NETRA: Seeing Through Access Control

Prasad Naldurg
Microsoft Research India
Bangalore 560080, India
prasadn@microsoft.com

Stefan Schwoon*
Universität Stuttgart
70569 Stuttgart, Germany
schwoosn@fmi.uni-stuttgart.de

Sriram Rajamani
Microsoft Research India
Bangalore 560080, India
sriram@microsoft.com

John Lambert
Microsoft Corporation
Redmond, WA 98052, USA
johnla@microsoft.com

ABSTRACT

We present NETRA, a tool for systematically analyzing and detecting explicit information-flow vulnerabilities in access-control configurations. Our tool takes a snapshot of the access-control metadata, and performs static analysis on this snapshot. We devise an augmented relational calculus that naturally models both access control mechanisms and information-flow policies uniformly. This calculus is interpreted as a logic program, with a fixpoint semantics similar to Datalog, and produces all access tuples in a given configuration that violate properties of interest. Our analysis framework is programmable both at the model level and at the property level, effectively separating mechanism from policy. We demonstrate the effectiveness of this modularity by analyzing two systems with very different mechanisms for access control—Windows XP and SELinux—with the same specification of information-flow vulnerabilities. NETRA finds vulnerabilities in default configurations of both systems.

Categories and Subject Descriptors

D.4.6 [**Operating Systems**]: Security and Protection—*access controls, information flow controls*; D.2.4 [**Software Engineering**]: Software/Program Verification—*validation, formal methods*

General Terms

Security, Design, Verification

Keywords

Vulnerability Reports, Privilege Escalation, Static Analysis

*Work done while the author was a visiting researcher at Microsoft Research India, on leave from Universität Stuttgart.

FMSE'06, November 3, 2006, Alexandria, Virginia, USA.

1. INTRODUCTION

Modern operating systems (OSs) make access-control decisions using configuration metadata such as access tokens, security descriptors, capability lists, and access-control lists (ACLs). The metadata are stored in different formats and can be manipulated in a variety of ways, directly influencing what is perceived as access control behavior. We argue that existing interfaces to query and manipulate this information are too low-level, and do not allow application and system developers to specify information-flow goals and verify their intent effectively. This is for a variety of reasons:

- In feature-rich operating systems such as SELinux (Security Enhanced Linux) or Windows XP, there is a complex interplay between different access-control security mechanisms. For example, in Windows XP, access checks based on a principal's access token and a resource's security descriptor can be quite involved. For instance, the token may contain group membership information which is inherited from some parent object, and could have various attributes that users may not be aware of that influence the outcome of the decision. Further complications arise because of impersonation and privileges. SELinux, on the other hand, mixes many different types of access-control mechanisms (discretionary, role-based, and mandatory). It is difficult for even a dedicated system administrator to keep track of all these features, and an ordinary user may not even be aware of them.

- Simple information-flow policies such as confidentiality, integrity and privilege-escalation cannot be specified and enforced directly. Typically, a system administrator or developer wants to ensure certain properties such as *integrity*, where a lower-privileged process/user should not be able to modify data used by a higher-privileged process, or *confidentiality*, where security-sensitive information should not be accessible to lower-privileged processes. However, in order to enforce these conceptually simple properties, a variety of low-level settings need to be configured correctly. Some security-related dependencies are not directly visible. For example, in Windows XP the membership of a user in the group "Interactive Users" is decided implicitly during logon.

- The protection model underlying many OSs is rigid, typically restricted to an all powerful kernel mode and a lesser privileged user mode, and cannot be changed easily. Thus, many applications run with far more privileges than required to execute the task at hand [3].

As a result of these difficulties, system developers may inadvertently create access vulnerabilities, e.g., by configuring overly permissive ACLs and assigning higher privileges. As we show in this paper, new OS versions are released routinely with vulnerable default configurations. Consequently, when a new application is integrated with the rest of the system, regardless of its protection level, it is difficult to analyze the impact of its default permission and privilege settings and choose a configuration that best minimizes the risk of an information-flow vulnerability.

Our main contribution is a novel analysis framework for organizing access-control metadata, and automatically detecting information flow vulnerabilities. Our tool, NETRA, performs static analysis on a dynamically generated snapshot of the access-control metadata. We represent both policies and mechanisms in an augmented relational calculus with Datalog-like rules [15], and NETRA uses a custom-built inference engine to compute least fixpoints and store all derivation trees corresponding to violations of these properties.

Our framework is designed to be flexible and modular—the architecture has two layers that effectively separate the intended security policies from the mechanisms by which these may be implemented. The *mechanism layer* consists of a list of OS-specific access-control derivation rules that take the metadata as inputs, and output simplified sets of inferred permissions available to principals in our system, masking model-specific implementation details. The *policy layer* consists of queries which are interpreted over these derived relations, and correspond to the information-flow properties of interest. When a property violation is detected, the derivation trees with the mechanism-level details can be retrieved on-demand.

The policy layer is not OS-specific, but NETRA can accommodate different underlying access-control models at the mechanism layer, as long as these specifications eventually generate the simplified relations used in the policies. We demonstrate this by instantiating our framework for both SELinux and Windows XP. In these examples, the same policies can be queried independently on our models without any modifications. Similarly, new queries can be integrated into our framework with different underlying models easily. We argue that this decoupling between the behavioral model of the underlying access-control system and the security policies it satisfies is extremely powerful. It provides us with a methodology for closing the gap between mechanisms and policies.

Using this framework, we found security vulnerabilities in both Windows XP and SELinux. In Windows XP we found a number of world-writable resources whose contents were used by admin processes. In SELinux we found processes that were supposedly restricted to the least privileges necessary to fulfill their tasks, but were in fact running with rather larger privileges, thus defeating the purpose of confinement. Our tool also maintains and outputs the proof trees for the vulnerabilities it finds, allowing the user to inspect the report (and possibly fix the underlying problem). The separation of policy from mechanism also enables the tool to present vulnerability reports hierarchically, starting with the policy violation, adding more implementation details at each level, eventually leading to the corresponding violations in the mechanisms. We demonstrate that this leads to readable and understandable vulnerability reports.

The remainder of the paper is organized as follows: In Section 2, we present an architectural view of our framework and highlight its main components, and explain how they provide the abstractions to specify and validate our properties of interest. In Section 3, we describe the mechanism layer of our framework in more detail, and present two examples to illustrate the general nature of our framework by modeling the access-control rules of both Windows XP and SELinux. Section 4 describes the policy layer, where we specify information-flow vulnerabilities concisely in our specification language, specifically in terms of integrity, confidentiality and privilege escalation vulnerabilities, and infer the potential spread of a threat for a given configuration. In Section 5, we present empirical results of running NETRA on default Windows XP and SELinux configurations. In Section 6, we present related work relevant to our methodology. We end with concluding remarks and discussion in Section 7.

2. BACKGROUND AND OVERVIEW

Access control is typically defined as a relational model over the following domains: the set of subjects S (or principals), the set of objects O (or resources) and the set of rights R (or permissions). Access control is a characteristic function on the set $A \subseteq S \times O \times R$. A principal s is granted permission r over resource o iff $\langle s, o, r \rangle \in A$.

In operating systems, the access-control model is typically implemented with a reference monitor using a data structure called the access matrix. For efficiency and other reasons, the access matrix is either stored as an access list, which is associated with a resource and is the list of all principals and their associated permissions on the given resource, or as a capability list, which is the list of resources and associated permissions a given principal is capable of accessing.

Most access-control models impose further restrictions or constraints on the derivation of the characteristic function, or the set of allowed access tuples. For instance, classical access-control models include the concept of *ownership*. The two most popular models, the Discretionary Access Control (DAC) and the Mandatory Access Control (MAC), differ in terms of who is allowed to change permissions associated with a resource, with the DAC model deeming that it is the discretion of the owner, whereas MAC stipulating a system-wide policy to all principals and resources, which cannot be changed by individuals. Most systems today are a combination of MAC and DAC, with resources that can be owned by individual users, but with the system being able to override any user permissions under special circumstances.

Though the mechanisms used to implement the characteristic function can be different for different access-control models, the function is derived from a (usually implicit) higher level information-flow policy specification. These policies that govern the flow of information in a system are independent of the mechanisms used to implement the access function. As mentioned earlier, typical information-flow properties in this context are confidentiality and integrity, expressed in terms of desirable read-write and write-read traces. We can also model some privilege escalation

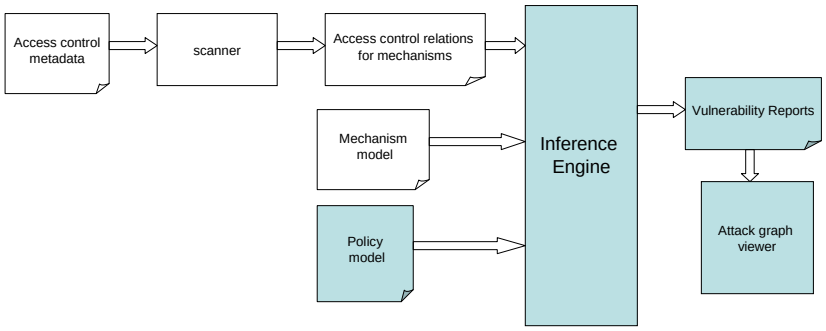

Figure 1: Architecture of NETRA

vulnerabilities as information-flow properties. A privilege-escalation attack that exploits a buffer-overflow vulnerability takes advantage of an underlying coarse-grained protection model that grants access to protected system functions. A more direct vulnerability is when a resource can be written by an admin-user and subsequently executed by a lower-privileged user. An attacker can take advantage of this vulnerability to introduce malicious code that will be executed by the admin-user and as a consequence successfully mount a privilege escalation attack.

Our tool NETRA, analyzes access-control systems and detects such information-flow vulnerabilities.[1] NETRA works by statically analyzing a snapshot of the access-control metadata. The architecture of NETRA is shown in Figure 1. NETRA exaggerates the separation between policy and mechanisms in access-control systems—the shaded boxes in Figure 1 remain unchanged when we apply NETRA to different systems. Adapting NETRA to a specific system (say Windows XP) requires writing a scanner that parses a dynamic snapshot of the low-level access-control metadata, and converts it into relational form. Next, a description of the access-control mechanisms as a declarative set of rules is required. Once these two steps are done, NETRA's policy rules can be applied by its inference engine to look for information-flow vulnerabilities.

The core of NETRA is a relational query-inference engine. The metadata, the policy, and mechanism specifications form a deductive database system. All the inputs to the engine, including the access-control relations, the mechanism specification and policy specification are given in the form of declarative rules. These rules are similar to rules in Datalog, with custom augmentations that do not impact the decidability of query-satisfaction for their fixpoint semantics. Each rule is of the following form:

$$\begin{aligned} L(X_1, X_2, \cdots) \quad :- \quad & R_1(X_{i_1}, X_{i_2}, \cdots), R_2(\cdots), \ldots, \\ & \sim F_1(X_{j_1}, X_{j_2}, \cdots), \sim F_2(\cdots), \ldots, \\ & X_k = f(V_{k_1}, V_{k_2}), \ldots \\ & (V_{n_1} \circ V_{n_2}), \ldots \end{aligned}$$

The right-hand side of each rule contains four kinds of predicates: (1) positive predicates $R_1, R_2 \ldots$, (2) negated predicates $F_1, F_2, \ldots$, (3) functions $X_k = f(V_{k_1}, V_{k_2})$, where V_{k_1} and V_{k_2} are either variables or constants, and the function f is an arithmetic operator such as $+$ or $-$, (4) relational predicates such as $(V_{n_1} \circ V_{n_2})$, where V_{n_1} and V_{n_2} are either variables or constants, and $\circ$ is a relational operator such as $\geq$ or $\leq$. Rules can be recursive—the predicate L from the left-hand side can also occur in the right-hand side.

In order to ensure that the fixpoint semantics of the rules are well-defined, we impose the following two restrictions. We note that we are able to specify access control models of Windows XP and SELinux, as well as our information flow properties in this language, in spite of these restrictions.

First, we require that occurrences of negations be "stratified". More precisely, we build a dependency graph with a node for every predicate, and add an edge from every predicate on the right-hand-side of each rule to the predicate on the left-hand-side. An edge is marked as negated if the right-hand-side predicate is negated. We require that negated edges cannot occur within a strongly connected component in the dependency graph. Our inference engine first partitions predicates into strongly connected components, based on their dependencies, and processes the strongly connected components in reverse topological order. Within each strongly connected component, the inference engine runs the rules and generates new facts until a fixpoint is reached. Since negations occur only between, and never within strongly connected components, the fixpoint computation is well-defined.

Second, every variable used in a negated predicate, in the right-hand-side of a function, or in a relational predicate, also needs to be used in a positive predicate in the left-hand-side of a function in the same rule. Due to this restriction, each rule can be evaluated by first doing a "join" operation on the positive right-hand-side predicates, applying the functions, and finally applying the negative predicates and filters on the rows of the resulting tables.

We use a standard bottom-up fixpoint algorithm to evaluate our queries. NETRA's inference engine is custom-built and written in about 1800 lines of F# [19]. Every proof found by the system as a counterexample to our safety property corresponds to a (different) security vulnerability, therefore our engine maintains all possible proofs for every fact it derives. The proofs are maintained along with the infer-

[1] NETRA means "eye" in Sanskrit.

ences, taking care to maximally share sub-proofs. When a vulnerability is discovered, NETRA can display every proof as a DAG (directed acyclic graph), which represents the vulnerability in a graphical form. Due to our separation of rules into policy and mechanism, the graphs can be read hierarchically, corresponding to different levels of abstraction.

In the next section, we show how we can use this formalism to encode the access control functions in Windows XP and SELinux.

3. MECHANISM SPECIFICATION

We explain how we can model Windows XP and SELinux access control using the simple relational calculus introduced in Section 2. We use these examples to demonstrate the expressive power of our language, as well as to highlight the two-level abstraction framework that allows us to apply the same vulnerability specification to analyze different access-control models. In both cases, the input to the model is the OS-specific access-control metadata, and the output is a set of `Read`, `Write`, and `Execute` relations attesting which principals in the systems can access which resources, consistent with their respective model specifications.

3.1 Windows XP Access Control

Windows XP [16] implements the discretionary access control (DAC) model, where only the owner of a resource is authorized to change its access permissions.

Operationally, whenever a a thread (of execution) within a process wants to access a resource (such as a file, directory, thread, kernel object, pipe, socket), a component of the kernel called the *Security Reference Monitor* (SRM) is invoked. The SRM uses an access-check algorithm to decide if the entity can access the resource.

Each thread in Windows XP has a *token* that describes its security context. The token is assigned to a user during logon, and its contents depend on the rights of the user. It contains information about the owner of the thread and its groups. Owners and groups are represented by *security identifiers* (SIDs). An SID may be enabled or disabled for a particular context. In addition, an SID can be marked 'deny-only' or 'restricted', which is explained later. A token also contains a list of privileges associated with the process or thread. Threads inherit their token from the user that creates the containing process. However, the actual token of a thread typically has less rights than the process. Token attributes such as impersonation privileges and restricted SIDs decide the eventual permissions that are granted to a thread. For the sake of simplicity, we do not present all the details of these token attributes in our model description in this section.

Each resource in Windows XP has a *Security Descriptor* (SD) object associated with it. An SD contains, among other things, information about the owner of the object and an entry for the discretionary access-control list (DACL). This DACL can be either "null" (i.e. not present), or it can be an ordered list of Access Control Entries (ACEs). ACEs come in two types: (1) an "allow ACE" describes which entities are allowed access to the resource, and (2) a "deny ACE" describes which entities are denied access to the resource.

Whenever any thread tries to get a handle to a resource, the SRM invokes a function called *AccessCheck* to determine if the access can be granted. The *AccessCheck* function takes three inputs: (1) the SD for the object, (2) the token for the requesting entity or principal, and (3) the type of access requested (read, write, execute, etc). We give an informal description of the access-check function, simplifying some of the details that are not relevant for our presentation. Please refer to [16] for more details.

If the DACL for that resource is missing (i.e. "null"), all accesses to the resource are allowed. If a list of ACEs is present, it will be evaluated to determine the permission. The order of the ACEs matters: the first ACE that matches the request determines the decision. Thus, a deny ACE preceding an allow ACE would mean that the access is denied instead of being granted, and vice versa. Tokens can have restricted SIDs, and if such restricted SIDs are present, then *AccessCheck* runs the algorithm again with new information, as described below.

1. A check is made if the DACL in the SD object is NULL. If so, there are no conditions for access, and the access is granted.
2. The ACEs in the SD object are examined in increasing order of indices, and the following checks are performed for each index i:
   (a) If the ACE at index i is a deny ACE, and the ACE matches an SID in the token presented and the type of access requested, then the access is denied and *AccessCheck* terminates.
   (b) If the ACE at index i is an allow ACE, and the ACE matches an SID in the token and the type of access requested, then the algorithm proceeds with Item 3.
3. If the token does not contain any restricted SIDs, then access is granted and *AccessCheck* terminates. Otherwise, a second pass is made through the ACL and a similar check as Item 2 is made that operates only on the restricted SIDs.

A simplified version of the mechanism layer for Windows XP access control is given in Figure 2. The underlying metadata are obtained by a snapshot of running processes in the system (see Section 7 for more discussion on this design choice).

Given an SID `sid` and a resource `rsrc`, the first rule says that `Read(sid,rsrc)` holds whenever there is a thread owned by the SID `sid` with a token `token`, such that `AccessCheck(token,rsrc,'r')` holds. We represent tokens by the process ID to which they are associated; the relation `ProcessTokenUser` relates tokens and the SIDs of their owners. The variable `rsrc` can be thought of as a unique reference to a resource.

The rules for `AccessCheck` declaratively describe the functionality of Windows XP's *AccessCheck* function. The first rule says that in the case of a "null" DACL on the resource, any access is allowed. The predicates `FirstPass` and `SecondPass` model the two passes of the algorithm. The rule for the predicate `FirstPass` states that `FirstPass(token,rsrc,t)` holds whenever there is an allow ACE at index `i` and no deny ACE up to index `i` for some index `i`. The rule for the predicate `DenyAce` is recursive, and states that if `DenyAce` holds at some index `i`, then it holds at larger indices as well (up to `num`, which is the total number of ACEs for the resource). Thus, we faithfully model

```
Read(sid,rsrc) :- ProcessTokenUser(token,sid), AccessCheck(token,rsrc,"r").
Write(sid,rsrc) :- ProcessTokenUser(token,sid), AccessCheck(token,rsrc,"w").
Execute(sid,rsrc) :- ProcessTokenUser(token,sid), AccessCheck(token,rsrc,"e").

AccessCheck(token,rsrc,t) :- Token(token), NullDacl(rsrc), AccessType(t).
AccessCheck(token,rsrc,t) :- FirstPass(token,rsrc,t), SecondPass(token,rsrc,t).

FirstPass(token,rsrc,t) :- AllowAce(token,rsrc,t,i), ~DenyAce(token,rsrc,t,i).
AllowAce(token,rsrc,t,i):- Ace(rsrc,i,"allow",sid,t), HasEnabledSID(token,sid).
DenyAce(token,rsrc,t,i) :- Ace(rsrc,i,"deny",sid,t), HasEnabledSID(token,sid).
DenyAce(token,rsrc,t,i) :- Ace(rsrc,i,"deny",sid,t), HasDenyonlySID(token,sid).
DenyAce(token,rsrc,t,i) :- DenyAce(token,rsrc,t,d), NumAces(rsrc,num), i := d+1, i < num.

SecondPass(token,rsrc,t) :- NoRestrSIDs(token), FirstPass(token,rsrc,t).
SecondPass(token,rsrc,t) :- RestrAllowAce(token,rsrc,t,i), ~RestrDenyAce(token,rsrc,t,i).
RestrAllowAce(token,rsrc,t,i) :- Ace(rsrc,i,"allow",sid,t), HasRestrSID(token,sid).
RestrDenyAce(token,rsrc,t,i)  :- Ace(rsrc,i,"deny",sid,t), HasRestrSID(token,sid).
RestrDenyAce(token,rsrc,t,i)  :- RestrDenyAce(token,rsrc,t,d), NumAces(rsrc,num), i := d+1,
                                 i < num.
```

Figure 2: Windows XP Access Control Algorithm

the order-dependent processing of ACEs in the algorithm. The relation `Ace` is obtained by parsing the SD metadata, and contains each ACE present in the SD, ordered by the indices. The relations `HasEnabledSID` and `HasDenyOnlySID` model the SIDs associated with a token and are obtained from the token metadata.

The second pass is modeled using the predicate `SecondPass`. The first rule for `SecondPass` says that if there are no restricted ACEs, then the second pass is equivalent to the first pass. The remaining rules for `SecondPass` are analogous to the rules for `FirstPass` with the difference being that restricted SIDs are used.

In the next subsection, we show how to model SELinux access control using our specification language.

3.2 SELinux Access Control

SELinux is an enhancement to the Linux kernel that introduces mandatory access control (MAC) to standard Linux. It is shipped with a number of Linux distributions, e.g. Debian and Fedora. In Fedora Core 5, the *Targeted* configuration of SELinux is enabled by default. SELinux tries to confine each system server and user process to the minimum amount of privileges and rights required for their functioning. Thus, when one of these entities is compromised (e.g., through buffer overflows), their ability to cause damage to the system is reduced or eliminated.

The security architecture of SELinux supports many underlying policy abstractions. These include Type Enforcement (TE), Role-Based Access Control (RBAC) and Multi-Level Security (MLS), a type of MAC. The specific policy enforced by a particular installation is governed by a configuration file. The configuration is specified in a declarative language called "SELinux policy". While one would hope that this policy language would provide higher level abstractions, it is well known that it suffers from a granularity problem and is considered too low-level [20, 7, 8] to express information-flow goals effectively.

SELinux enforcement is built on top of the standard Unix DAC model. In order to grant access, a request has to be first allowed by this underlying model (except when an override option is set). The SELinux mechanisms are used to typically restrict these permissions and refine accessibility in terms of least privilege. The TE component defines an extensible set of domains and types. Each process in an SELinux installation has an associated domain and each object an associated type. Objects types may be further aggregated as classes. The configuration files specify how domains can access types (as a set of access vector rules) and interact with other domains. In addition, they specify transition rules that govern what types can be used to enter domains, as well as allowable transitions between domains, typically by executing programs of certain types (and classes). This ensures that certain programs can be placed in restricted domains automatically, depending on what they execute.

In addition to TE, SELinux provides support for RBAC in terms of an extensible set of roles. Each user in the system can be associated with multiple roles. The configuration specifies which users can enter what roles as well as the set of domains that may be entered by each role. MLS is the standard Bell-LaPadula lattice-based MAC model [17].

In the next subsection, we describe the SELinux configuration language and model it in our notation.

3.2.1 SELinux Targeted Configuration

For the purpose of this paper, we focus on the standard distribution of SELinux that ships with Fedora Core 5. Three different configurations are available: Strict, Targeted and MLS. The Strict configuration was developed by NSA and is meant for a controlled userspace that disallows DAC, but is impractical for enterprise networks. The MLS configuration focuses on servers only. The Targeted configuration that we analyze is intended to lock down specific daemons or processes, based on their vulnerability. These daemons run under the super-user account `root` that usually has full control over the system, but SELinux adds mechanisms designed to restrict them to the least privileges needed to fulfil their tasks. The rest of the system runs with original Linux default permissions. Untargeted processes run in the `unconfined_t` domain. The targeted processes switch to their protected domains when they are executed in the system. For example, `initd` runs as unconfined unless it executes a program belonging to any of the targeted domains.

The Targeted configuration is a combination of RBAC and TE. Its configuration files contain rules that form a declar-

```
Read(domain,resource):-  ResourceType(resource,type), DomReadType(domain,type),
                         ~NeverAllowRead(domain,type).
Write(domain,resource):- ResourceType(resource,type), DomWriteType(domain,type),
                         ~NeverAllowWrite(domain,type).
Execute(domain,resource):- ResourceType(resource,type), DomExecType(domain,type),
                           ~AnyTTR(domain,type), ~NeverAllowExecute(domain,type).
Execute(domainp,resource):- ResourceType(resource,type), ~NeverAllowExecute(domain,type),
                            TypeTransition(domain,type,domainp).

AnyTTR(domain,type) :- TypeTransition(domain,type,domain2).

DomReadType(domain,type) :- AllowRead(domain,type).
DomReadType(domain,type) :- AllowRead(domain,class), TypeClass(type,class).
DomWriteType(domain,type) :- AllowWrite(domain,type).
DomWriteType(domain,type) :- AllowWrite(domain,class), TypeClass(type,class).
DomExecType(domain,type) :- AllowExecute(domain,type).
DomExecType(domain,type) :- AllowExecute(domain,class), TypeClass(type,class).
```

Figure 3: SELinux Access Control Algorithm

ative mechanism specification. The rules define types, domains, roles, associations between roles and domains, access vectors, and domain-type transitions. We point the reader to [18] for more details on SELinux configuration. For the purposes of our analysis, these rules form the access-control metadata that is analyzed by our tool.

From this metadata, we directly extract the following relations:

- `AllowRead`, `NeverAllowRead`, and the analogous write and execute relations: these specify the read, write, and execute permissions that domains have over types (or classes), where the `NeverAllow...` predicates are used to deny permissions that would otherwise be granted by Unix DAC settings.

- `ResourceType` and `TypeClass`: these provide the membership relations of resources in types and of types in classes.

- `TypeTransition`: a fact of the form `TypeTransition (d1,t,d2)` says that if a process of domain `d1` creates a new process by executing an object of type `t`, then the new process will run under the domain `d2`. Note that no new domains or types are created by this rule.

The goal of our analysis is to find whether the targeted daemons in the given configuration run with unnecessary privileges that could lead to vulnerabilities when a daemon is compromised. Since the daemons are running as `root`, they are unrestricted by the Unix DAC model, which we therefore ignore. For the purposes of this presentation, our model treats domains as subjects (processes with the same domain have the same rights). To present this information more meaningfully to developers, we can also look up the role-user and user-domain associations to find vulnerable users, which we omit here for sake of brevity.

The resulting access-control model is specified in Figure 3.2. The `Read` predicate specified on the first line proceeds as follows: (1) The type of the resource is found. (2) We check if we can find an `AllowRead` association for the domain-type pair, either directly, or by virtue of the type being a member of a class for which this is allowed, as specified near the end of Figure 3.2. (3) We check if a never-allow association exists for the same domain-type pair. Access is denied if such an association is found, or if no allow relationship is found.

The rule for `Write` is similar to `Read`. For the `Execute` predicate, in addition to checking membership in allow and never-allow relations, we also check if there is a type transition rule triggering a transition to another domain where the file will be executed. If there is a transition to another domain, the file will actually execute under the new domain.

4. VULNERABILITY SPECIFICATIONS

In this section, we describe how we can use the same specification language to express information flow properties of interest with respect to explicit flows (we do not model covert channels). An interesting point to note here is that this vulnerability analysis is independent of the underlying implementation mechanisms. This specification of vulnerabilities can be evaluated against different access control models, as long as the metadata can be meaningfully expressed as the simplified read, write and execute relations. Later, we show we can use the same specifications to analyze vulnerabilities in both Windows XP and SELinux.

4.1 Information Flow Properties

Flow of information occurs from a resource to a user when a user either reads or executes the resource, and from a user to a resource when the user writes to the resource. While many of these flows are by design, certain types of flows are undesirable. For our analysis, we are primarily interested in flows that go across what are called protection boundaries.

Information-flow analysis of this nature is most useful when we start with susceptible programs, or have access to very sensitive data. Ideally, it should be impossible for susceptible programs that are run with lower privileges to access the same data available to the sensitive programs that run with higher privileges except in very constrained circumstances. Our queries therefore are about flows between lower-privileged users through the set of susceptible programs to the set of sensitive programs that can be accessed by users with higher privileges.

4.2 Vulnerability Specification

In Figure 4 we present specifications of information flows that are undesirable. The variables in the relations specified here will be interpreted in the context of specific models. For

```
WriteExecuteAttack(s1,s2,rsrc):- Write(s1,rsrc), ~Admin(s1), Execute(s2,rsrc), Admin(s2).

IntegrityAttack(s1,s2,rsrc):- Write(s1,rsrc), ~Admin(s1), Read(s2,rsrc), Admin(s2).

ConfidentialityAttack(s1,s2,rsrc):-Read(s1,rsrc), ~Admin(s1), Write(s2,rsrc), Admin(s2).

Tainted(s1,s2):- Write(s1,rsrc), ~Admin(s1), Read(s2,rsrc), ~Admin(s2).
Tainted(s1,s2):- Write(s1,rsrc), ~Admin(s1), Execute(s2,rsrc), ~Admin(s2).
Tainted(s1,s3):- Tainted(s1,s2), Tainted(s2,s3).

TransitiveAttack(s1,s3):- ~Admin(s1), Admin(s3), Tainted(s1,s2), WriteExecuteAttack(s2,s3,rsrc).
TransitiveAttack(s1,s3):- ~Admin(s1), Admin(s3), Tainted(s1,s2), IntegrityAttack(s2,s3,rsrc).
```

Figure 4: Specifications of Vulnerabilities

example, subjects in Windows XP are SIDs and domains in SELinux. Our property specifications are very abstract and simple. However, this does not imply that we do not have control over the level of abstraction at which we can manipulate queries using our tool. As we show in Section 5, we can encode implementation-specific details as filters in the mechanism-specification without changing our queries, and nevertheless improve the relevance of our results.

1. Privilege Escalation (W-E Vulnerabilities): The first rule in the specification is the Write-Execute (W-E) privilege escalation vulnerability. In its simplest form, a W-E vulnerability can be defined as one in which a resource has a write permission by a non-admin user and an execute permission by an admin user. Potentially the non-admin user (by writing malicious code into the resource) can make the admin user execute dangerous code which gives the non-admin user elevated privileges to the system. The rule states that a W-E vulnerability exists between two subjects s_1 and s_2 if one of them, say s_1 has write permissions on the resource, and is not an administrative or high-privileged user, and if the other user s_2 has execute privileges on the same resource. Later, we discuss how not all W-E vulnerabilities can be exploited in the context of specific models, and how we can refine our tool to produce relevant vulnerability reports.

2. Integrity and Confidentiality Concerns: Similarly, we specify an integrity vulnerability as a write-read flow between a non-admin and an admin subject, and an undesirable confidentiality flow as a read-write flow in this context.

3. Taint Analysis: The next specification is for taint analysis. The first and second rules state that a non-admin subject s_1 can taint another non-admin subject s_2 if there is a possibility of a write-read or write-execute flow between them. Furthermore, this relation is transitive if there is a third subject that is the destination of one flow and the originator of another. Taint analysis is useful to explore the potential spread of compromised information (e.g., virus) in a system.

4. Transitive Vulnerabilities: Taint analysis forms the basis for the specification of a transitive vulnerability. A possibility of a transitive vulnerability between a non-admin s_1 and an admin s_3 exists if s_1 can taint s_2 and there is a W-E vulnerability or a W-R vulnerability between s_2 and s_3, as specified.

5. RESULTS

In this section, we discuss the results obtained by running NETRA on both Windows XP and SELinux. Note that the vulnerability results our tool produces are only possible attacks. We show how we can refine our specifications to improve the relevance of vulnerabilities found by adding appropriate filters.

5.1 Results on Windows XP

We run NETRA for the Windows XP specification in Figure 2, together with the vulnerability specifications from Figure 4 and for a test configuration the tool produced 4853 vulnerabilities over 1326 unique resources. Several of these vulnerabilities are benign, and running the tool for the model at this level of abstraction produced a large amount of "white noise". A typical vulnerability report was of the form "User u has privileges to write into a resource r, and an admin a has execute permissions on r." However, even if the admin *can* execute r, it might never actually do so. To make the results of the tool more relevant, we need to look at implementation details so as to produce more plausible vulnerabilities. To do this, we refine the `Execute` rule in Figure 2 to also add that for an admin user to be considered to have a chance of executing a resource `rsrc`, the admin user should also have an open handle for `rsrc`. We therefore push implementation details that improve the usability of our tool to the model level, retaining the power of abstraction at the property specification level. We believe that this choice greatly improves the usability of our tool.

After this filtering, the tool produced 176 vulnerability reports on 58 different resources (multiple attacks on the same resource usually differed only in the identities of the admin and non-admin user involved). Every report we looked at was a plausible vulnerability. Several of these vulnerabilities have been fixed and patched, but these patches may not have been installed everywhere. We give examples of two of these vulnerability reports in Figure 5.

The first vulnerability report (in the top portion of the figure) says that: (1) the `Admin` user has administrative privileges, (2) the `Admin` user has both a handle and execute permissions on `AttackedResource`[2], and (3) `LS` (or Local Service, which is a group with low privileges) is running a process with write permissions to `AttackedResource`. Now, if we ask the question, "How does that process have write permissions to `AttackedResource`?", we can descend down the

[2]The actual name of the resource and the `AdminProcess` have been withheld to prevent exploitations on unpatched machines.

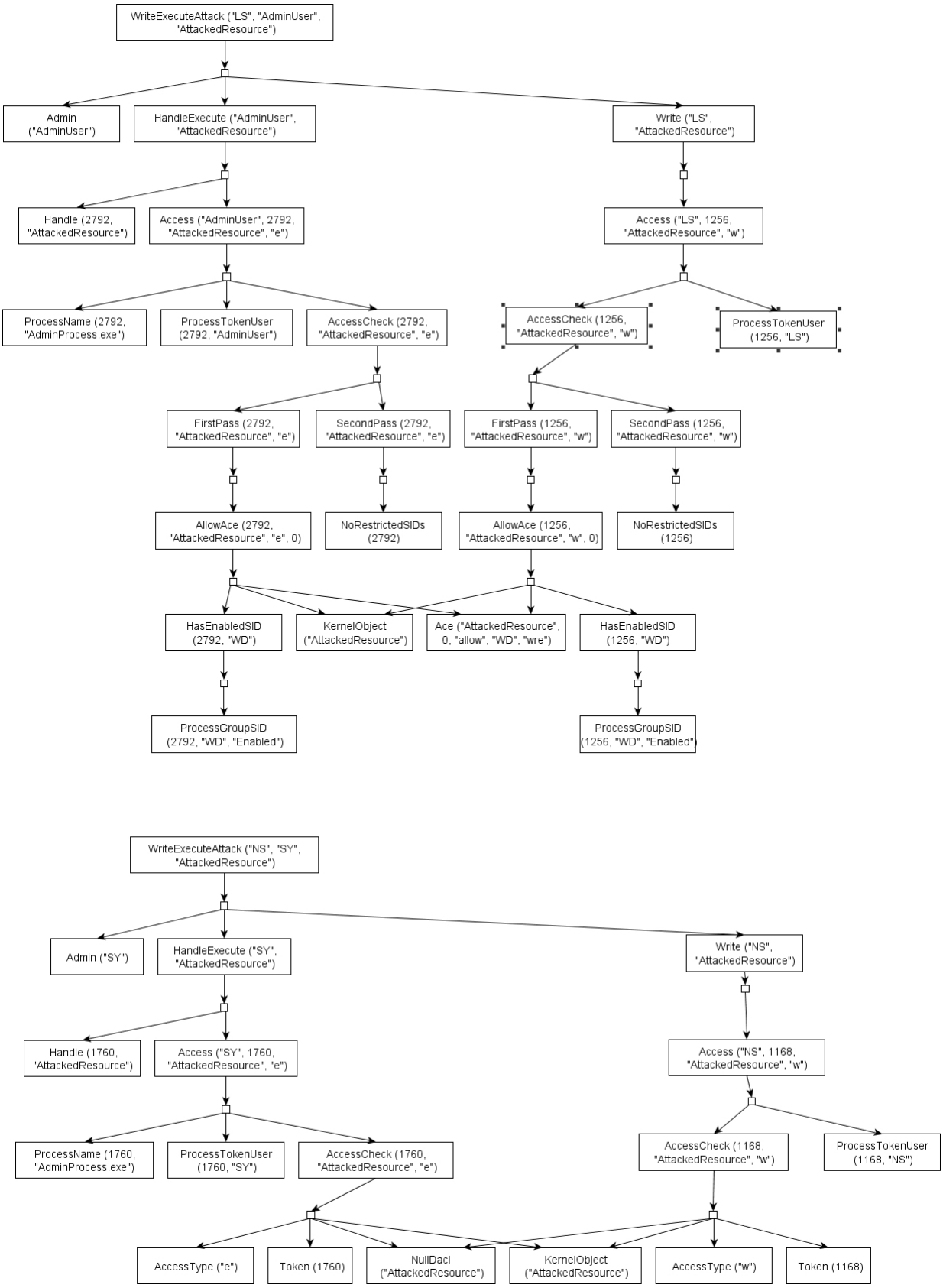

Figure 5: Sample Windows XP Vulnerabilty Reports

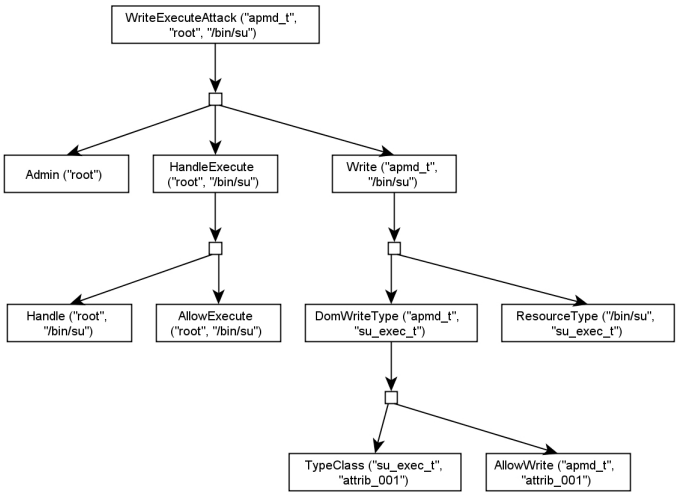

Figure 6: Sample SELinux Vulnerabilty Report

node labeled `Write(‘‘LS’’,’’AttackedResource’’)` and learn that this is due to a process that is owned by LS and has the appropriate token. If we ask the question, "How does this process have access to `AttackedResource`?", we can descend down the tree further, and locate the actual ACE on the security descriptor for `AttackedResource` due to which Windows XP's `AccessCheck` granted this access. The ACE in question actually makes the resource "world-writeable" (denoted by `WD` in the graph), which allows any user, not just LS, to write to the resource.

The second vulnerability report (in the bottom portion of the figure) also shows a write-execute vulnerability, but here the user gets access to the `AttackedResource` due to a "null" DACL.

5.2 Results on SELinux

As mentioned in Section 3.2, we analyze the Targeted configuration of the standard distribution that ships with Fedora Core 5. To discover whether the targeted daemons run with overly permissive rights, we check whether they can compromise files used by an unrestricted `root` process. Since such processes are not directly represented in the SELinux configuration, we added an artificial, all-powerful 'root' domain to the configuration, which, for the purposes of the vulnerability specification, was considered to be an admin. Like in our experiments on Windows, we improve the usability of our tool by adding a filter at the model level, i.e. we specify that only vulnerability reports on certain files would be considered interesting. For the purpose of our experiments, we searched for attacks on the `su` binary that is used for logging in as a super user. In general, a broader scope for the search would be desirable, e.g., one could look for attacks on all system executables whose 'setuid' attribute is set.

With this configuration, NETRA reported 26 different domains in the SELinux configuration that had write access to the `su` binary. A sample vulnerability report is shown in Figure 6. In this example, a process of type `apmd_t` can write to a binary of type `su_exec_t` (which includes `su`) by virtue of having write access to the class `attrib_001`, of which `su_exec_t` is a member. An example of a process running as `apmd_t` is the `acpid` daemon, whose purpose is the management of Advanced Configuration and Power Interface (ACPI) events.

We believe that the permission to rewrite security-critical binaries is not required for the operation of this daemon, and that the same holds for almost all the 26 domains found by NETRA. If a daemon running under any of the above domains is compromised (e.g., by a buffer overrun), an attacker could, e.g., trick the daemon into rewriting the `su` binary to remove the password check, thus giving himself full access to the system, a kind of vulnerability that SELinux was actually designed to prevent.

To verify that our conclusions are correct, we conducted a proof-of-concept experiment in which we simulated the effect that a compromised service would have. We modified `acpid` so that it would replace `su` with a different binary when starting and then carry on with its usual service. Indeed, SELinux failed to prevent this attack. When we repeated the experiment with another daemon on which our tool did not report a vulnerability (`sdpd`, a Bluetooth daemon), SELinux did prevent the attack. These facts suggest that the overly permissive rules were not given deliberately, but indeed represent a configuration flaw.

6. RELATED WORK

Formal modeling and analysis of access-control properties in operating systems has a long history [14]. The correspondence between an access policy and the meaning of its associated mechanisms is expressed as the access-control safety property. This amounts to analyzing whether a user can obtain a permission he/she was not authorized to obtain, using the implementation mechanisms for a given model. Validating access control safety in its most general sense is undecidable [6]. However, there are restricted and abstract models of access control such as [11, 2], where some properties can be validated efficiently.

In addition to access-control safety, researchers have also

looked at access control properties from the viewpoint of users in a system. The three standard properties here are confidentiality, integrity, and availability. These properties are also referred to as explicit information-flow properties. A variety of models with formal guarantees for confidentiality and integrity properties have been proposed. However, such models, for instance, for preventing information disclosure in multi-level sytems [1, 13] are not very popular outside constrained userspaces and have been shown to be unachievable in practice [12] due to covert channels.

We note that explicit information-flow vulnerabilities are not the only kinds of problems associated with access control systems. Other vulnerabilities include physical attacks, covert channels, user negligence, programming language safety, memory access safety, etc., that are beyond the scope of this paper. Nevertheless, this is an important class of vulnerabilities that we hope can be fixed by rigorous design and analysis.

The simple access-control models of Unix and the original Windows operating systems have evolved over time, in step with a demand for flexible and finer-grained control over access. As described in this paper, commercial operating systems as well as open-source systems now support a wealth of mechanisms that allow policy engineers enormous flexibility in expressing information-flow requirements. However, policy models seem to have lost their effectiveness as a useful abstraction in this context.

In the context of SELinux, this is acknowledged as a serious concern, and a number of researchers have studied the problem of validating different SELinux configurations against higher-level goals [20, 7, 9, 8]. Most of these tools still present a low-level abstraction to their users and typically allow them to browse whether some permission or transition is allowed or not.

The work most closely related to our effort is [5], which presents a formal framework for verifying information-flow goals in SELinux. From a given SELinux configuration, they construct a labeled transition system (LTS) with security contexts as states and read, write, and execute actions as transition labels. The policy is specified as a desirable property of states (e.g., as an assertion that somebody running a HTTP script file cannot become root). This specification is checked against the LTS using the NuSMV model-checker as the analysis engine. A counterexample gives a path that leads to a state where this property is violated.

In contrast, our work is more general. We are able to specify information-flow goals at a higher level of abstraction that is closer to human intuition. The specifications themselves are programmable, and we can produce all counterexamples. Furthermore, our two-layer framework also allows us to plug in different access-control models and hide the complexity of underlying implementation mechanisms from a policy engineer.

This work was inspired by an internal tool used within Microsoft that analyzes privilege-escalation vulnerabilities [10]. In the original version of that tool, the analysis was mechanism-driven and patterns of known vulnerabilities were matched against configuration information. Our goal was to generalize this analysis to other information-flow properties as well as to make it extensible and programmable in terms of mechanisms, models and queries. We were able to add integrity checking, confidentiality, checking, and taint-analysis by writing only a few lines of specification. Another tool that has the same inspiration is [4], which is implemented using a similar logic-programming framework, but does not have the two-level abstraction, and uses only known attacks to model vulnerabilities. Logic programming is generally recognized as a natural way of expressing security goals.

7. CONCLUDING REMARKS

The goal of any access-control model is twofold: (1) to allow principals to share resources and communicate with each other legitimately for their functionality requirements, and (2) to disallow bad information flows that compromise integrity and confidentiality.

Mechanisms that implement access control can be complicated, and it is very easy to misconfigure a system with exploitable information-flow vulnerabilities. We have demonstrated that it is possible to capture these implementation details as well as desirable properties using our specification language, and validate these properties in a platform-independent fashion. We do this by demonstrating that two disparate operating systems such as Windows XP and SELinux can be analyzed using the same property specifications. Our tool NETRA successfully found vulnerabilities in both systems. NETRA can be extended to validate other access control models in a similar fashion.

Our analysis captures a dynamic snapshot of the access-control metadata. For Windows XP, this snapshot give us precise information only about the current state of the subjects, objects and permissions in the system. For SELinux, the configuration information also includes some assertions about their future states. In both cases, this analysis is neither sound nor complete. For a given snapshot however, we can argue that we find all information-flow vulnerabilities as long as our model of the mechanisms is faithful to the actual implementation. We believe the model can be validated over time (by analyzing source code for instance) and a correct specification can emerge by consensus.

Our current analysis is incomplete because the errors we find are only possible vulnerabilities, which may not be exploitable. This analysis is useful only if it does not miss important vulnerabilities and if it has a low false-error ratio. With NETRA we have found that one way to reduce false errors is to add filters to the rules that select specific tuples with relevant attributes from the underlying implementation relations. For example, for Windows XP we were able to reduce false errors drastically by adding that an admin process that is the target of a bad information flow can also have an open handle to the resource in question. Adding too much precision would reduce the functionality of our tool to that of a reference monitor. Using a purely static approach it would be difficult, if not impossible, to enumerate all the security states of the system to guarantee completeness. Any approach that is likely to be useful is therefore a combination of the two. Finding the right balance between static and dynamic analysis is important, and running our tool as a daemon that incrementally analyzes the configuration information whenever the state of the system changes is one option. We plan to investigate these directions in future work.

8. ACKNOWLEDGEMENTS

We thank Aditya Parameswaran, Aditya Nori, and Yamini

Kannan for their help with an earlier version of the NETRA tool. We also thank Neill Clift, Ramarathnam Venkatesan, Venky Ganti, Jim Larus, and Chandu Thekkath for helpful discussions. We thank Joseph Joy, Aditya Nori, Avik Chaudhuri, and Chris Conway for their comments on this paper.

9. REFERENCES

[1] Bell, D., and LaPadula, D. Secure computer systems: Mathematical foundations. Tech. rep., MTR-2547, Volume I, Mitre Corporation, 1993.

[2] Bishop, M., and Snyder, L. The transfer of information and authority in a protection system. In *Proc. SOSP* (1979), ACM Press, pp. 45–54.

[3] Chen, S., Dunagan, J., Verbowski, C., and Wang, Y.-M. A black-box tracing technique to identify causes of least-privilege incompatibilities. In *Proc. NDSS* (2005).

[4] Govindavajhala, S., and Appel, A. Windows access control demystified. Tech. rep., Princeton University, 2006.

[5] Guttman, J. D., Herzog, A. L., Ramsdell, J. D., and Skorupka, C. W. Verifying information flow goals in Security-Enhanced Linux. *J. Comput. Secur. 13*, 1 (2005), 115–134.

[6] Harrison, M. A., Ruzzo, W. L., and Ullman, J. D. Protection in operating systems. *Commun. ACM 19*, 8 (1976), 461–471.

[7] Herzog, A., and Guttman, J. Achieving security goals with Security-Enhanced Linux. Tech. rep., Mitre Corporation, 2002.

[8] Hinrichs, S., and Naldurg, P. Attack-based domain transition analysis. In *2nd Annual Security Enhanced Linux Symposium* (2006).

[9] Jaeger, T., Zhang, X., and Cacheda, F. Policy management using access control spaces. *ACM Trans. Inf. Syst. Secur. 6*, 3 (2003), 327–364.

[10] Lambert, J. Security analysis. Personal communication.

[11] Lipton, R. J., and Snyder, L. A linear time algorithm for deciding subject security. *J. ACM 24*, 3 (1977), 455–464.

[12] Loscocco, P. A., Smalley, S. D., Muckelbauer, P. A., Taylor, R. C., Turner, S. J., and Farrell, J. F. The inevitability of failure: The flawed assumption of security in modern computing environments. In *Proc. NISSC* (1998).

[13] McLean, J. A comment on the 'basic security theorem' of Bell and LaPadula. *Inf. Process. Lett. 20*, 2 (1985), 67–70.

[14] McLean, J. The specification and modeling of computer security. *Computer 23*, 1 (1990), 9–16.

[15] Ramakrishnan, R., and Gehrke, J. *Database Management Systems*. McGraw-Hill Science/Engineering/Math, 2002.

[16] Rusinovich, M. E., and Solomon, D. A. *Microsoft Windows Internals, Fourth Edition: Microsoft Windows Server 2003, Windows XP, and Windows 2000*. Microsoft Press, 2005.

[17] Sandhu, R. S. Lattice-based access control models. *Computer 26*, 11 (1993), 9–19.

[18] Smalley, S. *Configuring the SELinux Policy*. NAI Laboratories, 2005.

[19] Syme, D. F#. http://research.microsoft.com/fsharp/fsharp.aspx.

[20] Tresys Technology. *Apol:SE Policy Tools for SELinux*. http://www.tresys.com/selinux/selinux_policy_tools.shtml.