WEB SERVER AUTHORISATION WITH THE POLICYUPDATER ACCESS CONTROL SYSTEM

Vino Fernando Crescini
University of Western Sydney
Penrith South DC, NSW 1797, Australia
jcrescin@cit.uws.edu.au

Yan Zhang
University of Western Sydney
Penrith South DC, NSW 1797, Australia
yan@cit.uws.edu.au

ABSTRACT

The PolicyUpdater system is a generic access control system that provides policy evaluations and dynamic policy updates. These functions are achieved by the use of a logic-based language to represent access control policies. In this paper, we discuss the underlying details of the PolicyUpdater system as well as the issues arising from its application to a web server access control system. Integrating the PolicyUpdater system with a web server provides a more flexible and expressive means of representing authorisation policies.

KEYWORDS

security issues, web software, logic programming, authentication

1. INTRODUCTION

The simplest method of web access control is to grant resource access only to registered users that have been authenticated by the system using a username and password pair. While this method is very effective and easy to implement, its simplistic nature and inflexibility prevents it from being used in applications where access control is needed between authenticated users or even non-registered (unauthenticated) users.

Another common form of access control allows conditional policies to be defined. Such conditions include user authentication, client hostname, time of day, etc. Access control lists may be formed by grouping together rules, which are grant or deny actions associated with one or more conditions. While this method is provided as a standard core feature by most web servers, it is only capable of modeling simple access control policies.

Recent advances in the broader access control field have produced a number of different approaches to logic-based access control systems. Bertino, et. al. [4] proposed such a system based on ordered logic with ordered domains. Jajodia, et. al. [7] on the other hand, proposed a general access control framework that features handling of multiple policies. Another important work is the system proposed by Bai and Varadharajan [2, 3]. Their system's key characteristic is the ability to dynamically update the otherwise static policy base. These systems, effective as they are, lack the details necessary to address the issues involved in the implementation of such a system to be used in a web server access control application. The *Policy Description Language*, or *PDL*, developed by Lobo, et. al. [9], is a language for representing event and action oriented generic policies. *PDL* is later extended by Chomicki, et. al. [6] to include *policy monitors* which, in effect, are policy constraints. Bertino, et. al. [5], again took *PDL* a step further by extending *policy monitors* to allow users to express preferred constraints. While these generic languages are expressive enough to be used for web server access control systems, systems built for such languages will not have the ability to dynamically update the policies.

To overcome these limitations, we propose the general purpose PolicyUpdater access control system, which, with its own authorisation language, provides a formal logic-based representation of policies, with

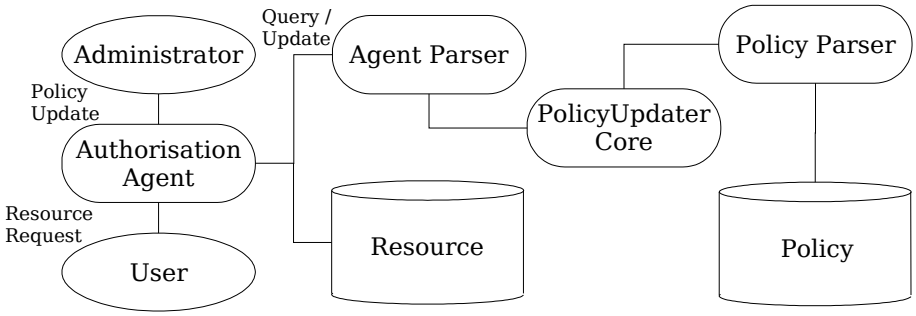
variable resolution and default propositions, a mechanism to conditionally and dynamically perform a sequence of policy updates, and a means of evaluating queries against the policy base.

In Section 2, the paper introduces the PolicyUpdater system, its internals, how it interacts with other components of access control systems and how access control policies are stored, evaluated and updated. The section also presents language L, a logic based language used by PolicyUpdater to represent theauthorisation policies. Section 3 gives an overview of the standard access control system built into existing web servers. More importantly, the section discusses how the PolicyUpdater system is used as anauthorisation module for the Apache web server. Finally, Section 4 has some concluding remarks that describe a possible extension and improvement for the system.

2. POLICYUPDATER SYSTEM

The PolicyUpdater system is a generic access control system designed to be used in a variety of applications. The key feature of the PolicyUpdater system is that policies are stored and evaluated as logic programs. By using this approach, the PolicyUpdater system can provide the means to allow the policies to by dynamically altered or updated in the policy base. Another feature of the PolicyUpdater system is that it allows policies to be defined with conditional constraints and default rules. Like the dynamic update facility, this feature is possible because of the PolicyUpdater system's logic representation of the policy base.

Figure1. Overview of the core PolicyUpdater System



As shown in Figure 1, the core PolicyUpdater system works with an external authorisation agent, which functions as an access control enforcer and at the same time provide an interface for administrators to allow policy updates to be performed. The policy parser is used by the system to read the policy definition file, while the agent parser is responsible for all interactions with an authorisation agent. At this stage, it is important to note that the enforcer agent and the two parsers, although part of the entire access control system, are separate from the core PolicyUpdater system.

2.1 Language L

Language L is a first-order logic language that represents a policy base for anauthorisation system. Two key features of the language are: (1) providing a means to conditionally and dynamically update the existing policy base and (2) having a mechanism by which queries may be evaluated from the updated policy base.

2.1.1 Identifiers

As the most basic unit of language L, identifiers are used to represent the different components of the language. There are three main classes of identifiers:

- *Entity Identifiers* represent constant entities that make up an atom. They are divided further into three types, with each type again divided into the *singular* and *group* entity categories:
  - *Subjects*. e.g. bob, administrators, users.
  - *Access Rights*. e.g. read, write, own.
  - *Objects*. e.g. file, directory, database.
- *Policy Update Identifiers* are used as name handles or labels for policy update definitions and directives. These identifiers are used to reference policy updates when applying them.e.g. grant_read, revoke_write.
- *Variable Identifiers* are used as place holders to represent defined entity identifiers.

2.1.2 Atoms, Facts and Expressions

An atom is composed of a relation with 2 to 3 entity or variable identifiers that represent a logical relationship between the entities. There are three types of atoms:

- *Holds*. Atoms of this type state that the subject identifier *sub* holds the access right identifier *acc* for the object identifier *obj*.

```
holds(<sub>, <acc>, <obj>)
```

- *Membership*. This type of atom states that the singular identifier *elt* is a member or element of the group identifier *grp*. It is important to note that identifiers *elt* and *grp* must be of the same base type (e.g. subject singular and subject group).

```
memb(<elt>, <grp>)
```

- *Subset.* The subset atom states that the group identifiers *grp1* and *grp2* are of the same types and that group *grp1* is a subset of the group *grp2*.

```
subst(<grp1>, <grp2>)
```

A fact makes a claim that the relationship represented by an atom or its negation holds in the current context. Facts are negated by the use of the negation operator "*!*" . The following shows the formal syntax of a fact:

```
[!]<holds_atom>|<memb_atom>|<subst_atom>
```

An expression is either a fact, or a logical conjunction of facts, separated by the double-ampersand characters " *&&*" .

```
<fact1> [&& <fact2> [&& ...]]
```

Atoms that contain no variables, i.e. composed entirely of entity identifiers, are called *ground atoms*. Similarly, facts and expressions that are made up of ground atoms are called *ground facts* and *ground expressions*, respectively.

2.1.3 Entity Identifier Definition

All entity identifiers (subjects, access rights, objects and groups) must first be declared before any other statements to define the entity domain of the policy base. The following entity declaration syntax illustrates how to define one or more entity identifiers of a particular type.

```
ident sub|acc|obj[-grp] <ent_id>[, ...]];
```

2.1.4 Initial Fact Definition

The initial facts of the policy base, those that hold before any policy updates are performed, are defined by using the following definition syntax:

```
initially <ground-exp>;
```

2.1.5 Constraint Definition

Constraints are logical rules that hold regardless of any changes that may occur when the policy base is updated. The constraint rules are true in the initial state and remains true after any policy update.

The constraint syntax below shows that for any state of the policy base, expression *exp1* holds if expression *exp2* is true and there is no evidence that *exp3* is true. The *with absence* clause allows constraints to behave like default propositions, where the absence of proof of an expression implies that the expression

holds. It is important to note that the expressions *exp1*, *exp2* and *exp3* are non-ground expressions, which means identifiers within them may be variables.

```
always <exp1> [implied by <exp2> [with absence <exp3>]];
```

2.1.6 Policy Update Definition

A policy update is defined by the following syntax:

```
<up_id>([<var_id>[, ...]]) causes <exp1> if <exp2>;
```

up_id is the policy update identifier to be used in referencing this policy update. The optional *var_id* list contains the variable identifiers occurring the expressions *exp1* and *exp2* and will eventually be replaced by entity identifiers when the update is referenced. The postcondition expression *exp1* is an expression that will hold in the state after this update is applied. The expression *exp2* is a precondition expression that must hold in the current state before this update is applied.

2.1.7 Policy Update Directives

The policy update sequence list contains a list of references to defined policy updates in the domain. The policy updates in the sequence list are applied to the current state of the policy base one at a time to produce a policy base state upon which queries can be evaluated. The following four directives are the policy sequence manipulation features of language L:

- *Adding an update into the sequence*. Defined policy updates are added into the sequence list through the use of the following directive statement:

  ```
  seq add <up_id>([<ent_id>[, ...]]);
  ```

  where *up_id* is the identifier of a defined policy update and the *ent_id* list is a comma-separated list of entity identifiers that will replace the variable identifiers that occur in the definition of the policy update.
- *Listing the updates in the sequence*. The following directive may be used to list the current contents of the policy update sequence list.

  ```
  seq list;
  ```

  This directive is answered with an ordinal list of policy updates in the form:

  ```
  <n> <up_id>([ent_id[, ...]])
  ```

  where *n* is the ordinal index of the policy update within the sequence list starting at 0. *up_id* is the policy update identifier and the *ent_id* list is the list of entity identifiers used to replace the variable identifier place-holders.
- *Removing an update from the sequence*. The syntax below shows the directive to remove a policy update reference from the list. *n* is the ordinal index of the policy update to be removed. Note that removing a policy update reference from the sequence list may change the ordinal index of other update references.

  ```
  seq del <n>;
  ```

- *Computing an update sequence*. The policy updates in the sequence list does not get applied until the *compute* directive is issued. The directive causes the policy update references in the sequence list to be applied one at a time in the same order that they appear in the list. The directive also causes the system to generate the policy base models against which query requests can be evaluated.

  ```
  compute;
  ```

2.1.8 Query Directives

A ground query expression may be issued against the current state of the policy base. This current state is derived after all the updates in the update sequence have been applied, one at a time, upon the initial state. Query expressions are answered with a *true*, *false* or an *unknown*, depending on whether the queried expression holds, its negation holds, or neither, respectively. Syntax is as follows:

```
query <ground-exp>;
```

2.2 Query Evaluation and Policy Updates

The PolicyUpdater system evaluates queries by translating language L policy base into a *normal logic program*, which can be evaluated using the *stable model semantics* [12]. With the use of the *smodels* library to generate a set of all the stable models S of the policy, PolicyUpdater can then perform query evaluations by checking to see if the query expression holds in set S.

Without policy updates, the translation of language L into a normal logic program is a straightforward procedure: simply express each statement as a logical rule of the form:

$$Q \text{ <- } P_0, P_1, ..., P_n$$

where P_i ($0 <= i <= n$) are the premise facts and Q is the consequent fact.

However, with policy updates, the policy base has states: a state transition occurs when a policy update is applied. Since atoms cannot express the states in which its relation is meaningful, facts cannot express the states in which they hold or do not hold. Such policy bases cannot be translated into normal logic programs as rules in one state may not necessarily apply to another state, and there is no means to distinguish between two states.

$$S' = update(S, U)$$

where S is the current state, S' is the state after update U is applied and the function *update* applies update U to state S. By using this technique, we can now translate each statement of language L to a normal logic program:

- Initial state facts are assigned the extra parameter S_0.
- Constraints are rules that apply regardless of any updates, so for each state S_i ($0 <= i <= n$, n is the total number of applied updates), each fact in each constraint is assigned the new parameter S_i.
- Policy Updates are applied by translating them into logic rules: preconditions are premises and postconditions are consequents. Note that facts in the premises are given the parameter S_b and facts in the consequents are given the parameter S_a (S_b is the current state before update U is applied and $S_a = update(S_b, U)$).

2.3 An Example

The following language L program code listing shows a simple rule-based document access control system scenario. In this example, the subject *alice* is initially a member of the subject group *grp3*, which is a subset of group *grp1*. The group *grp1* also initially holds a *read* access right for the object *file*. The constraint states that if the group *grp1* has *read* access for *file*, and no other information is present to conclude that *grp1* do not have *write* access for *file*, then the group *grp1* is granted *write* access for *file*.

```
ident sub alice;
ident sub-grp grp1, grp2;
ident acc read, write;
ident obj file;

initially
  memb(alice, grp2) && holds(grp1, read, file) && subst(grp2, grp1);
```

```
always holds(grp1, write, file)
  implied by holds(grp1, read, file)
  with absence !holds(grp1, write, file);

delete_read(SG0, OS0) causes !holds(SG0, read, OS0);

seq add delete_read(grp1, file);

compute;

query holds(grp1, write, file);
query holds(alice, read, file);
```

The result of the queries above is shown below:

- holds(grp1, write, file) = true
- holds(alice, read, file) = false

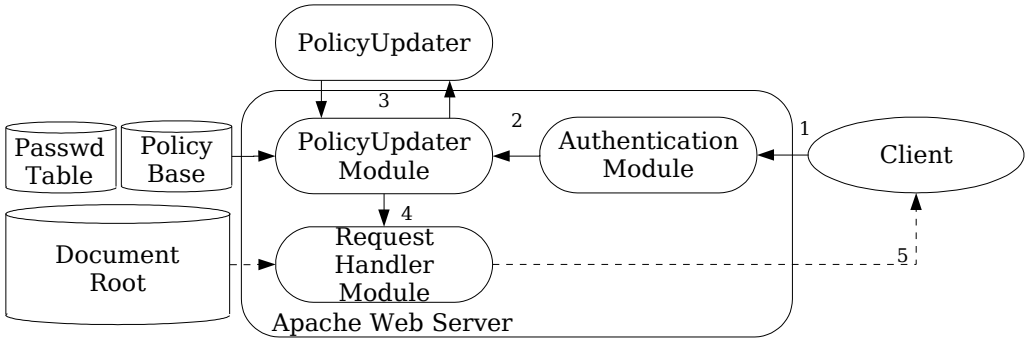
3. WEB SERVER ACCESS CONTROL

Most web servers include some form of built-in access control mechanism. The most popular one, the Apache web server, provides a rudimentary access control system as provided by its *mod_auth and mod_access* modules [1,8]. With this built-in access control system, Apache provides the standard HTTP *Basic* and *Digest* authentication schemes [11], as well as an authorisation system to enforce access control policies. Such policies may be defined as per-request rules with HTTP request methods (*GET*, *POST*, etc.) [10] as access rights; users and hosts as subjects and the resources in the server's document root as objects.

The Apache access control mechanism is best illustrated by a HTTP GET request. This mechanism may be summarized by four simplified steps: (1) a client (user) makes a request for a file; (2) if required by the authorisation policy, the client is authenticated against a password table; once authenticated, the client request is sent to Apache's access control module (3) where the policy is checked to decide whether to accept or deny the request; if the request is accepted, it is then passed on to other request handlers for processing and eventually, (4) the transaction is completed by sending the requested file back to the client.

3.1 Integration of PolicyUpdater

Figure2. Apache Access Control Mechanism with PolicyUpdater



As shown in Figure 2, Apache's Access Control module, together with its policy base, is replaced by the PolicyUpdater module and its own policy base. The sole purpose of the PolicyUpdater module is to act as an interface between the web server and the core PolicyUpdater system. The system works as follows: as the server is started, the PolicyUpdater module initialises the core PolicyUpdater system by sending the policy base. When a client makes an arbitrary HTTP request for a resource from the server (1), the client (user) is authenticated against the password table by the built-in authentication module; once the client is properly

authenticated (2) the request is transferred to the PolicyUpdater module, which in turn generates a language L query (3) from the request details, then sends the query to the core PolicyUpdater system for evaluation; if the query is successful and access control is granted, the original request is sent to the other request handlers of the web server (4) where the request is eventually honoured; then finally (5), the resource (or acknowledgment for HTTP requests other than GET) is sent back to the client. Optionally, client can be an administrator who, after being authenticated, is presented with a special administrator interface by the module to allow the policy base to be updated.

3.1.1 Policy Description in Language L'

The policy description in the policy base is written in language L', which is syntactically and semantically similar to language L except for the lack of entity identifier definitions. Entity identifiers need not be explicitly defined in the policy definition:

- *Subjects* of the access control policies are the users. Since all users must first be authenticated, the password table used in authentication may also be used to extract the list of subjects.
- *Access Rights* of the policies are built in: they are the HTTP request methods as defined by the HTTP 1.1 standard [10] (i.e. OPTIONS, GET, HEAD, POST, PUT, DELETE, TRACE and CONNECT).
- *Objects* are the resources available in the server themselves. Assuming that the document root is a hierarchy of directories files, each of these are mapped as a unique object of language L'.

3.1.2 Mapping the Policies to Language L

As mentioned above, one task of the PolicyUpdater module is to generate a language L policy from the given language L' to be evaluated by the core PolicyUpdater system. This process is outlined below:

- *Generating entity identifier definitions*. As mentioned in the previous section, subjects are extracted from the authentication (password) table; access rights are hard-coded built-ins; and the list of objects are generated by traversing the document root for files and directories.
- *Generating additional constraints*. The module also generates additional constraints to preserve the relationship between groups and elements. This is useful to model the assertion that unless explicitly stated, users holding particular access rights to a directory automatically hold those access rights to every file in that directory (recursively, if with subdirectories). The PolicyUpdater module achieves this assertion by generating non-conditional constraint rules that state that each file (object) is a member of the directory (object group) in which it is contained. All other language L' statements (initial state definitions, constraint definitions and policy update definitions) are already in language L form.

3.1.3 Evaluation of HTTP Requests

A HTTP request may be represented as a simplified tuple:

<*user*, *request_method*, *requested_resource*>

user is the authenticated username that made the request (subject); *request_method* is any of the standard HTTP request methods (access right); and *requested_resource* is the resource associated with the request (object). Intuitively, such a tuple may be expressed as a language L atom:

```
holds(user, request_method, requested_resource)
```

With each request expressed as language L atoms, a language L query statement can be composed to check if the request is to be honoured:

```
query holds(user, request_method, requested_resource);
```

Once the query statement is composed, it is then sent by the PolicyUpdater module to the core PolicyUpdater system for evaluation against the policy base.

3.1.4 Policy Updates by Administrators

After being properly authenticated, an administrator can perform policy updates through the use of a special interface generated by the PolicyUpdater module. This interface lists all the predefined policy

updates that are allowed, as defined in the policy description in language L', as well as all the policy updates that have been previously applied and are in effect. As with the core PolicyUpdater system, administrators are allowed only the following operations:

- Apply a policy update or a sequence of policy updates to the policy base. Note that like language L, in language L', policy updates are predefined within the policy base themselves.
- Revert to a previous state of the policy base by removing a previously applied policy update from the policy base.

4. CONCLUSION

In this paper, we have presented a logic based approach for access control in web servers. As we have shown, the core PolicyUpdater used in conjunction with an interface module that plugs into a web server provides a flexible policy base framework that provides dynamic and conditional updates and access control query request evaluations.

One possible future extension to this work is the integration of temporal logic to language L (and therefore language L') to allow time properties to be expressed in access control policies. Such extension will be useful in access control systems such as e-commerce applications where authorisations are granted or denied based on policies that are time dependent.

REFERENCES

[1] Apache Software Foundation,2004. Authentication, Authorization and Access Control. *Apache HTTP Server 2.1 Documentation*,http://httpd.apache.org/docs-2.1/.

[2] Bai, Y., Varadharajan, V., 1999. On Formal Languages for Sequences of Authorization Transformations. In *Proceedings of Safety, Reliability and Security of Computer Systems*. Also in *Lecture Notes in Computer Science*, Vol. 1698, pp. 375-384. Springer-Verlag.

[3] Bai, Y., Varadharajan, V., 2003. On Transformation of Authorization Policies. In *Data and Knowledge Engineering*, Vol. 45, No. 3, pp. 333-357.

[4] Bertino, E., Buccafurri, F., et. al., 2000. A Logic-based Approach for Enforcing Access Control. In *Journal of Computer Security*, Vol. 8, No. 2-3, pp. 109-140, IOS Press.

[5] Bertino, E., Mileo A., et. al., 2003. Policy Monitoring with User-Preferences in PDL. In *Proceedings of IJCAI-03 Workshop for Nonmonotonic Reasoning, Action and Change*, pp. 37-44.

[6] Chomicki, J., Lobo, J., et. al., 2000. A Logic Programming Approach to Conflict Resolution in Policy Management. In *Proceedings of KR2000, 7th International Conference on Principles of Knowledge Representation and Reasoning*, pp. 121-132, Kaufmann.

[7] Jajodia, S., Samarati, P., et. al., 2001. Flexible Support for Multiple Access Control Policies. In *ACM Transactions on Database Systems*, Vol. 29, No. 2, pp. 214-260, ACM.

[8] Laurie, B., Laurie, P., 2003. *Apache: The Definitive Guide* (3rd Edition). O'Reilly & Associates Inc.

[9] Lobo J., Bhatia R., et. al., 1999. A Policy Description Language. In *Proceedings of AAAI 16th National Conference on Artificial Intelligence and 11th Conference on Innovative Applications of Artificial Intelligence*, pp. 291-298, AAAI Press.

[10] Network Working Group, 1999. *Hypertext Transfer Protocol -- HTTP/1.1(RFC 2616)*. The Internet Society, ftp://ftp.isi.edu/in-notes/rfc2616.txt.

[11] Network Working Group, 1999. *HTTP Authentication (RFC 2617)*. The Internet Society, ftp://ftp.isi.edu/in-notes/rfc2617.txt.

[12] Simons, P., 1995. Efficient Implementation of the Stable Model Semantics for Normal Logic Programs. In *Research Report A35*, Helsinki University of Technology, Digital Systems Laboratory, Finland.