CERIAS Tech Report 2001-17

Authentication-driven Authorization on Web Access

Yuhui Zhong, Bharat Bhargava
Center for Education and Research in
Information Assurance and Security
&
Department of Computer Sciences, Purdue University
West Lafayette, IN 47907

Authentication-driven Authorization on Web Access*

Yuhui Zhong Bharat Bhargava
Center for Education and Research in Information Assurance and Security
and
Department of Computer Science
Purdue University
West Lafayette, IN, U.S.A.
CERIAS TR 2001-17

Abstract *Unlike in traditional computing environments such as operating systems or databases, the authorized user set, the mode of access, users' access rights, etc., are not predefined in WWW. In order to assign privileges to authorized but not predefined users in dynamic access environments, we propose an approach called authentication-driven authorization. In this approach, authentication is integrated with authorization. The validity of a user is checked by using authentication routines associated with the requested data object. The access permission is achieved by authentication rather than by inheriting from group/role membership relation. A logic-based authorization language has been designed. A prototype has been implemented, which can be used to enforce complex web-based application security policies.*

Keywords dynamic access environments, predicate, security rule

1 Introduction

The Internet is a cost-effective and ubiquitous vehicle for connecting business enterprises. With the growing use of the World Wide Web, security of the web-based systems is now an important business decision. Our research focuses on preventing the information access by unauthorized users and making the information secured and reliable for authorized users.

Current procedures for authorization on the web are classified as:

- *Domain based restriction*: The web allows access to resources that are restricted by domain. All access attempts that do not originate from a user or a website identified with a specific domain will be denied.
- *User based restriction*: The web maintains a list of authorized users and their passwords. Users can log in and access material from any computer on the Internet as long as they know a valid user name and the password. For systems that require a stronger authentication, username / password can be replaced by public key certificate. The login process is based on an authentication protocol such as Secure Socket Layer (SSL).

A major disadvantage of these schemes is that either the system has to maintain a predefined set of user names or a predefined set of domain names or both [4]. Therefore, these systems restrict information sharing over the web and reduce the possibility of doing business among parties who do not know each other, which in turn restricts the proliferation of the use of the web. Our research is directed to deal with the limitation of the current authorization approaches.

We envisioned that there exist two kinds of access environments on the web.

*This research is supported by CERIAS and NSF grants CCR-9901712 and CCR-0001788. This paper will appear in *Proc of International conference on Internet Computing* (*IC'2001*), Las Vegas, June 2001

- *Static Access Environment*: In this kind of environment, the contents of the web as well as the user requirements change rarely. A set of users and their access rights can be predefined. Intranet is an example of such access environments.
- *Dynamic Access Environment*: In this kind of environment, the contents of the web and the user requirements change frequently. Neither the user set nor their access right is predefined.

Our research focuses on the authorization in dynamic access environments. We investigate three security approaches for dynamic access environment.

- *Mobile Agent Approach*: Under this approach, users are asked to enter appropriate information that is used to authenticate them. The system performs various checks on this information using a mobile agent that decides the access restrictions to be imposed on the user. Details can be found in the technical report [11].
- *Integrate Data Mining Technology into Web Security*: Data mining technology has proved to be an efficient tool to discover interesting and useful patterns in a large amount of data such as web documents. We investigate the idea of integrating data mining techniques into web security. We aim at finding patterns, which are different from usual access pattern. We discover change in a pattern by comparing some traces of new patterns from the user access logs. Details can be found in the paper [12].
- *Authentication-Driven Authorization Approach*: Under this approach, authentication is integrated with authorization.

We present the authentication-driven authorization approach in this paper.

1.1 Research Considerations

The considerations for web authorization research are as following:

- Several access control models like discretionary access control model, mandatory access control model, and role-based access control model have been proposed [5,6]. These models are developed for static access environments such as operating system, database system and corporate intranet, where the sets of authorized user, the mode of access etc. are predefined. These access control models need be modified for dynamical access environments.
- The authorization language (i.e. the language for specifying security policies) should be expressive and flexible. The protection requirements of data objects on web vary from "no protection" to "highly sensitive". All these authorizations should be expressed using the authorization language [1, 3, 8]. The semantics of the authorization system should be extensible to satisfy the requirements of web applications [2].

1.2 Our Approach

We support authorization in a dynamic access environment by integrating authentication into authorization. In traditional settings, authorization is divided into authentication and access control. Authentication verifies that the identity of a user is valid. Access control determines whether the requester is authorized to perform the action or not. The access permission is achieved by inheriting from group/role membership relation. This view of authorization is not suitable for dynamic access environments where a valid user of a data object may not be a predefined user. The problem is in assigning privileges to strangers.

We solve this problem by authentication. A user is verified by using a set of routines associated with the data object requested by the user. These routines are called authentication routines. The access permission is granted if the user passes the authentication. The goal of authentication is not to prove the identity of the user (the identity of a user may be unavailable or meaningless to the web [3]), but to prove that the requestor satisfies the security constraints of the requested data object. Since the authorization decisions are made based on the results of authentication, we call this approach authentication-driven authorization. The following example illustrates the idea. Suppose the person who can access web page "Alice's wedding" is only required to know where the wedding ceremony took place. Carol becomes a legitimate user of the web page by proving that she has this knowledge. She does not have to provide any other informa-

tion (e.g. username/password, social security number, occupation, public key certification etc.).

Because of the use of authentication routines, we cannot use the existing authorization languages. A logic-based authorization language has been designed and presented in Section 3. This language has a special type of predicates, user-defined predicates, which makes the language expressive and extensible. A prototype of an authorization system based on this authorization language has been implemented.

The remainder of this paper is organized as following: Section 2 introduces related research. In section 3, our logic infrastructure is described. The architecture of the prototype system is presented in section 4. Section 5 discusses the implementation issues.

2 Related Work

The research of Professor S. Jajodia influences us in the design aspects of the authorization language. Most existing authorization systems are designed with one specific access control policy. The drawback of such systems is that it is difficult to capture all protection requirements that a user may enforce using a single policy [1]. S. Jajodia, P. Samarati and V.S. Subrahmanian proposed a logic-based authorization language whose expressiveness is strong [1,8]. However, this language is designed for static access environments. The access restrictions are enforced through static group/role membership relations.

A certificate-based approach to assign roles to strangers is proposed in [2]. This approach solves the problem of assigning privileges to strangers based on public key certificate. Our approach does not require web users to use certificates.

3 Logic Infrastructure of the Proposed System

We present the authorization language in this section.

3.1 Objects

The object set *O* is the data set to which authorization can be granted. The most important objects for access requests are web pages. In order to facilitate the system administration, the web pages are organized in a multilevel hierarchy. In addition to specifying protection constraints for each web page individually, we can specify the security rule for a set of web pages. The hierarchy reflects both physical and logical views. The hierarchy of the directories and web pages in the file system constitutes a physical view. The web pages can form the logical views. Logical views are defined based on application requirements. Similar to the definition of user group in traditional authorization systems, we define an object group as a set of web pages or other groups. Since user group is not defined in our authorization system (it can be implemented using user-defined predicate), we simply call "object group" as "group". Like directories, the nodes corresponding to groups in a hierarchical graph are internal nodes. There is a unique name associated with each view (logical or physical). Since the authorization can be granted to directories and groups, the object set *O* includes directories *D* and groups *G* in addition to web pages *P*.

3.2 Subjects

A subject is a user or a program on behalf of a user that requests a specific operation (e.g. read) on a web page.

3.3 Operations

We consider the following operations: *read, write, create, delete*.

3.4 Predicates

Predicates are major components of the logic infrastructure. Unlike the logic system proposed in [1,8], the set of predicates is not fixed in the system. We introduce the notion of user-defined predicate that allows system administrators to dynamically extend the authorization system. A user-defined predicate can have any semantic. User-defined predicates are used to achieve two goals: implementing authentication-driven authorization by incorporating authentication routines into the authorization system and enhancing the expressiveness of the authorization model. In the following predicates, the first four predicates are fixed by system. The fifth is the set of predicates defined by users.

- *ismember*, a ternary predicate whose first two arguments are two objects o_1, o_2, and third ar-

gument is the name of a view *v*. It captures the membership relation on a view between objects. This predicate is used in a security rule to specify the objects to which the rule is applicable.

- *grant*, a ternary predicate. The first argument of *grant* is an object term, the second one is a subject term, and the third is an operation. This predicate is used by system administrators to explicitly allow accesses to objects.
- *deny*, a ternary predicate, with the same arguments as *grant*. This predicate is used by system administrators to explicitly deny accesses to objects.
- *decision*, a ternary predicate, with the same arguments as *grant*. The predicate represents the authorization decision made by the authorization system. A request of operation *a* on an object *o* from a subject *s* is authorized if and only if we can infer that *decision*(*o, s, a*) is true from the security rules.
- P_1 is the set of the user-defined predicates. A user-defined predicate has the same arguments as *grant*. In contrary to the predicates defined above, the semantic of a user-defined predicate is interpreted by routines outside the authorization system. The expressiveness of our authorization system stems from this set of predicates. In Section 5, we will discuss how to implement user-defined predicates.

3.5 Security Rule

There are two kinds of security rules in our logic infrastructure. They are authorization rules and meta rules.

3.5.1 Authorization rule

Authorization rules are specified by system administrators to explicitly allow or deny accesses on objects.

(1) $grant(o,s,a) \leftarrow l_1 \& l_2 \ldots \& l_n$

(2) $deny(o,s,a) \leftarrow l_1 \& l_2 \ldots \& l_n$

For $1 \leq i \leq n$, l_i is a literal that is either a predicate belonging to the set of $\{ismember\} \cup P_1$ or its negation. And *o,s,a* are an object term, a subject term and an operation respectively.

The first kind of rules explicitly allows accesses to objects. The second kind of rules explicitly denies accesses to objects. The literals in the right hand side are used to specify conditions that must be satisfied. There are two types of conditions. One kind of conditions specifies the objects to which the authorization rules are applied. This kind of conditions is described using predicate *ismember*. The other kind of conditions is specified by user-defined predicates in the set of P_1. These predicates verify that a subject is the valid user of the requested object and all provisional actions are executed. The notion of provisional actions was first proposed in [2]. A set of provisional actions is defined in this paper. Provisional actions are security operations extending semantics of the authorization policies. For example, before a valid user is allowed to read sensitive information, he may be required to sign a term. Unlike the authorization system in [2], which defines a fixed set of provisional actions as primitives in the authorization system, we implement the semantics of provisional actions via user-defined predicates. It is more flexible to adding new provisional actions in our system than in the system proposed in [2].

Example:

The following authorization rule states the access restriction on web page 'wedding' (mentioned in Section 1). *Loc* is a user-defined predicate that checks whether a user knows the wedding place.

grant(u, wedding, read) ← *Loc(u, wedding, read)*

The following authorization rule illustrates how to include provisional actions in an authorization rule. *validuser* is a user-defined predicate that verifies that the user is entitled to read the data object 'code'. The user-defined predicate *sign* asks a valid user to sign a term before granting the access to him.

grant(*u, code, read*) ← *validuser(u, code, read)* & *sign*(*u, code, read*)

3.5.2 Meta Rule

Meta rules are used to specify which kind of policies (open policy or closed policy) should be enforced on data objects. In addition, conflict resolution policies and default policies are specified by meta rules.

- *Open/closed policy*: If a closed policy is enforced, a user can access an object if there exists an authorization rule stating that the user can access the object. If an open policy is enforced, a user can access an object if there

does not exist an authorization rule stating that the user cannot access the object. Open policy is very convenient for web authorization. For example, we can naturally specify the constraint 'those who have no social security id cannot read this page' by open policy. System administrators determine whether an open policy or a close policy should be enforced on data objects using a 3-tuple <*obj*, *operation*, *policy*>. System administrators describe the set of object to which the open/closed policy is applied by *obj*. *obj* is stated using predicate *ismember*. *policy* is either *close* or *open*. The open/closed policies are stored as following two rules in the system.

decision(o,s,a) ← *grant(o,s,a) (closed policy)*
decision(o,s,a) ← ¬ *deny(o,s,a) (open policy)*

- *Default policies*: A default policy is a 3-tuple <*obj, operation, permission*>. *obj and operation* specify the set of objects and the operation as in open/closed policies. *permission* is either *grant* or *deny*. When there is no authorization rule for a request on an object, we refer to the default policy of the object. Authorization decision is made based on *permission* field of the corresponding default policy. In case there exists either no default policy or contradictory default policies, we choose the conservative strategy and deny the request.
- *Conflict resolution*: The existence of the *deny* rules incurs potential conflicts (opposite decisions are made according to different rules). We use conflict resolution policies to solve this problem. A conflict resolution policy is a 3-tuple <*obj, operation, resolution*>. *obj and operation* specify the set of objects and the operation as in open/closed policies. The value of *resolution* field can be *denial-take-precedence*, *permission-take-precedence*, or *default*.

For the rules in the logic system proposed in [1,8], the expressiveness of the complex rule set is strong. However, it introduces overheads when system is enforcing rules. It is difficult for system administrators to write a complicate security rule. A complex rule is prone to errors. Based on these considerations, we choose a simple rule set. Specially, we separate the authorization rules and meta rules. Authorization rules, which are relatively straightforward, are specified in a logic form. Meta rules are difficult to be expressed by using first-order logic so that they are defined as 3-tuples. The complex security rules can be described with the aid of user-defined predicates in our infrastructure.

4 System Architecture

Figure 1 shows the architecture of the authorization system. The system consists of *Predicate Repository*, *Security Rule database*, *Rule Manager*, *Admin Tool*, and *Execution Environment*.

4.1 Predicate Repository & Security Rule database

Predicate Repository stores the information of user-defined predicates including the names and the registration entries of the corresponding authentication routines. *Security Rule database* stores the authorization rules and meta rules.

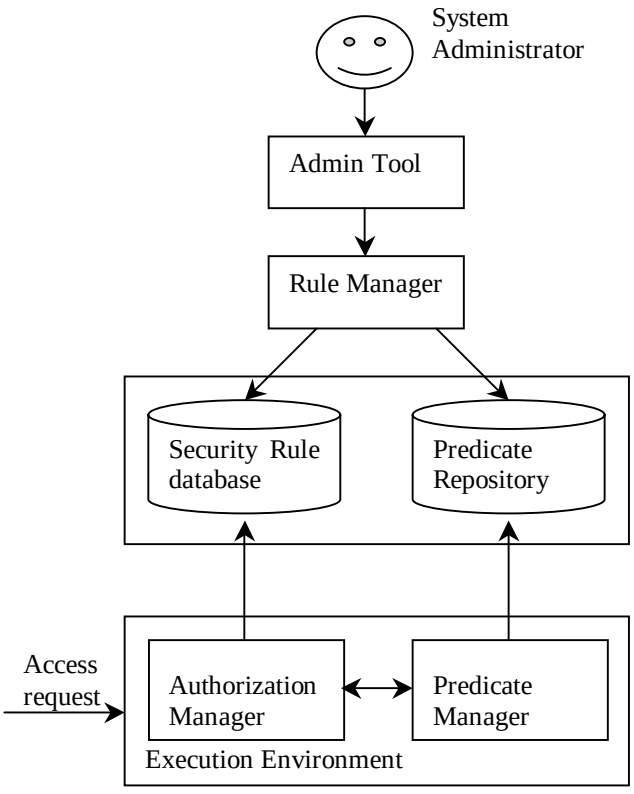

Figure 1 System Architecture

4.2 Admin Tool & Rule Manager

Admin Tool is the GUI for system administrators to register authentication routines, state user-define predicates and specify security rules. *Admin Tool* interacts with *Routine Repository* and *Security Rule database via Rule Manager*. The functionality of *Rule Manager* includes checking the authentication routines provided by system administrators, creating registration entries for these routines, transforming user-defined predicates and security rules into internal representations, and maintaining internal data structures to speedup the processes of making authorization decisions.

4.3 Execution Environment

Execution Environment is composed of *Predicate Manager* and *Authorization Manager*. Authorization decisions are made by *Authorization Manager*. Upon receiving an access request, *Authorization Manager* first searches the authorization rules associated with the object. If such an authorization rule does not exist, *Authorization Manager* makes the decision based on default policies associated with the object. In case there is no default policy, *Authorization Manager* denies the access request. If authorization rules exist, *Authorization Manager* infers the authorization decision based on authorization rules. When encountering a user-defined predicate, it sends an evaluation request to *Predicate Manager*. When receiving the request of evaluating a user-defined predicate from *Authorization Manager*, *Predicate Manager* retrieves the registration entry of the corresponding authentication routine in *Predicate Repository* using the name of the predicate. It prepares arguments for the routine, invokes it, and returns the result (true or false) to *Authorization Manager*. System administrators are allowed to specify timeout for each authentication routine to prevent a routine from consuming system resources for a long time. In case of timeout, *Predicate Manager* interrupts the execution of the authentication routine and returns false. If all the predicates on the right hand side of an authorization rule are true, the predicate on the left hand side (i.e. *grant* or *deny*) is true. The A*uthorization Manager* infers the result of *decision* based on the value of *grant/deny* and the open/closed policies. If several authorization rules exist, it is possible that contradictory results of *decision* are inferred based on different rules. In this situation, the conflict resolution policy is referred. If the conflict resolution policy is *denial-take-precedence*, the result of decision is determined as false. If the system administrator specifies *permission-take-precedence* as conflict resolution policy, the result of *decision* is true. The access request is granted if and only if *decision* is true.

5 System Implementation

We have implemented a prototype of the authorization system using Java. *Admin Tool* and *Rule Manager* are implemented as standalone applications. Server-side Java technologies are used to implement *Execution Environment*. Particularly, the *Execution Environment* is implemented as a Java Servlet [9].

- *Integrating User-define Predicates*: We defined a new interface *PredicateInterface* to unify user-defined predicates. This interface defines a method *evaluation*. The method *evaluation* is used to evaluate the predicate. It returns true if and only if the predicate is true. Three classes *ReqSubject*, *ReqObject*, and *ReqOperation* are defined to represent the subject, object and operation respectively. The parameters of method *evaluation* are instances of these three classes. In current implementation, an authentication routine must be a Java class that implements the *PredicateInterface* interface. When a user-defined predicate is registered, the predicate name and the class name of the corresponding routine are stored in *Predicate Repository*.

```
public interface PredicateInterface{
    boolean evaluation(ReqSubject sub,
                       ReqObject obj,
                       ReqOperation oper);
    …
}
```

- *Evaluating User-defined Predicates*: The Predicate Manager is responsible for invoking authentication routines of user-defined predicates. Since user-defined predicates are dynamically defined in the system, we cannot hardcode the predicates set in the implementation. We need a mechanism to create an instance whose class name is unknown until runtime. We solve this problem by using the

Reflection APIs of Java [10]. Upon receiving the request of evaluating a user-defined predicate, the *Predicate Manager* retrieves the class name of the authentication routine using the predicate name. It creates an instance using the class name and invokes its *evaluation* method to evaluate the predicate.

- *Integrating the prototype with web servers:* The prototype is integrated with a web server via a Servlet program in the *Execution Environment*. When a browser requests a page, the web server runs the Servlet program. The Servlet receives an instance of HttpServletRequest and creates instances of *ReqSubject, ReqObject,* and *ReqOperation*. The Servlet sends the instances to *Authorization Manager* that makes the authorization decision. After receiving the decision that access is granted by *Authorization Manager*, the Servlet sends the requested web pages to the web server.

6 Conclusions and Future works

We present an authorization approach and the authorization language for dynamic access environments. The architecture and the implementation of the prototype are discussed. The prototype has been implemented using Java. It is platform independent and can be integrated with web servers to enforce complex application polices.

Currently, we focus on making decisions locally and avoid using authorization delegation. We plan to incorporate trust delegation in the future. A prototype of user-modeling mobile agent has been developed in West Michigan University [11]. It will be integrated into our system. We plan to apply the prototype as a middleware between a web server and an information system to evaluate the proposed scheme in terms of security capabilities.

Reference

[1] S. Jajodia, P. Samarati, V.S. Subramanian, and E. Bertino. A Logical Language for Expressing Authorizations. In *Proc of the 1997 IEEE Symposium on Security and Privacy*, 1997

[2] M. Kudo and S. Hada. XML Document Security based on Provisional Authorization. In *Proc of ACM Conference on Computer and Communications Security*, 2000

[3] N. Li, B. N. Grosof, and J. Feigenbaum. A Logic-based Knowledge Representation for Authorization with Delegation. In *Proc of the 12th Computer Security Foundations Workshop*, 1999

[4] M. Mohania, V. Kumar, Y. Kambayashi, and B. Bhargava. Secured Web Access. In *International Conference on Digital Libraries: Research and Practice*, Nov. 2000

[5] J. Joshi, W. Aref, A. Ghafoor, and E. Spafford. Security Models for Web-Based Applications. *Communications of the ACM*, Feb. 2001

[6] G. Ahn and R. Sandhu. The RSL99 Language for Role-Based Separation of Duty Constraints. In *Proc of the fourth ACM workshop on Role-based Access Control*, 1999

[7] A. Herzberg, Y. Mass, and J. Mihaeli. Access Control Meets Public Key Infrastructure, Or: Assigning Roles to Strangers. In *Proc of the 2000 IEEE Symposium on Security and Privacy*, 2000

[8] S. Jajodia, P. Samarati, V. Subramanian, and E. Bertino. A Unified Framework for Enforcing Multiple Access Control Policies. In *Proc of the 1997 ACM international SIGMOD Conference on Management of Data*, May 1997.

[9] Java Servlet API, http://java.sun.com/products/servlet/

[10] Java Reflection API http://java.sun.com/j2se/1.3/docs/guide/reflection

[11] W. Tu, S. Tang and M. Mohania. Web Databases and Agent Technology. In *Technical Report, Department of Computer Science, West Michigan University*, Mar. 2001

[12] M. Mahoui, B. Bhargava, and M. Mohania. Data Mining for Web Security: User-watcher. In *Proc of the 2001 International Conference on Internet Computing*, Jun. 2001.