properties. We do not address certification here, but we examine the plausibility of meeting Clark-Wilson enforcement requirements using SELinux (paraphrased from the Clark-Wilson paper [6]):

- **E1:** Each TP operates on a particular list of CDIs and CDIs are only manipulated by a TP.
- **E2:** The system must maintain a list of subjects, TPs, and the CDIs those TPs may reference, and only those references are permitted.
- **E3:** The system must authenticate the identity of each user that attempts to execute a TP.

[2] Of course, the use of so many root services gives has given a false impression of the integrity of many programs.

- **E4:** The list of TPs and IVPs can only be changed by a subject permitted to certify those TPs and IVPs.

SELinux identifies object types, and thus, the objects manipulated by programs as stated in E1. However, it does not distinguish between CDIs and UDIs. Thus, we know whether CDIs are only processed by TPs, nor do we know which subjects are TPs, as required by E2. The satisfaction of E3 must be provided by the authentication infrastructure in a dependable manner (i.e., using TPs). The mandatory nature of SELinux policies implicitly enforce E4.

Thus, our task is to identify the TPs that define the SELinux example policy's TCB (to satisfy E2). Since we want to ensure the integrity of our TCB, the only CDIs are those processed by the TCB subjects. As a result, we only need to ensure the integrity of these. However, the TCB may operate on UDIs, so we need to distinguish between UDIs and CDIs. Thus, we will perform the following tasks: (1) propose TCB subjects; (2) identify possible low-integrity UDIs (i.e., data whose value may depend on some low-integrity subject); (3) determine whether these are true UDIs; and (4) resolve cases where we suspect that a CDI is processed by a non-TCB subject (i.e., a subject that is not executing a TP).

Since the identification of the use of low-integrity data is essentially Biba, we perform a Biba analysis on our proposed TCB relative to the SELinux example policy. We then perform analyses to classify possible UDIs based on the possible resolutions to the integrity issue.

2.5 Low-Integrity Data

We first distinguish between two types of dependencies on low-integrity data that violates Biba: (1) read integrity violations and (2) read-write integrity violations. The difference is that, in the second case, writes by our TCB may be revised by a lower integrity process. While this is not strictly an issue in Clark-Wilson (i.e., these data may be UDIs), we are not comfortable with the possibility that a TCB subject write UDI data. Thus, we always classify read-write integrity violations as likely CDIs.

If we believe data are likely to be CDIs, then we have the following options to resolve the conflicts: (1) trusting the low-integrity subject (i.e., add it to the TCB); (2) exclude the low-integrity subject; (3) exclude the conflicted object type; (4) policy override; and (5) change the policy. It is possible to extend the proposed TCB, but since we want to keep the TCB minimal and the addition of more subject types will probably introduce more constraints, this is a low priority option.

We can exclude either the conflicting object type or the low-integrity subject type from the system to resolve an integrity conflict. Since we are analyzing the SELinux example policy to create a security target, it is perfectly reasonable to remove subject that cause significant integrity issues that we do not trust. For example, `insmod_t` installs modules into the kernel, but for a high integrity system we will compile the modules into the kernel. Thus, integrity conflicts caused by this service can be ignored. The exclusion of object types may be less plausible given that the object may be necessary for correct processing, but there are some cases where this makes sense. For example, we can eliminate integrity conflicts if we preclude objects of the type `removable_device_t` which may be acceptable for a secure system.

Lastly, we can change the policy by adding overriding statements (e.g., denying the violating read or write) or by modifying the SELinux example policy itself. We have found that handling integrity violations as exceptions or defining special handling for conflicting assignments with common semantics are both effective in reducing the need to express even more complex and fine-grained policies [12]. Modifying the SELinux example policy is a last resort: it is complex enough.

If we believe data are likely to be UDIs, we may assume that the TP is protected or protects itself by sanitizing its UDI inputs. Certification may depend on receiving only specific inputs, in practice, so providing external sanitization may also be an option. We may also identify the need for other security processing, such as auditing and intrusion detection, upon use of UDIs. We see this simply as another alternative to denials.

3 Analysis Approach

The basic approach to evaluating the proposed TCB for the SELinux example policy is as follows. First, we identify Biba integrity violations between the TCB subject types and the rest of the SELinux example policy. Second, we try to classify our conflicts based on the concepts such as the type of integrity violation (i.e., read or read-write), the proposed integrity of the conflicting subject type (i.e., high or low), and the likelihood of exclusion (i.e., of object type or subject type). Third, we

perform some manual analysis to determine the likely solution and see if these results correlate with the classifications. This includes outlining implementations to support these classifications, particularly where sanitization or policy modification is the choice.

3.1 Gokyo Policy Analysis Tool

We have built a policy analysis tool called *Gokyo* that is designed to identify and enable resolution of conflicting policy specifications [12, 13]. The general concept that Gokyo enforces is *safety*. A policy specification is said to be *safe* if no subject can obtain an unauthorized permission [10]. If we take policy administration into account, then any permission can be assigned to any subject type by the administrator in a policy such as TE. Therefore, we need an additional specification concept, typically called *constraints*, to express the safety requirements on the administrators to ensure that our policy meets our goals.

Gokyo implements an approach called *access control spaces* whereby semantically meaningful permission sets are identified and handling can be associated with each set independently. While there are a variety of semantically meaningful permission sets, the most common to consider are: (1) those assigned to a subject type; (2) those precluded from a subject type by a constraint; and (3) the permissions whose assignment or preclusion status is unknown. The overlapping regions of these sets also form subspaces, so we can consider the set of permissions that are both assigned and precluded. Of course, these sets can be further decomposed to provide more effective control. For example, we may deny all integrity conflicts, except log writes, which we allow, and input from network objects, which we sanitize and audit.

The general idea is that by identifying semantics to the subspaces it may be possible to attach handling semantics with the entire subspace. This would permit administrators to keep the basic, simpler constraints that largely work and specify only the additional handling semantics. We have found that the Apache administrator policy for SELinux 2.4.16 largely adheres to a Biba integrity model, such that 19 conflicting cases can be handled as eight access control spaces [13].

Gokyo represents policies using a graphical access control model shown by example in Figure 3. Permissions (i.e., object types and the permitted operations), subject types, and subjects are represented by graph nodes. In addition to this information, permission nodes also store the *object class* (i.e., datatype) and *operations* permitted by the permission. Note that object types are represented by permissions with no rights. In general, a node represents a set, so it is possible to build set-hierarchies consisting of aggregations of individual permissions, subject types, and subjects.

Example 1 Figure 3 shows an example of an access control specification using this model. Subject $s1$ has values $S(s1) = s1$, $R(s1) = r2$, and $P(s1) = P(r2)$. That is, $s1$ represents one subject, $s1$, and is assigned to one subject type, $r2$. Since the only route from propagation of permissions is through $r2$, $s1$'s permissions are defined by $P(r2)$. The value of $P(r2) = P(p6)$ and, since $p6$ is an aggregate its permissions are $P(p6) = P(p4) \cup P(p5)$. Since $p5$ is an aggregate as well, its permissions can be further decomposed.

For expressing constraints in this model, we also use a set-based approach [11]. In general, constraints are expressed in terms of two sets and a comparator function, $set_1 \bowtie set_2$, where $\bowtie$ represents some comparator function. Such comparators are set operations, such as disjointness (i.e., null intersection), cardinality of intersection, subset relations, etc.

Example 2 We define a constraint type for *integrity*. An *integrity* constraint $x \parallel y$ where $x \in R \cup S$ and $y \in R \cup S$ means that the set of read and execute permissions of x must not refer to any objects to which y has write permissions.

For each subject type, Gokyo stores the assigned permissions and the prohibited permissions. The prohibited permissions are the permissions whose assignment to the subject would result in the violation of a constraint, so these permissions are represented in terms of the constraint [3]. Further, Gokyo identifies the access control space consisting of the intersection between the assigned and prohibited spaces. It is this space where conflict resolution is necessary.

3.2 Identifying Integrity Conflicts

Returning to the problem of analyzing our proposed TCB, the SELinux example policy represents the assigned permissions. We add Biba integrity constraints

[3] Details are beyond the scope of this paper. See the detailed Gokyo writeup [13].

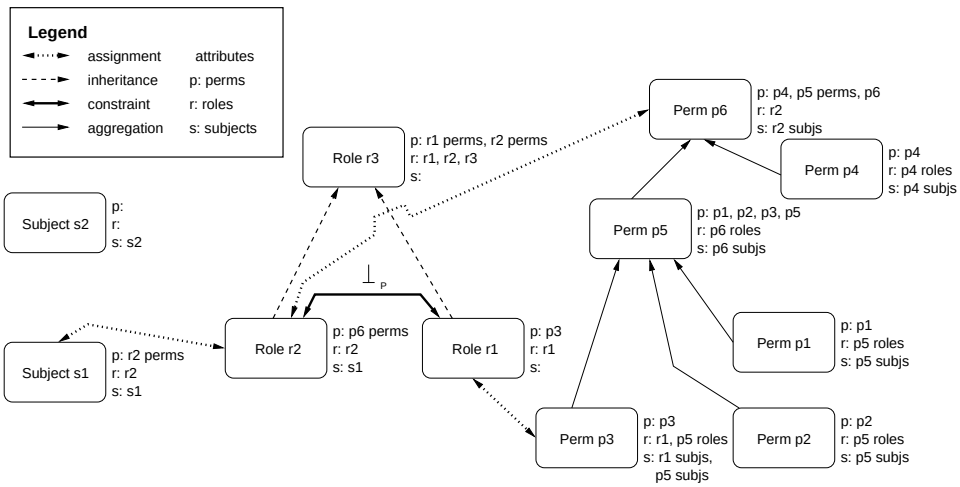

Figure 3: Example access control representation (the fields "p:", "r:", "s:" refer to the permissions, subject types, and subjects assigned to these entities, respectively)

between each of our TCB subject types and all other subject types in Gokyo. That is, for each TCB subject type, we add a constraint that it cannot apply a read/execute operation (i.e., an operation involving input of data to the subject type's processes) to any object type and class combination that is written by any other subject type in the system. Note that this constraint is more restrictive even than our original proposal, but understanding which subject types actually have integrity relationships may be useful in resolving conflicts.

Gokyo implements this constraint by computing read/execute permissions for each object type and class combination written by the other subject types. This set of read permissions is the set of permissions that the TCB subject type may not read. When the constraint is tested, if the TCB subject type has a read/execute permission that intersects one of the precluded permissions then a constraint violation is generated. In some cases, a single allow statement may result in several constraint violations. This can occur when an allow is made on a type attribute rather than a type directly. For example, access to read a variety of network input is the result of a single allow statement. Such assignments are propagated to each object type that has this type attribute. Gokyo shows only the unique assignments that result in violations, but can print all the individual violations to a file. In our results, we will also focus on the unique assignment mainly.

3.3 Classifying Conflicts

Once the conflicting subspace for a TCB subject type is generated, we could choose a handler for this subspace. In general, Gokyo permits overriding the conflict by granting or denying the subspace. However, both grants and denials can be augmented by arbitrary analysis code ranging from audit to complex intrusion detection. Thus, if the integrity conflicts that we find are all representative of the same situation we could choose a single approach to handling them.

In Section 2.4 lists five approaches for dealing with integrity conflicts that are summarized in Table 1. The problem is to determine which integrity conflicts imply which resolutions; the SELinux example policy does not provide any further input explicitly. To address this we classify conflicts in a manner that does not unequivocally identify the resolution, but does identify the possible resolutions. First, members of the TCB subject types may be trusted writers, so if the subject type of an integrity conflict is a TCB subject type then all handling options are possible. Further, some subject types may be candidates to be added to the TCB. Subject types with a significant number of conflicts should be considered. We use the heuristic that subject types with an average of greater than one conflict per TCB type are candidates for trusted types.

Second, we propose that the analysis also include a security target definition that specifies the *required subject types*. Rather than requiring that the system administrators enumerate all subject types individually, we can use the type transition hierarchy to estimate the set of

Class	*Description*
TCB or Candidate	Trusted subject types
Exclude Type	Type can be excluded from secure system with this TCB
Sanitize	A sanitized read may be used to protect TCB
Denial	Denial of conflicting rights can be used to protect TCB
Modify Policy	Policy must be edited to protect TCB

Table 1: Classifications for TCB integrity conflicts.

types required as all subject types that may transition to a required subject type. Type attributes or other semantically meaningful identifiers can be used to identify desired subject type sets. If a subject type does not belong to the set of required subject types it can be considered for exclusion and the other remaining handling methods.

Identifying object types that may be excluded is more difficult. If we are too ambitious, we may remove an object type that is really needed by the system. In general, when we exclude a subject type, we may remove object types depend on the existence of this subject type. For example, if we remove X windows subject types, we no longer need X windows object types. This may prevent integrity violations for TCB subject types with broad rights, such as the system administrators. The dependency of a subject type on the availability of particular object types is not currently identified. All we know are the operations that can be performed. A conservative approximation is the object types for which objects can only be created by the excluded subject type. Without the subject type, objects of this type would not exist in the system. We have to account for all possible ways of making objects of this object type, including relabeling (specifically *relabelto* permission).

Third, some Biba integrity violations involve reading low integrity that the subject type can actually handle, such as requests for operations. The Clark-Wilson integrity policy accounts for these by allowing transformation processes (TPs) to operate low integrity data (unconstrained data items or UDIs) and even convert them to high integrity data (constrained data items or CDIs). We refer to the ability to correctly function given UDI input as *sanitization* of this input. In Clark-Wilson, TPs must be certified to perform their tasks. We identify both where TPs require sanitization and where they must handle CDIs properly. Our initial assumption is that all data used by TCB subject types are CDIs, but some data may be downgraded to UDIs and used via sanitization.

Recall the distinction between read integrity and read-write integrity violations. We state that read integrity violations may be sanitized, but read-write integrity conflicts have no possibility of sanitization (i.e., data written by a TCB subject type is always a CDI, in Clark-Wilson terms). Recall that read-write integrity violations mean that the subject type writes and reads data that can be modified by a lower integrity subject type. Depending on synchronization, a lower integrity subject type may be able to change an objects as the higher integrity subject type is writing them. While sanitization may be possible in general, we flag these violations as being beyond sanitization.

Fourth, the read-write integrity violations are classified for ad hoc denial of rights. In many cases, more rights are assigned than are really necessary for the application, which is a problem of least privilege. In some cases, it may be sufficient and simpler to simply deny the conflicting rights. Gokyo enables partitioning of conflicts, and assigning independent handling to each partition. Therefore, it is possible to simply denial these rights without further modifying the SELinux example policy. Application-specific examination is necessary to determine if these denials are really possible.

Lastly, if we find that the permission assignment is necessary for the convenient execution of a required application, then modification of the policy is the only remaining option.

3.4 Manual Analysis

Manual analysis involves starting at the highest level handling method and determining whether it can actually be applied. If not, then the subsequent methods must be considered.

Identifying trusted writers and excluded writers can be done automatically, so the main effort here is on determining whether sanitization is possible and identifying the sanitization method. This is a fairly ad hoc process, so we examine it relative to our integrity analysis results in Section 4.

If sanitization is not possible, then expressing a denial for this exception or policy modifications are the remaining options. Both of these must be done manually at present.

4 Integrity Analysis

In this section, we use Gokyo to analyze our proposed TCB to identify the integrity conflicts, classify according to best possible resolution, and choose the likely resolution. The likely resolution is chosen based on manual analysis of the conflict. The key results are the resultant TCB (i.e., does it need to be expanded and how?) and proposed SELinux policy changes needed to achieve this TCB. Detailed discussion of the Gokyo tool itself is provided elsewhere [13].

4.1 Analysis Implementation

The integrity analysis for the proposed TCB in Section 2.3 is performed on the SELinux example policy for Linux 2.4.19. This policy consists of over 50,000 policy statements [4]. In Gokyo, the SELinux example policy comprises over 700 subject types and type attributes, over 40,000 individual permissions, and over 100,000 explicit assignments between subject types and permissions.

The integrity of the SELinux system is represented by two integrity constraints between the set of proposed TCB subject types and the set of all other subject types as shown in Figure 4. To represent this we create two subject types, TCB subject types (high integrity) and non-TCB subject types (low integrity), and aggregate the subject types into their respective group. The permissions assigned to each subject type node are automatically propagated to the aggregate subject types.

Our integrity protection goal is expressed using two constraints: (1) read-integrity constraint and (2) read-write-integrity constraint. Read-integrity constraints are violated if the low integrity subject has write permission (i.e., a permission representing the ability to modify the data in that SELinux class) to an object type and class pair that high integrity subject type has read permission to. Read-write integrity is violated if the high-integrity subject also has write permission to the object type and class pair in addition to read permission.

[4] Statement count is taken from the macroexpanded policy in `policy.conf`.

To implement these constraints, Gokyo assigns the invalid permissions to the high integrity set. For read-integrity, Gokyo creates a permission with all read permissions assigned for each object type and class pair that the low integrity subject can write. Similarly, for read-write-integrity, Gokyo creates a permission with all write permissions assigned for each object type and class pair that the low integrity subject can write. In this case, constraint verification takes an intersection of the invalid permissions and the ones assigned to the TCB subject types (i.e., different types of constraints may have different algorithms).

Note that it may be more efficient to test this constraint in the opposite manner by determining if the low integrity set has write permissions to object type and class pairs that the high integrity subject can read. At this point, we actually create both integrity test sets, but we should determine which is smaller and test only that one instead.

Analyzing integrity protection is basically a task of identifying all integrity conflicts and classifying them into their best legal classification. We have found that the number of conflicts that exist in the entire SELinux policy is too large to be effectively considered together. Fortunately, conflicts are independent. That is, the existence of one conflict has no effect on another. This means that a classification to eliminate one conflict cannot be undone by another conflict. For example, if we find that we can sanitize the use of a particular conflicting permission, the emergence of a conflict later does not impact this sanitization. This is true for all classifications. The one caveat is that we may find that a particular subject requires so many sanitizations that it should be trusted or excluded, but these cases are not excessive and easily handled. Usually, we determine whether a subject should be trusted or excluded before we do the hard work of figuring out a sanitization.

The result is that we can consider the conflicts in small groups, and make classifications based on a subset of the information. Currently, we use Gokyo in a mode in which it identifies a single conflict for each invalid permission (i.e., constraint-generated). Sometimes, attribute assignments result in multiple, unique conflicts, but Gokyo only presents the attribute assignment once. Gokyo generates a log containing all conflicts and the assignment paths between nodes involved in the conflict, including the line numbers in which the assignments are specified. This assists with the manual analysis phase. However, addition metadata, such as the frequency of conflict for a particular invalid permission, would also be useful. The log of a constraint violation is shown below (`class.conf` is SELinux policy file, `kernel.cst`

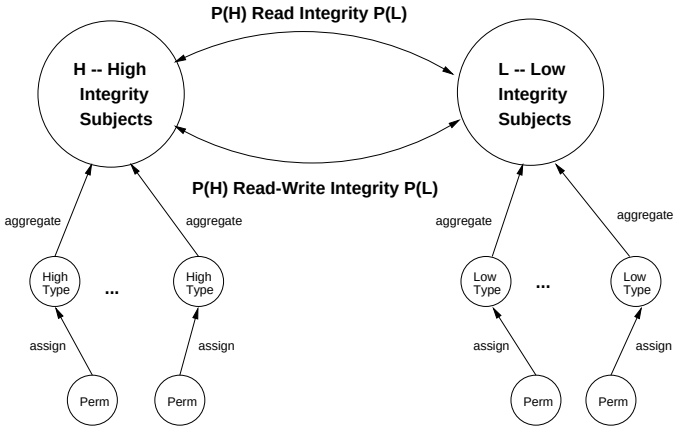

Figure 4: Gokyo graphical policy model implementation of integrity.

Trusted Type	*Conflict Type*	*Object Type & Op*	*Class*	*Resolution*
dpkg_t	tmpreaper_t	tmp_dpkg_t:file rw	exclude	exclude
initrc_t	many	file_type:blk/chr/file r	sanitize	sanitize
initrc_t	useradd_t	etc_t:file r	trust	trust
initrc_t	hwclock_t	clock_device_t:chr/blk rw	trust	trust
initrc_t	gpm_t	psaux_t:chr rw	exclude	exclude
initrc_t	sound_t, xdm_t	sound_device_t:chr rw	trust	exclude
initrc_t	httpd_admin_xserver_t	framebuf_device_t:chr rw	deny	exclude
initrc_t	many	initrc_t:fifo rw	deny	sanitize
kernel_t	slapd_t, squid_t, +	*:*_socket r	sanitize	sanitize
kernel_t	dhcpc_t	resolv_conf_t:file r	trust	exclude
kernel_t	dhcpd_t	var_run_dhcpd_t:file r	trust	exclude
kernel_t	quota_t	file_t:file r	trust	trust
local_login_t	many	proc_t:file r	sanitize	sanitize
local_login_t	insmod_t	local_login_t:process r	exclude	exclude
local_login_t	logrotate_t	local_login_t:process r	trust	trust
mount_t	automount_t	autofs_t:dir rw	exclude	trust
mount_t	bootloader_t, fsadm_t	fixed_disk_device_t:* rw	trust	trust
sysadm_t	user_t	misc_device_t:* rw	deny	exclude obj
sysadm_t	many	sysadm_devpts_t/ptyfile:* rw	deny	change
sysadm_t	sysadm_*_t	sysadm_home_t:* rw	deny	change/sanitize one file
sysadm_t	sysadm_*_t	sysadm_tmp_t:file rw	exclude	change
sysadm_t	sysadm_irc_t	sysadm_irc_t:file rw	exclude	change/sanitize
sysadm_t	sysadm_xserver_t	sysadm_xserver_t:shm rw	exclude	exclude
sysadm_t	sysadm_xauth_t	sysadm_home_xauth_t:file rw	exclude	exclude
sysadm_t	admin	kernel_t:system avc_toggle rw	trust	trust
sshd_t	many	sshd_devpts_t/userpty:* rw	deny	change

Table 2: Integrity conflicts in the initial TCB proposal.

is our constraint file, and `kernel.cfg` contains aggregate subject type definitions):

```
On constraint: kernel.cst(25)
Role 151: mount_t
has constraint:  "integrity protected"
with node: Role 882: non-mount

Violating Assignments:
  Permission 2876: autofs_t:dir 00110000
  (1) From: class.conf (60810) Perm 2876:
autofs_t:dir 00110000
  (2) to: class.conf (60759) Role 151: mount_t

Violating Preclusions:
  Permission 45131: autofs_t:dir 003fffff
  (3) From: kernel.cst (25) Role 882: non-mount
  (4) to: class.conf (60759) Role 151: mount_t
  (5) to: class.conf (0) Perm 42608:
autofs_t:dir 003e1c7e
  (6) to: class.conf (60639) Perm 2857:
autofs_t:dir 003e1c7f
  (7) to: kernel.cfg (94) Role 148: automount_t
  (8) to: kernel.cfg (91) Role 882: non-mount
```

The *violating assignments* is the permission assigned to `mount_t` whose integrity may be violated. Line (1) indicates where the permission was assigned to `mount_t`, and line (2) indicates where `mount_t` was identified as a subject. The file `class.conf` is a truncated version of `policy.conf` for SELinux 2.4.19. For the *violating preclusions*, the path for the assignment of the constraint-generated invalid permission is shown. Line (3) refers to declaration of the aggregate subject type (Gokyo-specific), and line (4) is the same as line (2). Line (5) refers to the generated permission (no file line number because it is generated), and line (6) shows the assignment of `autofs_t` permissions to `automount_t`. Lines (7) and (8) show that `automount_t` is assigned to the non-TCB aggregate.

For each conflict, Gokyo estimates the classifications based on: (1) the number of subject type conflicts (for trust); (2) whether the subject type or object type is required, see below (for excluding subject types); and (3) whether the conflict is read-integrity or read-write-integrity (for considering sanitization). Our proposal for removing object types based on whether the object type is created by only excluded subject types has not been implemented yet, so we use the object types required by our focal subject type.

For required subject types, we assumed that the purpose of our system was to run an Apache web server. Thus, we include all Apache subject types (i.e., those starting with `httpd`) and all those subject types that transition to an Apache subject type, directly or indirectly. In addition to our TCB subject types, we require `dpkg_t` (i.e., the Debian package manager), `rlogind`, several user subject types. Ultimately, we will choose to exclude `rlogind`, but include `user_t` in the analysis. Users may be actively involved in script generation (e.g., for personal pages in a corporate server). Because so few other required subject types are found this way, we will add others later. Note that the set of required object types includes the types accessible to the Apache subject types only.

4.2 Analysis Process

Table 2 shows the integrity conflicts that our proposed TCB has with the remaining system subject types and the possible resolutions of these conflicts. The *trusted type* field shows a trusted type that reads input written by an untrusted type. The *conflict type* field shows one or more of the untrusted types in the conflict. The *object type & op* field shows the conflicting data and the rights of the TCB subject type (i.e., read or read-write integrity conflict). The *class* field shows the classification of the conflict. The *resolution* field shows the manual resolution to the conflict.

The integrity conflicts are collected into groups based on the trusted type. First, `dpkg_t` (debian package management) has a common read-write integrity conflict also because `tmpreaper` (cleans temporary file directories) is given broad file access for cleaning up temporary files. `tmpreaper_t` is responsible for few violations, so the classification is *exclude*. This specification is consistent with `tmpreaper`'s task, so the only two alternatives are to trust or exclude `tmpreaper_t`. We manually choose the latter.

Second, `initrc_t` is involved in a read integrity conflict that affects most trusted types: it is given read access to all file data in the system. Since it can read all files, it certainly has an integrity conflict with the lower integrity subjects. However, the read access is to `getattr` for `stat`, so this can be sanitized.

Third, `initrc_t` has several other conflicts. The next two are identified as required and seem necessary, so we add `useradd_t` and `hwclock_t` are added to the TCB. The next three are not really necessary (`gpm_t` for mouse, `sound_t`, and `xdm_t`), so we choose to exclude them. The X window server introduces a number of other integrity issues, so much more work is necessary to have an X windows system running on an integrity-protected TCB. Thus, `httpd_admin_xserver_t` is excluded. Lastly, we determine that read-write integrity access to `initrc_t`'s fifo can be sanitized as necessary. It should involve only simple communication (e.g., on process start). Note that this is a manual override of our intended requirements.

Fourth, kernel_t has several integrity conflicts with receiving network data. This integrity conflict is common to most services in the TCB. Such interaction is necessary for convenient execution, so we will examine sanitization of network communication in Section 4.3. The other conflicts are so common that the framework assumes that they are trusted. Manual analysis keeps on quota_t (file quota management) in the TCB and excludes dhcpc_t and dhcpd_t.

Fifth, the conflict over access to /proc is found for local_login_t. Since this access is for reading only, we will aim to sanitize this access. Next, we assume that installing modules is not necessary for our secure system, so insmod may be excluded. On the other hand, logging is an important process, so logrotate is added to the TCB.

Sixth, mount_t has conflicts with automount_t, fsadm_t, and bootloader_t. Although only the latter two are common conflicts, all of these are added to the TCB.

Seventh, there are a variety of conflicts with sysadm_t. sysadm_t has a conflict over access to misc_device_t with user subjects. These object types will be excluded. Also, access to sysadm_devpts_t is shared with many subject types. Many of these subjects are application-specific administrators which are intended to be of lower integrity. A different object type should be designated for these. Next, sysadm_t has read-write integrity conflicts with the application-specific administrators over the sysadm_home_t. Conflicting access is provided to permit lower-integrity administrative processes to write to an error log (.xsession-errors). We recommend breaking the object type into two for the higher and lower integrity home objects, so access to the latter can be sanitized. Since we have excluded X windows this object type can also be excluded in this case. Access to sysadm_tmp_t and sysadm_irc_t should be changed similarly. Finally, sysadm_t has conflicts that can be excluded for X windows subject types and trusted for administrative subject types. The following subject types are added to the TCB: ipsec_mgmt_t, apt_t, and admin_passwd_exec_t. install_menu_t is excluded.

Lastly, sshd_t has a read-write integrity conflict over the use of pseudo-terminals. Type change is used for some to change the subject type to a lower-integrity subject upon use of a user pty for sysadm_t, so we presume that this should be added for sshd_t as well.

After the trusted types, excluded types (including object types), and sanitized accesses are added to their respective lists, the next iteration of the analysis can be performed. After some number of iterations, 5 in our case, all the exclusions, sanitizations, and trusted subject types are accounted for, and no conflict remains unclassified. However, resolving the efficacy of sanitizations and reduce file read permissions (or at least managing them) remain.

4.3 Analysis Findings

The base TCB for the SELinux example policy for supporting an Apache is shown in Table 3. Note that the set of subject types that Apache must ultimately depend on would be somewhat larger (around 50% larger given our analysis [13]). Starting with our original 12 types, we have found that 30 subject types must be trusted. The correctness of this TCB depends on the resolution of the full access that these subject have to application and user files which they should probably rarely access, as discussed below. Also, not all forms of authentication are necessary at once. Ultimately, it is probably possible to reduce this set slightly, but this provides a good estimate of most SELinux TCBs.

Interestingly, not long after this paper was submitted, Wayne Salamon independently proposed a "core policy" to the SELinux community [19]. The intent of this proposal was to define a base system policy upon which any other system policies would be derived. There is some notion of usability here rather than TCB, as the intent is for base function rather than base security. After some discussion with the community he settled on 40 policy files (roughly equivalent to 40 subject types) to comprise a core policy. 17 of the subject types in the TCB are common to the two groups. The ones that we include that are not in the core policy proposal are indicated in Table 3. We think that many of the subject types in our proposal are TCB subject types, although some authentication subject types, such as sshd_t, and administrative types, such as sysadm_t and dpkg, are not necessarily core.

As part of the analysis, we identified subject types and object types for exclusion from our system. The 25 subject types we excluded are listed in Table 4. We need to verify empirically that such services are not actually necessary for an Apache system on SELinux, but most of these do not seem controversial.

In Table 5, we summarize the sanitizations required for our TCB. Note that in Clark-Wilson terms, these saniti-

kernel_t*	init_t	initrc_t	sysadm_t*	getty_t
mount_t	fsadm_t	load_policy_t	dpkg_t*	checkpolicy_t
setfiles_t	syslogd_t	klogd_t	automount_t	sshd_t*
sshd_login_t*	local_login_t	quota_t*	ldconfig_t	useradd_t
hwclock_t*	apt_t*	cardmgr_t*	ipsec_mgmt_t*	admin_passwd_exec_t*
bootloader_t	logrotate_t	newrole_t	snmpd_t*	passwd_t*

Table 3: Final trusted computing base subject types (* indicates not in proposed SELinux core policy).

insmod_t	rlogind_t	remote_login_t	sysadm_xserver_t	xdm_t
sysadm_xauth_t	sound_t	tmpreaper_t	httpd_admin_xserver_t	kmod_t
lpd_t	xdm_xserver_t	vmware_user_t	sendmail_t	procmail_t
hotplug_t	traceroute_t	update_modules_t	gatekeeper_t	smbd_t
dhcpc_t	dhcpd_t	install_menu_t	devfsd_t	gpm_t

Table 4: Final excluded subject types.

zations indicate the unconstrained data items (UDIs) that our TCB's transformation procedures (TPs) must handle. By sanitization, we envision that SELinux modules can be annotated with sanitization policies to verify the format of the inputs. This is a non-trivial endeavor, so such a proposal is quite preliminary. However, such sanitization services on top of a verified and simple integrity policy can enable fulfilling of our security goals without major policy tweaking.

Some of these sanitizations are focused and can be handled as exceptions, but some (the first four) have many instances. Our impression is that the fifos can be handled because each instance serves the same purpose. Socket access is both extensive in number of communicators and variety of communications. Significant effort is required to comprehensively address these. Most of the information in `/var` and `/proc` does not seem to impact the processing of our trusted subjects, but more investigation is necessary.

The two conflicts that remain are: (1) between trusted subject types and the pseudo-terminals that they share with user process and (2) the permission assignments that permit trusted subjects to access to all files (the first and last entries in Table 2). The first conflict is probably best handled by a SELinux *type change* statement. These are used to change the type of an object based on the subject type of the accessor. When a pseudo-terminal is accessed by a high integrity subject, it gets a high integrity type and its previous state is cleared.

The second conflict could be addressed by leveraging Gokyo. Using Gokyo's conflict spaces, we could declare auditing or intrusion detection to be initiated when an integrity-conflicted file object is accessed by a high-integrity subject. This would not require a modification to the SELinux policy, but a Gokyo conflict space assignment would be necessary [13]. Such a solution depends on infrequent use of conflicting permissions. If there is frequent use of some conflicting permissions, then alternative measures are needed. This task remains as future work.

Note that a SELinux *auditallow* statement would not quite work in this case because it would audit all file accesses instead of the ones that violate integrity. Of course, we could always modify the SELinux policy, but this would take significant effort and perhaps lead to further conflicts.

5 Related Work

SELinux includes tools for policy analysis. *neverallow* statements enable the policy designer to express assignments that should not be expressed in the policy. The *checkpolicy* tool verifies that no *neverallow* statement is violated when the policy is compiled. Such statement enable the identification of conflicts, but any resolution requires changing the SELinux policy directly. These statements are suitable for expressing cases that should not ever occur, but they are not suitable for expressing high level security goals.

The Tresys Corporation has been developing tools for analyzing SELinux policies for over one year [23]. Tresys defines tools for helping administrators understand the SELinux policy (e.g., statements for a par-

Object Type	*Sanitization*
*_t:fifo	Mainly for use following `exec`
*:*_socket	Must be able to handle network data or big policy mod
proc_t:file	Mainly expected to print this information
sysadm_home_t:*	Only need to read .xsession-errors log

Table 5: Sanitized conflicts and brief analysis.

ticular subject type) and helping perform administrative tasks (e.g., correctly adding a new user). Such tools are valuable for the development of SELinux policies, but do not address the questions of whether the policies can meet particular high-level goals.

We are aware of work ongoing at MITRE to analyze SELinux policies for more complex relationships, such as reachability [7]. The SELinux example policy is so large that the theorem proving tools being used are not efficient enough for effective analysis yet.

Access control policy analysis itself is a fairly recent area of work. Bertino et al define a general framework for representing and reasoning about access control models [3]. The goal here is to compare models (e.g., for expressive power) rather than compare policies to properties. We believe that their model is expressive enough to do the latter, however.

Further, Jajodia et. al. [14] support conflict resolution in their model. In their case, the goal is to find a general strategy of conflict resolution, not to support different strategies. Ferrari et. al. [8] examine conflict resolution problems and strategies as well.

6 Conclusions

In this paper, we present an approach for analyzing integrity protection of the SELinux example policy. The SELinux module supports the recent Linux Security Modules (LSM) framework for implementing mandatory access control on the Linux kernel. The SELinux example policy is undergoing active development and is being applied in several installations. The aim is for administrators to take the SELinux example policy and customize it to their site's security goals. This quite difficult, however, because the SELinux policy model is quite complex and the SELinux example policy is large.

Our aim is to provide an access control model to express site security goals and resolve them against the SELinux policy. In particular, we want to identify a minimal system TCB for the SELinux example policy that satisfies Clark-Wilson integrity restrictions relative to the rest of the system. UNIX systems are not designed to meet Biba integrity, but the Clark-Wilson integrity policy enables a description where key data can be identified (those data used by TCB subject types), and sanitization of low integrity data is possible.

We have developed a tool called Gokyo that represents the SELinux example policy and our integrity goals, identifies conflicts between them, estimates the resolutions to these conflicts, and provides information for deciding upon a resolution. Further, Gokyo represents the state of the integrity resolution which could be used by the access control module to make authorization, audit, and intrusion detection decisions. Using Gokyo, we found a minimal TCB containing 30 subject types that meets Clark-Wilson integrity including sanitization requirements and resolution of overly broad file access rights. More investigation is needed to verify the proposed sanitization requirements and determine the effectiveness of audit versus restriction of file rights, but the Gokyo's ability to support the analysis of integrity protection is helpful in understanding and managing higher level security goals on complex policies.

Acknowledgements

The authors would like to thank the anonymous referees for their useful comments, and those people participating in the SELinux community, particularly Stephen Smalley and Russell Coker.

References

[1] L. Badger, D. F. Sterne, D. L. Sherman, K. M. Walker, and S. A. Haghighat. A Domain and Type Enforcement UNIX Prototype. In *Proceedings of*

the 1995 USENIX Security Symposium, 1995. Also available from TIS online archives.

[2] D. Bell and L. La Padula. Secure Computer Systems: Mathematical Foundations (Volume 1). Technical Report ESD-TR-73-278, Mitre Corporation, 1973.

[3] E. Bertino, B. Catania, E. Ferrari, and P. Perlasca. A logical framework for reasoning about access control models. *ACM Transactions on Information and System Security (TISSEC)*, 5(4), Nov 2002.

[4] K. J. Biba. Integrity considerations for secure computer systems. Technical Report MTR-3153, Mitre Corporation, Mitre Corp, Bedford MA, June 1975.

[5] W. E. Boebert and R. Y. Kain. A Practical Alternative to Hierarchical Integrity Policies. In *Proceedings of the 8^{th} National Computer Security Conference*, Gaithersburg, Maryland, 1985.

[6] D. D. Clark and D. R. Wilson. A comparison of commercial and military computer security policies. Proceedings of the 1987 IEEE Symposium on Security and Privacy, 1987.

[7] A. Herzog. Personal communication.. November 2002.

[8] E. Ferrari and B. Thuraisingham. Secure database systems. In O. Diaz and M. Piattini, editors, *Advanced Databases: Technology and Design*, 2000.

[9] T. Fraser. LOMAC: Low Water-Mark Integrity Protection for COTS Environments. In *Proceedings of the 2000 IEEE Symposium on Security and Privacy*, May 2000.

[10] M. A. Harrison, W. L. Ruzzo, and J. D. Ullman. Protection in operating systems. *Communications of the ACM*, 19(8), August 1976.

[11] T. Jaeger and J. E. Tidswell. Practical safety in flexible access control models. *ACM Transactions on Information and System Security (TISSEC)*, 4(2), May 2001.

[12] T. Jaeger, A. Edwards, and X. Zhang. Managing access control policies using access control spaces. In *Proceedings of the 7^{th} ACM Symposium on Access Control Models and Technologies*, June 2002.

[13] T. Jaeger, A. Edwards, and X. Zhang. Policy management using access control spaces. *ACM Transactions on Information and System Security (TISSEC)*, to appear.

[14] S. Jajodia, P. Samarati and V. Subrahmanian. A Logical Language for Expressing Authorizations. *Proceedings of the IEEE Symposium on Security and Privacy*, 1997.

[15] P. Karger and R. Schell. Thirty years later: Lessons from the Multics security evaluation. IBM Technical Report, RC 22534, Revision 2, September 2002.

[16] P. Loscocco, S. Smalley, P. Muckelbauer, R. Taylor, J. Turner, and J. Farrell. The inevitability of failure: The flawed assumption of computer security in modern computing environments. *Proceedings of the 21^{st} National Information Systems Security Conference*, October 1998.

[17] S. Minear. Providing policy control over objects in a Mach-based system. *Proceedings of the Fifth USENIX Security Symposium*, 1995.

[18] National Security Agency. Security-Enhanced Linux (SELinux). http://www.nsa.gov/selinux, 2001.

[19] W. Salamon. Core policy, second pass. SELinux mailing list archives, http://www.nsa.gov/selinux/list-archive/3941.html, 2003.

[20] S. Smalley. Configuring the SELinux policy. NAI Labs Report #02-007, available at www.nsa.gov/selinux, June 2002.

[21] R. Spencer, S. Smalley, P. Loscocco, M. Hibler, and J. Lapreau. The Flask security architecture: System support for diverse policies. *Proceedings of the Eighth USENIX Security Symposium*, August 1999.

[22] C. Wright, C. Cowan, S. Smalley, J. Morris, and G. Kroah-Hartman. Linux Security Modules: General security support for the Linux kernel. *Proceedings of the Eleventh USENIX Security Symposium*, August 2002.

[23] Tresys Technology. Security-Enhanced Linux research. www.tresys.com/selinux.html, 2001.