Model-based validation of enterprise access policies

Sandeep Bhatt, William Horne, Joe Pato, S. Raj Rajagopalan, Prasad Rao
Trusted Systems Laboratory
HP Laboratories Princeton

HPL-2005-152(R.1)
January 3, 2006*

security, access control, policy, validation

Coordinating security seamlessly across an enterprise is a challenge. Enterprises deploy multiple access control mechanisms at different technology layers; each mechanism is painstakingly configured and maintained using specialized user interfaces, most likely by different administrators in different organizations at different sites, perhaps employing different notions of users and roles. This piecemeal approach makes security management labor-intensive and, therefore, expensive, error-prone and slow to adapt.

We present a model-driven technique for automated policy-based access analysis. Based on the ideas presented in this paper, we have built a prototype, the Integrated Security Management (ISM) system which, given the security configurations of hosts, applications and network devices, automatically validates whether the enterprise is in compliance with high-level enterprise access policy. The system relies on composable models that capture the access control semantics of applications, middleware and devices, in a manner that enables efficient enterprise-scale analysis.

* Internal Accession Date Only

Model-based validation of enterprise access policies

Sandeep Bhatt, William Horne, Joe Pato, S. Raj Rajagopalan, Prasad Rao
Trusted Systems Lab, Hewlett Packard Laboratories
Princeton, NJ 08540

Abstract

Coordinating security seamlessly across an enterprise is a challenge. Enterprises deploy multiple access control mechanisms at different technology layers; each mechanism is painstakingly configured and maintained using specialized user interfaces, most likely by different administrators in different organizations at different sites, perhaps employing different notions of users and roles. This piecemeal approach makes security management labor-intensive and, therefore, expensive, error-prone and slow to adapt.

We present a model-driven technique for automated policy-based access analysis. Based on the ideas presented in this paper, we have built a prototype, the Integrated Security Management (ISM) system which, given the security configurations of hosts, applications and network devices, automatically validates whether the enterprise is in compliance with high-level enterprise access policy. The system relies on composable models that capture the access control semantics of applications, middleware and devices, in a manner that enables efficient enterprise-scale analysis.

1. Introduction

How can an organization verify that access controls implemented on host, application and network infrastructures match the business intent specified by an organization's access control policies? The semantic gap between these policies and the low-level device-specific access controls complicates the verification problem. According to the Aberdeen report "Security Policy Automation in the Enterprise," … "*IS buyers are mired in tuning technology knobs that bear little, if any, relationship to the business policies and procedures employed by the enterprise.*"

Enterprise access policies constrain access to services depending on the identity and role assigned to the user, the method of access, and the actions to be performed. An example of an enterprise access policy is "Users authenticated as employees may change their personal information via the corporate self-service portal." A financial institution's policy may state that "Analysts must be denied access to investment banking data." An example of a Segregation of Duties policy is "No employee can both create and approve a purchase order."

Implementing a high-level policy requires carefully setting low-level access control rules on network devices, servers and applications throughout the enterprise. Unfortunately, systems administrators do not currently have tools that relate rules on their individual system to high-level policies. The result is that administrators are wary of making any change lest it violate business security policy or make a critical service inadvertently inaccessible.

1.1 An Example

To illustrate the nature of the problem, consider a 3-tier web site architecture consisting of a perimeter firewall protecting an Apache web server, a Tomcat servlet container to host web applications, a backend MySql database server and an LDAP server. A large number of configuration parameters restrict access in this simplified example:

1. The perimeter firewall rules determine whether the client request will reach the machine hosting the Apache server.
2. Apache server configurations will determine if, and where, the user credentials are checked.
3. LDAP directory data map users to roles.
4. Apache mapping rules use these roles to determine where the request is forwarded, and with what credentials.
5. Tomcat configurations determine if, and where, user credentials are checked, and which web apps and database proxies the request is allowed to invoke.
6. Within tomcat, the database proxies are configured with user credentials to establish connections to specific databases.
7. In the tomcat engine, java security policy (Catalina.policy) configurations specify permissions granted to web applications for reading or writing to the file system, and opening socket connections, etc.
8. Web applications hosted on Tomcat may have their own notion of users, credentials, and use application specific filters to permit or deny requests, independent of Tomcat configurations.
9. MySql configurations determine if the server accepts network connections, and if it checks permissions for establishing connections and running queries.
10. The privileges configured within the MySql database tables determine whether specific queries are to be run or not.
11. Finally, the file system is configured with its own access control mechanisms.

Every device and application must be configured carefully so that information flows exactly as intended: desired accesses must not be blocked, while undesired accesses must not get through. As a simple example, consider a scenario in which all servers are initially hosted on a single machine.[1] One of the system-wide security policies states that no unauthorized user may connect to the database. Setting the mysqld configuration parameters `skip-networking` and `skip-grants-tables` both `ON` ensures that processes local to the machine, which are all trusted, can open connections to the database. Now, suppose that the Tomcat and MySql are migrated onto two different machines. Tomcat's db proxy must be reconfigured to open a connection to the new location of MySql, and `skip-networking` must be turned `OFF` so that MySql will allow incoming connections. At this point, all previous accesses will be available. But undesirable accesses are also made available, for example, because skip-grants-tables is still on, mysqld serves requests without checking the credentials of the querying entity. Thus, mysqld will execute queries received from any client, including those who should not be permitted by policy to access the database.

This simple example reveals the complexities of administering information access. Several factors conspire to make this difficult:

1. Information flows are complex. For example, while the initial request from a user to a portal might contain user roles and credentials, subsequent requests issued by the web server to the backend application server and to the database, will contain credentials not of the original user, but of software components issuing the requests. As a result, the end application is unaware of the identity of the client who initiated the chain of requests. Poor coordination of component configurations could result in either a legitimate access being blocked or an illegitimate access allowed. For example, a user may be denied

[1] This example is deliberately simplistic for ease of illustration; the techniques presented in this paper apply generally to larger and complex architectures.

access to query a database, but if the user is allowed to access a web application hosted on Tomcat, and the web application is allowed to query the database, then the user has a "backdoor" into the database. Each individual access is locally authorized, and therefore will not trigger an intrusion detection alarm, but the net effect violates the global intent to block the user from accessing the database.

2. A small configuration change in one component can have a significant impact on the set of possible information flows in the system. This is because the security properties depend on interactions between components. There are few, if any, tools to analyze the system-wide impact of a change. Penetration testing cannot uncover all possible security violations resulting from misconfigurations, only the ones known and tested for.
3. In large enterprises, the responsibility of administering networks, devices and applications is distributed across administrators in different organizations and sites. Changes occur organically in a decentralized manner, and while change procedures may be documented and followed, these cannot anticipate all system-wide effects. In many cases, these procedures are cumbersome and make the enterprise slow to adapt to changing business requirements.

The net result is that the systems evolve to allow undesirable information flows without anyone being aware of the possibility. These flows may be caught by intrusion detection systems, but only if the security gap is actively being exploited and monitored. It would be far better to detect the potential violation before that happens.

1.2 Our Goals

The goal of our work is to investigate the feasibility of automatically detecting possible access violations resulting from system misconfigurations. Such a mechanism could be used proactively to help plan and analyze configuration changes before they are deployed, and, in monitoring mode, to audit live systems for access loopholes before any violations occur. Our vision is that a security administrator will only have to specify the desired behavior of a system at a high level of abstraction and our mechanism will enforce that behavior throughout the system lifecycle.

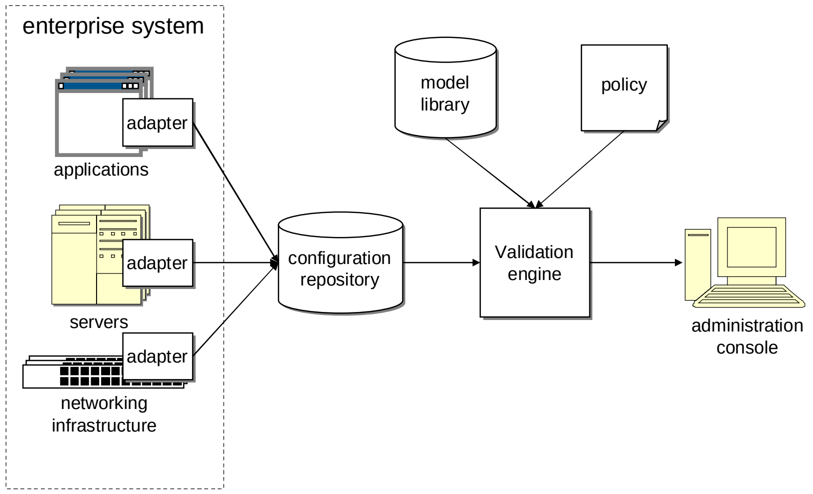
The remainder of this paper is organized as follows. Section 2 outlines the technical issues raised and gives a high-level overview of our system architecture. Section 3 formulates the problem, and develops the technical details with examples. Section 4 presents an overview of the policy validation algorithm. Sections 5 and 6 conclude with extensions and related work.

2. Our Approach

From the business perspective, an enterprise possesses information assets such as data on employees, customers, products, partners and suppliers, and applications that manipulate these assets. Access to enterprise data depends on the identity and role assigned to the user, the method of access, and the action to be performed.

From the administrative perspective, an enterprise consists of components, some of which store enterprise data and others host applications to access and manipulate the data. A component can represent a physical device such as a router or server; it can be an operating system, file system, database or other infrastructural software; or it can be application software such as a web application for example. A component's configuration determines how it processes incoming requests, and which other components it interacts with. These configurations viewed together specify overall system behavior; that is, they determine which users can access and manipulate specific data sets in specific ways. The crux of the administrative problem is to configure the components so that the overall system behavior matches the executive policies.

The high-level conceptual architecture of our system is shown below. It consists of four main modules.



Conceptual system architecture

1. **Adapters** that read and write configuration parameters from enterprise components, and translate these into a standard format for analysis. These adapters can be custom scripts, or interfaced with existing commercial asset and configuration management tools, for example HP OpenView, CiscoWorks, etc.
2. A library of **models** for system components which define the behavior of each component as functions of their configuration parameters.
3. A set of enterprise security **policies** represented as access invariants.
4. The **analysis engine** which, given the system configuration and policies, validates whether the system is configured to satisfy policy and, if not, suggests new configuration values to restore the enterprise to a policy-compliant state, when possible.

We have prototyped the analysis engine, as well as models and adapters for components such as the Apache web server, Tomcat servlet container, specific JSP applications, MySql database server, network file system, LDAP, and basic firewalls. Initial tests to demonstrate the feasibility of our ideas and the performance of the analysis engine are encouraging. We do not delve into the engineering details of the prototype or the experiments in this abstract, but these will appear in the final version.

2.1 Technical Overview

Resolving the following questions is crucial to the feasibility of our approach.

1. What analysis do we perform?

2. What is a model and how is it represented?
3. Is the analysis efficient enough to scale to the complexity of an enterprise?

In the following sections, we outline our answers to these questions along with some examples of how our approach can be effective in solving the enterprise security problem.

2.1.1 Analysis

We briefly describe the kind of analysis we wish to perform. The enterprise environment that we wish to analyze is composed of entities or components such as applications, servers, clients, firewalls, etc. Each entity contains a set of attributes, each of which is assigned a value when the entity is configured. We call the set of attribute-value pairs the "configuration data" for the entity. Suppose that we are given all the configuration data for all the entities in an enterprise. We wish to answer questions of the form: "will requests from client C to entity E to perform operation O with arguments A succeed, or will the request be denied?"

Our analysis is bounded by the following constraints:

1. Our analysis is *memoryless*: the set of allowed operations in any snapshot depends on the configuration state at that point in time, but not on the temporal history of the system.
2. Our analysis is *static*: the configuration values represent a snapshot of the system at one point in time. Our analysis tells us who can access what services at this point in time. We do not analyze sequences of events and state transitions over time. Instead, our analysis can be conducted periodically, using different snapshots of the system in time. Furthermore, we do not look at actual traffic. Our analysis is restricted to what can be deduced solely by looking at the configurations and inferring all the possible traffic the configurations allow. As a result, our analysis can only conclude whether a violation can occur in the current state of the system, not if it has occurred.
3. Our analysis is restricted to security "holes" created by inconsistent or mistaken configurations that can be exploited by an honest-but-curious adversary who tries to gain access to assets by using operations that are permitted to her, but does not try to break the system – say by a buffer overflow attack or by guessing passwords. Thus, we are not looking at attacks that change the behavior of a component by exploiting its vulnerability. Rather, our approach to vulnerabilities is simply to update the model for a component with different behavior that captures the vulnerability.

This formulation of the access validation problem resembles compiler code analysis -- both problems are statically defined, and both involve reachability analysis – in one case to track control flows, and in our case to more general information flows. Although the details are different, our problem is similarly amenable to efficient algorithmic treatment.

2.1.2 Modeling

For each entity we create models that operate as follows. The model of a component describes all possible information flows through that component as a function of its configuration parameters, starting from each specific type of request it receives to the response that the component sends back to that request.

A model is an abstraction of an entity and as such only approximates its behavior. For security analysis, we require that our models be conservative: if the model says that a client request is blocked, then it must never succeed in the real system with the same set of configurations. The converse need not be true; if the model says that a client request is not blocked, in practice the request may fail depending on the configurations of non-security parameters and decision logic that is not captured in the model as well as dynamic conditions such as congestion or failure in the environment that are unknown at analysis time. When each model is conservative we can guarantee that our analysis over a set of models representing an enterprise will also be conservative.

The first design decision to be made in creating a model is: Of the dozens of configuration parameters that a component has, which ones do we include in the model and which ones can we ignore? Such parameters include machine settings, OS parameters, and application specific settings. Examples of configurations that do affect the flow of information are: access control rules (on routers, firewalls, file systems, data bases, web servers, etc.), and connectivity settings (i.e. which components talk with each other). Parameters that are not of interest to us are those that determine performance but do not change the existence or absence of information flows, such as cache sizes or timeout values. Because of our snapshot analysis, an additional consideration is that configuration parameters of interest remain under administrator control and change infrequently relative to the periodicity of our analysis. Thus, for example, variables which change frequently and automatically are not modeled.

In general, deciding whether or not a configuration parameter affects access control behavior and information flows is an art. From our experience, the considerations outlined above enable us to formulate component models quite naturally and simply.

2.1.3 Complexity

The computational complexity of our analysis is determined by the representations we choose for our models – in particular, on the expressive power of the modeling language. The greater the expressive power, the easier it will be to formulate models for entity behavior, but the greater will be the complexity of performing our analysis. The key is to find the right tradeoff between expressive power and complexity of analysis.

We represent models as predicates defined in a way that is equivalent to the logical power of datalog [Imm]. This choice is key, since the complexity of evaluating Datalog queries is known to be polynomial time. There are several systems that evaluate datalog programs very efficiently, and that scale to large problem instances. In our initial prototype, we use the XSB Prolog engine [XSB] which evaluates datalog programs using tabled evaluation – a technique equivalent to bottom-up evaluation. As we shall see, the main part of the verification consists of computing transitive closures over monotonic relations; since this can implemented directly, we are not dependent on XSB Prolog.

3. Enterprise Access Policies

3.1 Terminology

We first establish basic terminology and definitions. Enterprises contain entities of various types – firewalls, database servers, and file systems for example. Each entity type has an associated model that specifies its information flow as a function of the configuration parameters of the entity type. An enterprise consists of a set $\{E_1, \ldots, E_n\}$ of configured entities, where a configured entity is an instance of an entity type with all its configuration parameters given concrete values. A request from a client C (possibly with an attached role, IP address, etc) to a configured entity E is a pair O, A consisting of an operation and an argument. This request to E is either blocked, or else a result R is returned to C.

We define the predicate *allows*(E, C, O, A, R) which is true if and only if when client C sends entity E the request O, A, then E successfully performs the requested operation (i.e., it is not blocked), and result R is returned to the client.

A service S is a subset of C x O x A x R. The service S_E provided by entity E is defined to be the set {(C, O, A, R) | *allows*(E, C, O, A, R) = true}. Thus every entity E provides one service, which consists of a set of tuples each of which represents a client C performing an operation O on an argument A that stands for a resource governed by the entity.

In order to provide a service to a client, an entity may itself depend on other services. For example, an apache server that provides http service may depend on a file system and an application server (which itself depends on a database). Our intention is to have the predicate *allows*(Apache, Client, Get_http, URL,-) evaluate to false if either the apache server is configured to block the client access, or if the file system blocks the apache request, or if the database blocks the application server request, etc. For the predicate to be true, no dependent service must be blocked by an acl setting. On the other hand if, say the acls are set properly to allow all necessary information flows, but the named url does not map to an existing file name or an application, then the predicate must evaluate to true, with the return value being an error message.

3.2 Access Policies

The key to policy-driven enterprise access management is to directly capture the high-level intent of the network as policy statements that are precise, simple, declarative, and free of implementation details. Our enterprise access policy specifies who (by role) is allowed (alternatively, denied) access to what enterprise services. Declarative policies allow us to declare intent without tying them to specifics of network and application implementations, which are subject to change. We require that policy statements be simple so that (i) it is efficient to verify whether or not the system is compliant, and (ii) maintaining a set of policies is not unnecessarily complex for humans to specify and manage. Indeed, allowing arbitrarily complex policies can make it intractable to reason about the net effect of even a small policy set.

Although, without loss of generality, we use the terms such as Client and Entity in our discussion, our assumption is that enterprise access policies will be defined using hierarchical definitions of roles, resources and services. This simplifies the problem of defining access policies across an enterprise. A standard RBAC framework can be overlaid atop our policy validation framework.

There are two basic types of access policy:

permit(Entity, Client, Operation, Argument), and
deny(Entity, Client, Operation, Argument)

Complex policies are represented using boolean combinations of the basic permit and deny policies. A permit policy *permit(*E, C, O, A*)* is satisfied when the service S_E provided by the configured system (set of requests for which the allows predicate for specified client and entity is true) is a superset of the service S specified in policy. In other words, every client permitted by policy to invoke a service on an entity must be able to do so by sending a request directly to that entity. An example policy is *permit(employee , corporate-portal, get, self-service.htm).*

The semantics of a deny policy are necessarily different. A deny policy *Deny(*E, C, O,A*)* is satisfied if and only if there is no request that client C can invoke on any entity, which can, via a chain of requests, cause some client (perhaps a different one) to invoke a request in O x A on entity E which is also allowed by E. In other words, satisfying a deny policy requires us to rule out the possibility of *transitive access* via all possible sequences of steps, including those involving intermediate applications. An example deny policy is *deny(contractors, corporate-portal, get, self-service.htm).*

Note that, under the above definitions, a deny policy is stronger than the negation of the corresponding permit policy. A permit policy can be violated if any of the following conditions holds: the client cannot send a request to the service, a filter rule denies the client request, or the service is unavailable because one of the dependent services is down. On the other hand, to satisfy a deny policy, there must be explicit filter rules that block the client from, directly or indirectly, sending the request to the service.

Policy Conflict Resolution. In a complex environment, it is reasonable to expect that some policies will be in conflict, so that a conflict resolution mechanism is necessary. A simple solution is to require that all policies be prioritized. Other solutions are to add a priority value to each policy along with a tie-resolution algorithm. Such solutions can be easily accommodated in our approach. For clarity, we will assume a total ordering on the set of access policies.
The ordering determines the exact set of services available to all clients. For example, the deny policy *deny*(*employee, finance_server, read, all*) will block all requests from employee clients which can transitively result in a read operation on any file on the finance server. But suppose we want employees to be able access the file via an application that can read the files. How do we express the policy that employee should be able to reach the finance_server but only via the stated application? We can state the positive policy as *permit(employee, finance_app, my_salary, _)* before the deny policy. Because the permit policy is more specific than the deny, all compliant configurations will allow the client (employee) to access the finance_app which, in turn, will be allowed to read on finance server. But all other paths from the employee to the finance server will be blocked.

Default-Deny. Any service that is not explicitly permitted by a policy or required to support a permit policy is automatically denied. Default deny does not obviate the need for explicit deny policies – for example, if we wish to permit all but a subset of clients access to a service, the straightforward solution is to write a specific deny policy for the subset, followed by a more general allow policy for all clients.

3.3 Model Specifications

Each model is specified using three predicates: *attributes, allows, and triggers.* The first predicate, *attributes,* is used to specify the name, type and values of all configuration parameters of the entity. The second predicate, *allows,* captures the set of possible conditions under which the entity will respond to a client request with a specific result. The third predicate, *triggers,* models the conditions under which one request to an entity can lead to subsequent requests by the entity to other entities.

3.3.1 Specifying attributes

Attributes are specified in the form:

attributes(*entityName,* [*attrtype*(*attributeName, attributeType, defaultValue*)]),

where entityName is the entity being modeled, and the second argument is a list of attribute name, type, and optional default value triples. A type is either a standard type, such as integer, string, etc., or the name of another attribute.[2] This enables us to define complex attributes such as a list consisting of access control rules. Such attributes can be viewed as extensional relations (see [Imm]) of our models and represent the ground facts for our system. For example, if the type of the attribute `installedApps` is a list of installed applications, then the predicate installedApps(Oid, App) is true if and only if *App* appears in the list referred to by the installedApps attribute.

There are two special attributes that a model can have: *filter* and *transform*. The filter predicate refers to an extensional relation that contains all the access control rules for the entity. The adapter for the entity is responsible for extracting the acls from the configuration file, converting it into normalized form, and creating the filter table. The transform predicate refers to a table which maps parameters of incoming requests into parameters of outgoing requests.

3.3.2 The "allows" predicate

The first predicate *allows*(E, C, O, A, R) is true if and only if the request O, A from C to a configured instance E will succeed with result R returned to the client.

Each predicate must be defined to cover all types of clients and requests that are modeled. For example, if an entity responds to client identified by any combination of host name, userid, and credential (password, token or cookie) then the "allows" predicate must be defined over all combinations, using multiple rules as necessary.

The conditions which appear on the right side of a rule contain configuration parameters of the entity, which may be simple attribute values or may be a table (an access control list for example). The right side may also contain variables that are uninterpreted, for example, client attributes or credentials, or arguments to an operation. Uninterpreted variables help determine abstract properties that cannot otherwise be determined statically, such as, whether the service request will be authenticated at least once somewhere along the path of the request and response. The right side can also contain an "allows" predicate when, for example, the server must invoke another service to fulfill the request.

As an example, we present a model for the web server apache below consisting of two rules to define the "allows" predicate. The first argument identifies the "flow label," which we describe in Section 3.3.5; the second argument identifies the modeled entity, the third identifies the client, and the remaining arguments identify the requested operation, arguments passed and the returned result. The syntax follows the conventions of prolog – commas between predicates denote "and," strings beginning with lower case letters denote constants, and those beginning with upper case characters denote variables.

The model invokes the filter and transform predicates apacheFilterAllows and apacheXform. These predicates are lookups into relational tables that are produced by the apache adapter which reads the instance configuration files and parses and normalizes relevant directory and location data. In database terminology, these are extensional relations. The model also invokes nwDirectory, a predicate that serves as a directory of network services.

Apache first consults its filters to determine whether the request made by the client is allowed or not. If yes, apache either serves a request for a path by invoking an operation on the file system of the machine on which it is installed (the first rule), or it forwards the request to one of the

[2] We do not allow recursive type definitions.

modules installed on it (the second rule). The decision on whether or not to forward the request to a module is performed by looking up a transform table (this table is created by the adapters by parsing entries such as jkMount from the apache config files).

```
allows(Label, ApacheOid, client(_Realm,ClientId,ClientLoc),
        Op, url(_Proto, _Loc, Path, _Args), ok) :-
      object(ApacheOid, apache,_), attr(ApacheOid, filters, Filters),
      apacheFilterAllows(Filters,ClientId, ClientLoc, Path, Op),
      attr(ApacheOid,xforms, Xforms),
      apacheXform(Xforms, Path, undef),
      attr(ApacheOid, filesystem,Fs), attr(ApacheOid, osusername,Uid),
      allows(Label, Fs, Uid, Path, read, ok).

allows(Label, ApacheOid, client(Realm,ClientId,ClientLoc),
        Op, url(_Proto, _Loc, Path, Args), Response) :-
      object(ApacheOid, apache,_), attr(ApacheOid, filters, Filters),
      apacheFilterAllows(Filters,ClientId, ClientLoc, Path, Op)
      attr(ApacheOid,xforms, Xforms),
      apacheXform(Xforms, Path, Match),
      Match = nwService(NIp, NProto, NPort),
      nwDirectory(Oid, NIp, NProto, NPort), %network service directory
      allows(Label, Oid, client(Realm,ClientId, ClientLoc), Op,
              url(NProto, NIp ,Path, Args), Response)]).

apacheXform(Transforms, Path, Target)  :-
      true iff the transform maps specified Path to Target.

apacheFilterAllows(Filters, ClientId, ClientLoc, Path, Op) :-
      true iff filters allow Clientid logging in from Clientloc access
      to Path.
```

3.3.3 A crucial restriction

The crucial restriction we make is that all the “allows” terms in the body (the right side of a rule) appear positively – no “allows” term is negated. Our intuition behind this restriction is that information flows are monotonic – enabling a new information flow does not disable existing enabled flows.

When none of the “allows” is negated, the set of all allow rules are equivalent to a datalog program. An important consequence is that, given all attribute values for all entities, we can efficiently evaluate the conditions under which each “allows” predicate is true. These conditions are expressed as propositions containing only uninterpreted variables, and equality checks and extensional predicates over the uninterpreted variables.

From datalog theory (see [Imm]), we know that datalog programs (alternatively, transitive closures over monotonic relations) can be evaluated in time polynomial in the size of the program (number of rules and facts). In our case, if the number N of entities determines the size of the program (each model is of fixed size, independent of the size of the enterprise system). Each allows predicate is defined on $O(N^2)$ points. In each iteration we compute the value on at least one point, and each iteration takes $O(N)$ time, for an upper bound of $O(N^3)$ on the overall complexity of validation.

In practice, bottom-up evaluation of rules combined with tabling of intermediate results is an efficient way to evaluate transitive closures. We use the XSB Prolog engine which uses this strategy and is efficient in practice.

3.3.4 The "triggers" predicate

The triggers predicate is related to our earlier point that a deny policy is not a negation of the corresponding permit policy. The *triggers* predicate captures the notion that a request made by an entity can be triggered by another request that was made to that entity. For example, a write request to the file system made by a database server can be caused by a query request made to the database. We capture this notion as follows.

The predicate *triggers*(C_1,O_1,A_1, E_1, C_2, O_2,A_2 ,E_2) is true if and only if one request – O_1,A_1 from C_1 to E_1 – triggers a subsequent request – O_2,A_2 from E_1 (as client C_2) to E_2 .

In our example above, the entity E_1 is the database server, E_2 is the file system, and C_2 is the userid of the database server presented to the file system.

We make two special notes here. First, we note here that the allows predicate has a natural implicit definition of triggers as well. What we seek to model explicitly in "triggers" predicates are those phenomena that are *not* captured by allows predicates alone. For example, consider the policy *deny(employees,finance_server,write,_)*. As noted in the semantics of *deny* we need to ensure that the requisite operation cannot be invoked at all. However, the negation of a suitably defined allows predicate will not suffice. In our particular example, it may be possible to configure the system so that the write request from the client to finance_server is executed but succeeds only partially (we do not care about the particular mode of failure). If this is modeled, this is sufficient to infer that the corresponding allows predicate fails. However this is not adequate to infer that the write operation cannot be invoked. We use the triggers predicate to model this.

The second note is that a triggers predicate may be written even when no corresponding allows predicate exists. This is useful in modeling "secondary" behaviors that are exhibited by entities that are difficult to describe via the allows model. This is especially true when modeling software that show undocumented behaviors. For example, web servers may use a cgi-bin script to access databases. Because of the difficulty of accurately modeling such scripts, the modeler may model the script as simply querying the database. In addition the modeler may want to capture the fact that the script may be optionally accessing a filesystem for unknown reasons. Since the filesystem access is not required for the database querying service, this will not be captured in the allows predicate. We can capture such behaviors via a triggers predicate.

In order to make the following discussion clearer, we also define here the predicate *triggers**(C_1,O_1,A_1, E_1, C_n, O_n,A_n ,E_n) as the transitive closure of *triggers*; the intuition is that an operation invoked on E_1 can result in, via a series of intermediate invocations, the request O_n,A_n made by C_n to E_n.

As noted, the triggers predicate will be used to analyze deny policies. For example, suppose we wish to verify that there is no action that client C_1 can perform which will eventually lead to operation O_n,A_n being performed on entity E_n. This can be verified by checking that *triggers**(C_1,O_1,A_1, E_1, C_n, O_n,A_n ,E_n) is not true for any choice of O_1,A_1, E_1, C_n. In practice, we will first compute the closure, represented as a propositional formula ranging over the variables O_1,A_1,

E_1, C_n and verify that this formula cannot be satisfied for the given client, and any choice of requests from that client.

The following is an example of a triggers rule which states the conditions under which a request from apache to tomcat is triggered by a request from a client to apache.

```
triggers(client(machineID, IPaddr, credentials(user,password)),
         get, Url, apacheI, Me, TomcatI, jk2invoke, Url) :-
      attr(ApacheI, runAs, Me),
      isJSP(Url),
      attr(ApacheI, jspserver, TomcatI).
```

In general, the definition of triggers will include triggers predicates on the right hand side, but as with allows, the triggers on the right side appear in positive form. Therefore, the triggers predicate, and its closure, can be efficiently computed.

3.3.5 Flow labels

Consider the following sequence of events:

1. An employee, using a browser, requests a url from apache and supplies his user name and password.
2. Apache, configured to do a basic authentication check, requests the password_file file from the file system.
3. Having verified the client supplied password, apache requests the file corresponding to the url from the file system.

In this example, a client request triggered a read request by apache on the password file. From the discussion in the previous section, this would violate the policy *deny(employee, password_file, read)*! This is an unfortunate consequence of our definition. The read request for *password_file* from apache to the file system, followed by the file sent back to apache is conceptually distinct from the client request to apache, followed by apache's request to the file system for the corresponding file. But we have not distinguished between these two flows.

Fortunately, there is a simple way to resolve this problem. The trick is to explicitly model the notion of "flow" using a flow label in each predicate. Roughly speaking, flow labels capture the role on whose behalf a flow has been created. A flow created to serve information to a client is different from the flow created for the apache server to verify the client's credentials. For example, the model for the apache web server can distinguish between the two flows by using a distinct flow label for the request for the password file, and reusing the flow label on the incoming client request for the read request for the content file. In our example, this modification ensures that the client request no longer appears to trigger the read request for password, and the deny policy is no longer violated.

The choice of what flow label to assign is made at modeling time. We offer the following concrete guideline: In any allows predicate the default *outgoing* flow label is always the same as the *incoming* flow label. An outgoing allow should have a new flow label if it is the case that the allow is a "guard" condition and no information (other than the guard information) is passed back from this allows into the response part of the incoming allows. For example, the authentication check is a guard condition on the incoming request. Since no information (except whether the authentication succeeded or not) passes back to the requestor, the outgoing allows to read the password file can have a different flow label.

4. Policy Validation

With the terminology and semantics defined in the previous sections, we now present a high-level overview of the algorithm to validate access policies. We assume that we are given a datalog model for each entity type, and that for every entity instance, the input configuration data includes values for all attributes of the corresponding model.

Step 1. *Attribute substitution.* For every "allows" and "triggers" rule, replace all attribute names with the given configuration values.

Step 2. *Datalog reduction.* Reduce each rule for every entity instance to a boolean value or to a propositional formula over uninterpreted variables only.

One way to conceptualize this step is as follows. Each allows term unfolds as a tree, where each node corresponds to an allow rule, and its child nodes correspond to allows terms appearing in the body of the rule. As this tree is evaluated bottom-up, propositional formulas are combined at each node. These propositions are attribute-free, but may contain uninterpreted variables such as client information, or arguments of operations.

Step 3. *Policy validation.* Validate policies in order of decreasing priority as follows.

i. For each *permit* policy *permit(*E, C,O,A*)*, check every allows rule of the form *allows*(E, C, O, A, R) for all O, A. If at least one of these rules is true then the policy is upheld, else it is violated.

ii. For each deny policy *deny*(E, C,O,A), check if *triggers**(C,*,*, *, *, O,A ,E) is false or if *allows*(E,C,O,A,*) is false for all choices of wildcards (*). If so, the deny policy is upheld. Otherwise it is violated.

Step 4. *Verify least privilege holds.* For every permit policy *Permit(*C,E,O,A*)* the set of allow predicates that must be true to uphold the policy consists of the allow terms in the subtree formed in Step 2 when an allows rule is unfolded. In this step we must verify that any allows term that is true must support a permit policy, i.e., it must fall within one of these subtrees. If not, the default deny policy is violated.

5. Conclusions and Extensions

Based on the ideas presented in this abstract, we have implemented the ISM prototype system outlined in Section 2. The ISM system runs in a laboratory setting --- with a number of Linux workstations, each with multiple virtual machines installed and running Linux (Debian) and Windows. We configured instances of apache 2.0, tomcat 5.0 and MySql server 4.1 on these virtual machines. As mentioned earlier, we have developed and tested adapters and models for these components. In our experiments, the adapters fetch the configuration data and parse them to build the extensional relations in a central database. The analysis engine operates on the normalized relational data and the policies that are provided separately. We test the results of the analysis directly against operations on the live system. Thus far, our results, both in terms of correctness and performance, have been encouraging. The final version of this abstract will discuss our implementation and experimental results.

In addition to modeling a greater variety of software components, we are also studying requirements for a modeling framework that will ease and standardize the process of developing and testing models. Finally, we are also investigating the problem of efficiently restoring the enterprise to a policy compliant state, when possible, by generating a new set of component acls.

6. Related Work

There is a wide body of recent work that is related to this project from various aspects. We mention some of these below.

The notion of policy based management has been around for some time. The Ponder project [PON] allows policies to be complex quantified statements on the attributes and methods of objects. In particular, security policies are typically restrictions on access to individual methods on objects. Ponder however does not address composability or end-to-end reasoning on the objects under consideration. As a result there is no concept of high-level policy and policy-based management is seen as a means largely to specify constraints on classes of objects rather than constraints on the behavior of the system as a whole.

Filtering Postures [FPOS] proposes using logic-based specifications of the requirements and binary decision diagrams to validate whether network layer devices such as firewalls, router ACLs and static routes are configured consistently.

Firmato [FRM] was the first analysis engine to generate rules for multiple firewalls that comply with high-level policy. Firmato generates firewall configurations given policies and topology. The "Smart Firewalls" project [SFW] developed closed-loop, policy-based security management for the network layer to both check policy compliance and generate policy compliant acls. Modeling network layer access control devices is particularly simple – incoming packets either flow through without modification, or are blocked depending on the configured rules. Application models are more complex since they must capture message types and transformations.

The MulVal system [MVL] uses Datalog models to solve the problem of analyzing vulnerability reports to infer privilege escalations in a system. MulVal operates in two phases. In the first phase, a scanner consumes reports of vulnerabilities in common entities (hosts, operating systems, network devices, applications) in a standardized XML format and searches the given network to find any components that are affected by these reports. In the second phase, MulVal logically deduces the privilege escalations (e.g. normal user becoming "root") that are possible due to these vulnerabilities using Datalog descriptions of the relationships between the components in question. Although MulVal addresses a complementary problem using similar ideas, they do not propose a formal modeling methodology.

Attack Graphs [JAJ, SHY] also model the cascading of attacks through a network but they do not have formal modeling methods. Unlike MulVal they do not deal with phenomena such as privilege escalation. Attack graphs analyze the potential impact of vulnerabilities on an environment without the accompanying context of enterprise security policies.

Transitive Closure-based reasoning: [NetKuang] which is based on [Kuang] proposed to find transitive vulnerabilities on networked computer systems created by poor system configuration using a rule-based approach. They use a backward-chained, goal-based expert system search on rules describing access structure as well as vulnerabilities. While Kuang is limited to analyzing a single host, NetKuang can analyze a LAN: the analyzer accesses hosts on the LAN using a customized network search algorithm. These works do not suggest a rigorous and general methodology that goes beyond vulnerability exploitation on hosts

Operating system vulnerabilities modeling [OSV]: This paper deals with the problem of modeling the transitive effects of bugs in operating system routines such as buffer overflow. This work has the ability to deal with sophisticated attacks such as phased attacks but at the cost of stateful modeling that makes the complexity of the logic exponential. The paper does not address networked environments.

In the commercial arena, SolSoft's policy manager [SOL] allows users to specify high-level network layer policies that are used to automatically generate configurations for firewalls, routers and switches for any desired topology. TruSecure [TRU] offers an application layer security solution that claims to solve the problem of transitive access, but their solution uses best practices rather than formal methods.

7. References

[PON] N. Damianou, N. Dulay, E. Lupu, and M. Sloman. The ponder policy specification language. In Morris Sloman, editor, Proc. of Policy Worshop, 2001, Bristol UK, January 2001.

[FRM] Yair Bartal, Alain Mayer, Kobbi Nissan, and Avishai Wool. Firmato: A novel firewall management toolkit. In Proc. IEEE Computer Society Symposium on Security and Privacy, 1999.

[FPOS] J. D. Guttman. Filtering postures: Local enforcement for global policies. In Proc. IEEE Symp. on Security and Privacy, Oakland, CA, 1997.

[MVL] S. Govindavajhala and X. Ou. MulVal: Multiple Vulnerability Analyzer. To appear in Usenix Security Symposium, 2005.

[NetKuang] Zerkle, D., K. Levitt, "NetKuang—A Multi-Host Configuration Vulnerability Checker." *6th USENIX Security Symposium*. San Jose, California, July 22–25, 1996, pp. 195–204.

[Imm] N. Immerman. Descriptive Complexity, Springer 1999.

[JAJ] S. Noel and S. Jajodia. Managing attack graph complexity through visual hierarchical aggregation. Conference on Computer and Communication Security, 2004.

[Kuang**]** Robert W. Baldwin. Kuang: Rule based security checking. Documentation in ftp://ftp.cert.org/pub/tools/cops/cops.tar.

[SHY] O. Sheyner and J. Wing. Tools for Generating and Analyzing Attack Graphs. *Proceedings of Formal Methods for Components and Objects*, Lecture Notes in Computer Science, 2004, pp. 344-371.

[SFW] S. Bhatt, S. Rajagopalan, P. Rao. Automatic Management of Network Security Policy. MILCOM 2003.

[OSV] C.R. Ramakrishnan and R.C. Sekar. Model-Based Analysis of Configuration Vulnerabilities. Journal of Computer Security, v10, pp 189-209, 2002.

[SOL] Solsoft Policy Manager, SolSoft, Inc. www.solsoft.com

[TRU] Enterprise Policy Management. White Paper. TruSecure, Inc. www.trusecure.com

[XSB] P. Rao, K. F. Sagonas, T. Swift, D. S. Warren, and J. Freire. *XSB: A System for Efficiently Computing Well-Founded Semantics*. In J. Dix, U. Furbach, and A. Nerode, editors, Proceedings of the 4th International Conference on Logic Programming and Non-Monotonic Reasoning (LPNMR'97), number 1265 in Lecture Notes in AI (LNAI), pages 2--17, Dagstuhl, Germany, July 1997. Springer Verlag.