Policy and State based Secure Wrapper and its application to Mobile agents

Alexander Binun
Ben-Gurion University
Beer-Sheva
Israel
binun@cs.bgu.ac.il

Ehud Gudes
Ben-Gurion University
Beer-Sheva
Israel
ehud@cs.bgu.ac.il

ABSTRACT

Execution process in modern Web applications is usually represented as a partially ordered sequence of basic actions issued by a client (**login, buy**, **exit,** etc.; the login action usually precedes purchasing). Based on these actions, a finite automaton of fine-grained authorization checks, may be specified in a separate layer that is easily configurable for security needs of a particular application. In the Mobile case there may be two such state machines – one performing state-based authorization checks of the application execution process and the other performing such checks for the mobile agent execution process. Authorization checks of these machines may be both state-based and policy based, and the policies should distinguish between human clients and mobile agents cases.

We develop the framework to specify and enforce fine-grained state-based authorization checks of Web application execution, consisting of a Web browser (client) and a server. We adopt this framework to the Mobile case so that state machines representing fine-grained authorization checks of application and mobile agent execution are synchronized.

Keywords
Security, mobile agents, state-based fine-grained authorization.

1. INTRODUCTION

It is generally accepted that security should ideally be based on the concerned application. Only when the application logic is considered, it is possible to precisely determine security-related concerns such as context sensitive security requirements. Such requirements have most notably been expressed in the workflow context where, for example, someone's authorization to modify a given document depends on the current state of the workflow process.

It is our contention that application-layer security should be part of the design process of any major application and that an appropriate security model should be implemented as part of the application design process. It is, however, a fact that real-world applications are complex by nature and that it is very hard to ensure their correctness in general, and the implementation of their security features in particular. It is also common that complex applications will have different states and at each of these states may use different security rules. This situation is further complicated in the **Mobile** world environment, where instead of accessing applications directly users may send mobile software agents on their behalf who roam through the network and attempt to access various applications. Applications designers rarely have the expertise and knowledge required to protect their applications from such mobile agents.

In [8], Olivier and Gudes dealt with application based security by placing a `wrapper' around the application such that the user interface falls outside the wrapper and all communication between the user and the application has to occur via this wrapper. State-based behavior of an underlying application is modeled by supplying the wrapper with a finite automaton where each state is associated with a limited set of possible actions. A wrapper therefore contains enough information to conduct state-based access control checks at every state but does not subsume the security functionality of the underlying application.

In addition to the functionality described in [8], the extended wrapper (Ewrapper), presented in this paper supports the specification, resolution and enforcement of security policies at various levels of complexity. Security policies may be as simple as: do not accept requests from some class of URLs, or may involve more complex predicates that may also rely on application or history data. We argue that in some environments it is useful to also encode policies within a wrapper especially if they are general to more than one specific application. The

IEEE COMPUTER SOCIETY

syntax and semantics of such extended wrapper is discussed in this paper..

One environment where an extended wrapper may be useful is the environment of the *Mobile world*. Mobile computation is a paradigm based on the ability to launch an autonomous mobile agent to be executed remotely and act on behalf of the issued service. Mobile computation is attractive, because it is flexible and can surmount the deficiencies and extend the capabilities of traditional systems and improve the overall utilization of system resources.

Unfortunately, this paradigm also presents new security issues that fall into two main categories: securing and protecting the hosts from malicious agents and securing the agents from malicious hosts [12]. Looking at the issue of protecting the host from mobile agents, we note that many applications may share similar policies regarding mobile agents, thus, separating these policies into the wrapper may be more efficient and less prone to errors. Looking at the issue from the Mobile agent point of view, in order to enable a mobile agent to use a particular application, it may need to carry some sensitive information such as a credit-card number or a Role certificate. It is necessary therefore to preserve the integrity and confidentiality of such mobile agent, and the existence of the Ewrapper as a mediator between the agent and the application can be helpful for this purpose too.

The main contributions of this paper is first, the definition and specification of state-based security policies for both web applications and mobile agents. Treating both types of policies in one framework is new. Second, in the proposed design for the enforcement of these policies which is based on the use of the Wrapper architecture and the exchange of certificates.

The rest of this paper is structured as follows. Section 2 presents the related work. Section 3 presents an example application and discusses some example policies which we like the Ewrapper to enforce. Section 4 shows how to specify the policies and discusses the Ewrapper control and architecture. Section 5 discusses the mobile world case and the extended application and architecture for this case. We conclude and outline future research directions in section 6.

2. RELATED WORK

The specification of security policies for distributed applications is an active area of research. Damianou [9] surveys the various approaches for specifying and enforcing security in distributed systems and discusses the need for a policy language to support the specification of access control and other management policies. He proposes a policy framework to support security and management of distributed systems. The framework consists of a policy specification language and architecture for deploying policies based on the language and a set of tools for specifying and managing policies. Tripathi et.al.[3] suggests a language to specify policies for secure collaboration of agents. A compiler to produce the policy modules from these specifications is provided. The compiled specifications are interpreted to enforce collaboration policies.

Approaches that decouple security policies from applications, and enforce them in some generic and customizable way were suggested in [4,5,6,7,8]. Ribeiro et.al.[5] presents the language SPL to specify security policies for a system of distributed objects and a compiler to produce security modules mediating the whole communication between objects. SPL also provides the support for dynamic reorganization of the specified policies.

Beznosov [7] implements the full RBAC model [10], including dynamic reorganization of authorization rules and authentication mechanisms, using CORBA [2]. OASIS – Open Architecture for Secure Interworking Services (Bacon et.al [5]) is a role-based access control architecture for facilitating access control in distributed systems. OASIS, like Beznosov's RAD, supports generic specification and enforcement of authorization functionality thereby "decoupling" it from the application logic. The central idea of the OASIS model is credential-based activation of work-sessions. Assignment of users to roles and authorization are governed using concepts of activation rules and membership rules (a work-session of a user will not be continued unless a user is a member of some role). OASIS does not consider State-based information. On the other hand, OASIS includes the notion of Roles activation, which as we shall see in Section 5, can be useful for Mobile agents.

Biskup et.al.[6] suggest a general framework for performing state-based access control checks in a distributed objects system. A finite automaton describing state-based access control checks is represented as a sequence of states. The main idea of this approach is that such finite automaton does not subsume the security functionality of an underlying application, but serves as an additional security layer. Illegal (from the viewpoint of state-based access control rules) requests are rejected, and legal ones are passed to an application for further checks and execution. The approach of [6] is quite similar to the approach of [8] except that the copy of the state-based behavior is associated with a single object and not with the entire application. In [8], Olivier and Gudes elaborate on the above approach in the Web context. An additional layer, referred to as 'wrapper', that mediates the communication between the client and an application that

IEEE COMPUTER SOCIETY

contains all state-based logic is suggested. While Biskup et.al.[6] propose to attach a copy of the finite automaton representing state-based policy checks to every object in a distributed system, the wrapper suggested in [8] resides on a single host. As explained in the introduction, the main contribution of this paper is in the enhancement of [8] (and [6]) to handle policies, and its use in the Mobile agents case.

This paper also deals with security in the Mobile world. Mobile agents are software entities sent by clients over the network and may reside in one or more hosts to perform their computation. The mobile computation paradigm presents security issues that fall into two main categories: securing and protecting the active hosts from malicious agents and securing the agents from malicious hosts. It is worth to note that the term "agent" appears in some AI works and means "a software entity that is capable of performing autonomous actions". For example, Dignum [18] investigates collaboration principles of agent groups, pre-defined agreements before collaboration (so-called contracts). Other characteristics of autonomous actions performed by agents are also investigated. We do not deal with this notion of intelligent agent, but with simple software agents performing basic tasks in an e-commerce environment.

As it was explained before, security of mobile agents is classified into two categories. The first category, the protection of hosts from malicious foreign entities gained a lot of interest in recent years both in the research community and in industry. Current research approaches that aim to protect the hosts' objects and integrity rely on dedicated devices such as firewalls and on traditional methodologies such as access control mechanisms and cryptographic techniques [13,14].

The second category aims to protect the correctness and integrity of the agent's computation. This category is considered a "hard" problem due to the fact that the agent is fully controlled by the hosting computer. A malicious host can tamper with the agent's computation, for example, by manipulating the agent's code and/or intermediate results, stealing either secrets and/or digital money, or providing inaccurate / faulty data ,according to its preferences [15,16]. Approaches for solving it usually rely on cryptographic mechanisms and they may involve execution of an encrypted code on the one hand, or relying on encrypted communication between the client and the agent to verify the agent authenticity and integrity [17].

This paper does not deal with the second category and incorporates the first category within the Wrapper's security policies as will be shown in Section 5.

3. THE WEB BOOKSTORE

In this section, we provide a simple and a common e-Commerce scenario. Using the provided example we outline the motivations for extending the original Wrapper model. We assume that proper authentication and other cryptographic techniques are supported by the system.

Let us take as the example an electronic bookstore that supports two kinds of populations:

1) An entity in a role Client (C)- sends requests to the bookstore in order to purchase books or get books freely (there are few books in the store for which a client registered at a store does not have to pay)
2) An entity in a role Manager (M) - sends requests to the bookstore in order to administer it, catalog and order books, etc.

The electronic bookstore consists of 3 servers:

1) Administrative server A – maintains identification records and accounts for registered clients, a books catalog and an inventory
2) Pay-books-server P – maintains directories of books for which the client pays.
3) Free-books-server F – maintains directories of books that can be obtained without a fee.

Figure 1 depicts the state-based behavior of the bookstore. Each state transition in Fig.1 is represented as a triple **{requester, requested-server, action}**, where **requester** is either C or M, **requested-server** is either P or F, and **action** is a keyword describing a requested action (login, buy, etc.). A transition is done when **requester** performs **action** on **requested-server**. Figure 2 presents the general architecture of the Bookstore application including the Wrapper, for the client C:

IEEE COMPUTER SOCIETY

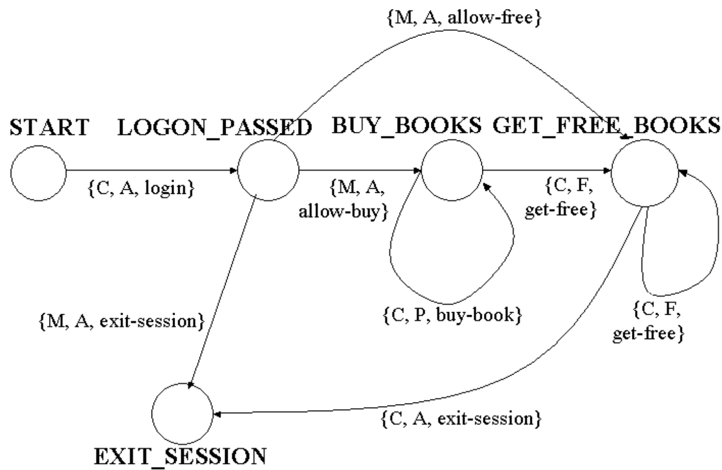

Figure 1. The state-based behavior of the electronic bookstore

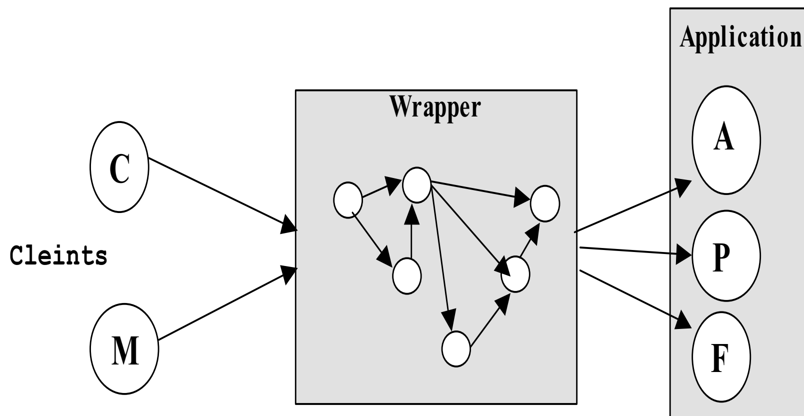

Figure 2: General architecture of the electronic bookstore application

When the client logs into the bookstore, it is assigned to a role “Client” based on its identity. The client identifies himself by username and is authenticated by the administrative database that resides at the server A. Upon completion of the login action the client sends the request **allow-buy, allow-free or exit-session** to the bookstore and the wrapper makes a corresponding state-based decision whether the client is permitted to either start with getting free books or start the purchasing phase and after that to get free books, or even to exit from the bookstore. The example above shows the state transitions of the bookstore application and the operations associated with each transition. However, some transitions may require additional checks to be defined by the application security policy.

A client that identified himself with username John is permitted to purchase only those books about databases which were published before the year 1999. This policy may be specified in a predicate form:

```
PROC john-purchase (request-type, client-role, client-id, book-name, book-year)
    if request-type="purchase" and client-id="John" and contains (book-name, "databases")
        if book-year<=1999 then return TRUE
        else return FALSE
    else return FALSE;
ENDPROC
```

The corresponding state transition is as follows:

```
BUY_BOOKS:
    …
 (john-purchase, BUY_BOOKS)
```


IEEE COMPUTER SOCIETY

In other words, a policy is represented as a constraint predicate on parameters of the request that should be checked before the transition is made. The wrapper state-based interpreter knows to parse and execute such predicates and the request is sent to the application based on these checks (see next section).

Another situation where additional checks will be required is as follows. Security checks may be represented as complete programs or modules developed by an application designer that contain implementations of security functions requiring communication with the application (we will refer to such checks as "black boxes"). Suppose that clients who have large debts on their accounts within the bookstore should not be permitted to purchase books. This requirement (called " good reputation"), should be checked before the purchasing phase. The check needs to scan the history of a client's actions that is kept within the application and therefore can be conducted by the application on behalf of the wrapper. This policy can be expressed as follows:

```
LOGON_PASSED:
    ( good-reputation, BUY_BOOKS),
    …
   EXECUTABLE good-reputation;
```

Here ***good-reputation*** is the name of an executable object.

Application security functionality, however, should not be embedded into the logic of the wrapper, in order not to compromise its customizability and not give an attacker any option to access application data by breaking the wrapper. An application developer can implement such particular security functions in a separate portable module (executable module, DLL or CORBA object) so that the name of this module will be known to the wrapper which launches this black box to perform the encapsulated security check. This will be elaborated further in Section 4.3.

4. THE POLICIES, CONTROL AND ARCHITECTURE OF THE EWRAPPER

In this section we formalize the definitions of policy specifications, discuss the Ewrapper's control and hint on its implementation. We start with definitions of some basic concepts.

4.1. POLICY SPECIFICATIONS

A *request* in the Extended wrapper is represented in the form of: *(request-type,Role , [*$param_1$ *...* $param_n$*]).* The semantics of *params* are interpreted in the context of *request-type. request-type and Role are* the mandatory part of the request. Optional *params* define the details of a request.

The Ewrapper security checks may be represented as a *predicate* or a *black box*. A *predicate* is a procedure that, being called with the request parameters, returns either success or failure. The body of a predicate is formed by variables that can be either parameters of the request or special environment variables recognized by the Ewrapper, basic control structures (if, while, begin) and applications of primitive operations supported by the Ewrapper control interpreter (string, logical, arithmetic). Expressions such as "only user in the role "rich client" may purchase books about databases" or "users may purchase books only after the login action succeeds" should be coded using predicates. An example for a predicate was shown in Section 3.

A *black box* is a module designed by the application developer, kept in some portable format and is capable of being launched from another process with returning either success or failure. Security checks involving application-specific logic and requiring access to application-specific resources should go to black boxes. An example of a black box is the "good reputation" check (see Section 3).

A *state-based policy* in the Ewrapper is a sequence of the form *(*$check_1$*, ...* $check_n$*, next-state)*, n>=1, where $check_i$ is either a predicate or a black box. The idea is that if upon arrival of a request ***r*** all checks in the state-based policy ***p*** succeed (we say that ***r*** *satisfies* ***p or r*** is *legal*) and ***r*** succeeds on the underlying application, the Ewrapper goes to ***p*****.next state**.

It is worth to note that a parameter list in a predicate specification may be followed by **CERT_ATTRIBUTES** keyword. Parameters of the certificate obtained with the current request and passed to each predicate are detailed in ***cert-attr*** and their values are accessible from within the predicate body. In such a way policies such as "certificates issued earlier than 19.08.2000 are rejected" can be constructed. The state-based policies are used in the automata specification as follows:

```
EWrapper ::= application-name,
   CHECKS
    (predicate_spec|black-box_spec)+
   END-CHECKS
   AUTOMATA (state-spec)+ END-AUTOMATA
predicate-spec::= PROC predicate-name
      (request-type,role, [optional-parameters])
      [CERT-ATTRIBUTES(cert-attr)]  body
   ENDPROC
black-box_spec::= EXECUTABLE box-name |
                    CORBA_OBJ box-name
 state-spec::= state-name:   (state-based-policy)+;
 state-based-policy::= (main-predicate-name,
   (predicate-name | box-name)* , next-state)
```

Here ***application-name, main-predicate-name, predicate-name and state-name*** are symbolic names of the wrapped application, predicates or a state respectively.

The specification of each state-based policy begins from a "main predicate". A main predicate performs the "main security check" associated with the **next-state** and is used to direct the Ewrapper to the checks that follow the main predicate in a given state-based policy. Typically, the main check deals with the most frequently used security checks, such as policy enforcement or checking the request-type or constraints imposed by the assigned role.

It is worth to note in our approach, the application designer is still responsible of *specifying* the predicate security checks, but he is not already responsible for *implementing* them (if security logic is embedded into the application source code, the application designer is responsible both of *specifying and implementing* security checks). Since application developers may not be completely trusted, our approach suggests mapping security specifications made by the developer into the wrapper predicates that are executed by a reliable and trusted program (the Ewrapper built-in interpreter). The application developer, however, remains responsible both of specifying and implementing application-specific security checks that go to black boxes.

The architecture of the wrapper is based on the foundation of the Web as depicted in Fig.3. The client is activated via a web browser and the application is accessed via the web server. All requests from the client to the application are first captured by a proxy which pass them to the wrapper. The wrapper, in turn, redirects these requests to the appropriate application. In order to facilitate the execution of "black boxes" the Ewrapper and the application are coupled by a CORBA infrastructure. This issue is more detailed in the next subsection.

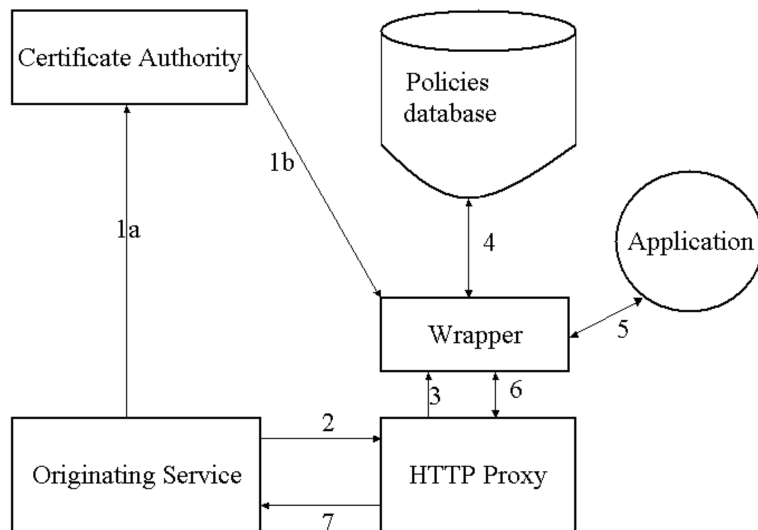

Figure 3. The General Architecture of the Web-based EWrapper

4.2. THE EWRAPPER'S CONTROL AND ARCHITECTURE

The Ewrapper's state machine is ensured by assuming that for each request r = *(req-type, ID, [opt-params])* and for each state S there exists at least one policy whose name correspond to the request type. The "execution" of the Ewrapper's state machine is carried out as follows

IEEE COMPUTER SOCIETY

(assuming *S* is the current state and a request *r* has arrived):

1) Try to find state-based-policy corresponding to request r.
2) If *p = (main-predicate,* $check_1$*, ...* $check_m$*, next-state)* is the policy found at step 1 (there can be at most one such p) and r succeeds on the underlying application, then go to *next-state.*

If either check fails then the wrapper reports FAILURE.

The detailed actions of the system are follows:

1a. The originating service obtains the encrypted certificate from the Certificate Authority containing the client's role ID for this system. (cf. 6.3)

1b. The Ewrapper obtains the Certificate Authority's public key in order to be able to decrypt clients' certificates and analyze their parameters (role ID checks, and so on)..

2. The Client sends a request to the proxy.

3. The request is redirected to the Ewrapper. The Ewrapper decrypts the client's certificate attached to the request and gains access to its parameters (role ID, and so on).

4. The Ewrapper fetches the policies from the policies database and performs the state-based checks.
During this stage a method **launch_black_box** may be invoked to execute a black box modul**e.**
5. The Ewrapper forwards legal requests to the application.

6. The results, either ones, are redirected to the proxy.

7. The proxy sends the results back to the client.

One may wonder about the similarity of the Ewrapper to an application Firewall. We like to emphasize the differences. First, enabling of state-based policies within the wrapper policies. Second, the closer coupling between the Ewrapper and the application by the "black box" checks which use application specific data.

4.3. IMPLEMENTATION ISSUES

In this section we briefly explain the Web-based implementation of the Ewrapper. We first want to explain the close coupling of the Ewrapper with the application. We assume the application has a front-end module in the Web-server which is basically a CGI script. The GET requests of this script are intercepted by the Ewrapper which ensures that each such request has passed the security checks.

As it was noted above, the client is a Web browser which communicates via Secure HTTP (sending GET requests) with the application (a CGI script). Once a login action succeeds, the client obtains a certificate (encrypted role ID and some additional parameters) in a **Set-Cookie** response header from the Certification Authority (which is also a CGI script). This certificate is presented to the application in the **Cookie** header of all subsequent requests. HTTP communication is performed via the HTTP proxy server. Each HTTP request is intercepted by the HTTP proxy server and decomposed into a certificate part (from the **Cookie** header) and additional parameters based on application semantics (a body of a GET request). The Ewrapper evaluates the predicates passing these parameters as arguments using its own built-in interpreter. A black box represented as an executable module that is launched by executing this module in a separate process passing GET request parameters as command line arguments and examining the return code. To execute a black box, the Name Service is contacted to obtain a reference to the CORBA object implementing a given black box logic from the name of a black box being specified. The CORBA object is evaluated by invoking the check method using the reference to this object just obtained from the Name Service. GET request parameters are passed as arguments to this method.

4.4 THE WRAPPER-APPLICATION RELATIONSHIP

Specifying an automaton may be done easily by hand, but it is harder to create one for larger real-world web applications. If one uses an external tool for such specification one runs the danger of creating inconsistencies between the application and the wrapper. We propose an automatic tool for this purpose.

It is assumed that the application consists of a set of CGI scripts (they may be written in C, Scheme, Java or any other language). The semi-automatic scheme we propose involves insertion *by the application developer* statements to specify the current state and possible transitions for each such GET in an appropriate place of a source code where it is handled. These statements are wrapped by language-specific comment signs **(/*P … P*/** in C++, for example; the letter P shows that it is not a " usual" comment, but rather a security policy specification. The application developer should maintain such "commented" CGI script in order to ease the work of the security expert creating Ewrapper policies. Then the "policy parser", implemented by us scans the source code, extracts "commented" security policies, generates all possible sequences of **{state, policy, next-state**} triples. Predicate bodies and black box declarations are extracted into a separate section (in our case, **PROCS …**

ENDPROCS section). Later on the application developer needs only to maintain "security-commented" CGI scripts. Because of the large variety of script languages we decided not to design a general parser that parses the source code of CGI scripts and uses program analysis techniques to deal with semantics of an implementation language.

5. THE MOBILE WORLD – THE USE OF CERTIFICATES

In this section we discuss the use of the Ewrapper in the Mobile world case. We first present an extension to the electronic bookstore example where the client submits a mobile agent to roam through the network in order to find the best server to purchase books from. Next we outline some of the new issues introduced by the Mobile World. Then we suggest some ideas of extending the Ewrapper to support these issues.

5.1 THE EXTENDED ELECTRONIC BOOKSTORE

The following entities participate in the Bookstore scenario:

1) **Client *C*** sends a mobile agent MA to hosts of the bookstores to carry out actions on its behalf. A MA is supplied by the client with a certificate so that the system is enabled to assign a MA with the proper role and policies. The MA also carries a list of wanted books (however, it can connect to the client to obtain requests to other books). The client can also connect to the hosts without using mobile agents (as in Section 3) to perform actions.

2) **Hosts (AS, H_1, H_2)** contain information about books. H_1 contains simple books, while H_2 keeps information about valuable books and mobile agents should present an additional certificate (obtained from an originating service) to prove the capability of accessing sensible information. Administrative Server (AS) keeps the information needed for authentication.

The new issue here is that the MA is also associated with state-based policies. The state-based behaviors of the MA and of the application are represented by two separate state-based automatons.

First, the MA arrives at the AS and identifies itself with the certificate. *AS* checks the validity of the certificate and examines the account of the user on whose behalf the *MA* acts and assigns it with the specified role. The MA may perform the following actions:

1) Exit the session (if all the listed wanted books are purchased, or for any other reason).
2) Try to purchase a book from the current host based on its preferences.(if the price is sufficiently good) or present required certificates to the application.
3) Migrate to another host for different reasons. In particular migrate to a *protected host* to purchase books of a particular sensitive subject. In some cases, the MA may require a *quality certificate* from the protected host to insure that it communicates with a reliable host and purchases a valid book.
4) Connect to the originating service in order to obtain additional information (additional wanted books or credentials needed to proceed with purchasing on a protected host containing "valuable" books) or send back to the originator a "certificate of quality" obtained from the application. It will be demonstrated below how these two actions are interrelated.

In order to purchase books from the "protected host" the MA may need a special certificate from the client, thus it may need to communicate with the client in order to receive it or communicate directly with the Certificate authority. This communication obviously needs to be protected from the current host. Suppose that a MA arrived to a protected host that contains valuable books so that more thorough security checks than presenting an usual certificate are needed. To purchase a book on a protected host, the MA does the following:

1) Gets an additional certificate AC from the originating service (possibly presenting the protected host name or parameters of a wanted book), by performing **get-AC** action.
2) Presents AC to the application (by performing **Present-AC** action on the current host).
3) Obtains a Quality Certificate (QC) from the *application* that indicates that a legal AC was presented and a requested book can be purchased. QC contains the host validation data concerning the requested book (**obtain-QC** action)
4) The MA sends the QC to the originating service by performing **send-back-QC** action (intuitively, it asks the originating service whether the "quality" of the book represented by QC satisfies it).
5) In the case of positive response from the originating service at the step 3, the MA purchases the book.

Figure 4 provides the state-based behavior diagram of the mobile agent (where the wrapper states are underlined, and state-based policies are *inclined*).

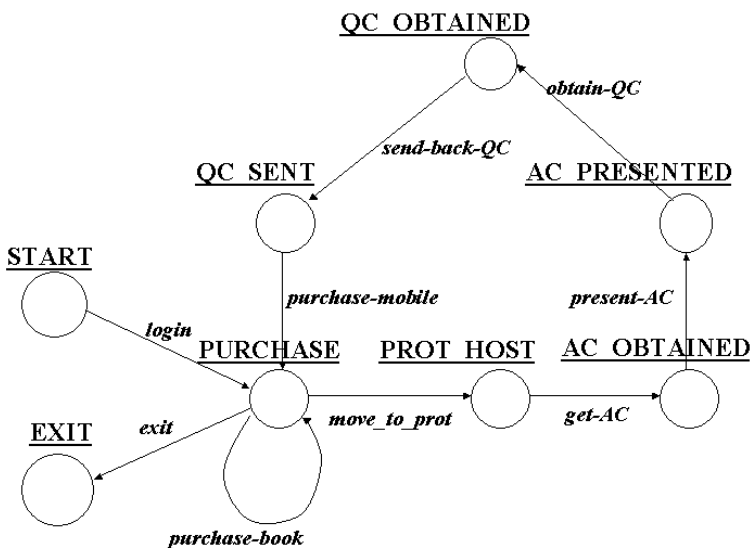

Figure 4. The State-based diagram of the MA in the extended bookstore

Note that there is a correlation between the state-based security policy of the application and the state-based computational process of the MA, since the MA should be able to deal with all different responses of the MA state machine. Thus the MA states must be synchronized with the corresponding wrapper states and must be encoded and protected within the MA executable code. The certificate exchange actions described above are needed to enforce state-based protection of the MA computational process. The application state machine is provided in Fig.5:

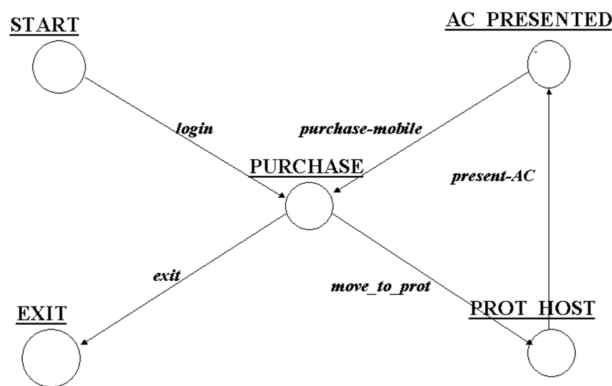

Figure 5. The application state machine

That is, to protect MA execution, three certificate exchange actions (**get-AC, obtain-QC** and **send-back-QC**) are added to the application state machine protecting application hosts. So, certain sensitive actions of the MA (purchasing books on protected hosts) are prevented unless the MA has additional privileges to do them. Figure 6 demonstrates how the MA execution is secured by using additional certificates:

IEEE COMPUTER SOCIETY

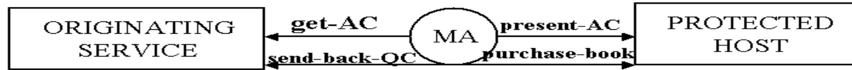

Figure 6. Securing the MA execution via certificate exchange

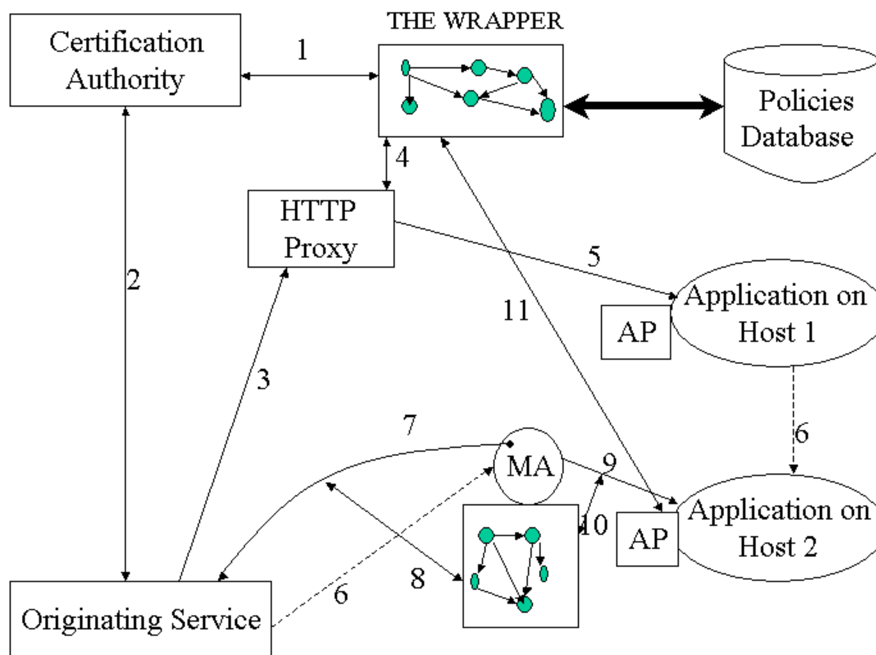

Figure 7: The EWrapper Architecture for Supporting the Mobile world

5.2 APPLYING THE EWRAPPER TO SUPPORT MOBILE AGENTS

The Ewrapper suggested in this paper can be easily extended to support the additional policy requirements introduced by the mobile paradigm as presented in figure 4. Note that *purchase-mobile* policy will be activated only when the Ewrapper identifies the request as coming from a Mobile agent and not from a regular client. This identification can easily be done by using common techniques for distinguishing between a human client and a mobile agent (e.g. by the use of common-sense questions…)

Figure 7 presents the modified architecture of Figure 3 for the Mobile world. The differences are the existence of a MA as a separate entity and the existence of a host it can migrate into.

Each host runs a special trusted program, named Agent Platform (AP). This program presents the execution environment for mobile agents relocating on a given host. It is the responsibility of an AP to ensure that MA code is not tampered with; to supply the MA with an decrypted role ID so that it can be presented to a host while doing

IEEE COMPUTER SOCIETY

local requests and to ensure the MA state-based security checks are not bypassed. We assume an AP is the reliable platform produced by some trusted software company. It is the AP responsibility to ensure that the MA's state-based security policy checks are done correctly. This is done by a module attached to the MA called "the MA state machine". (see also cf. 5.3). Our model executes in one of three main modes: the preparation mode, the "non-mobile" call mode and "mobile action" mode. Note that actions of the preparation mode precede all other actions of our system. Non-mobile and mobile calls can interchange. In each mode, actions occur in the same order as they are described below.

The preparation mode is described by steps 1 and 2:

1) The Ewrapper obtains the public key of the Certificate Authority via the secured channel (the Ewrapper will use this key for decrypting certificates attached to the client's requests).

2) The originating service obtains the certificate encrypted by the Certificate Authority's private key. The Ewrapper decrypts the certificate and gains access to its attributes while performing mobile or non-mobile calls (see further).

A **non-mobile call** is performed as in Section 4 (steps 3-5).

3) The service issues the HTTP request **r** to the application (a request in intercepted by the HTTP proxy).

4) The HTTP proxy consults the Ewrapper to perform state-based check of **r's** legality. Certificates attached to **r** are decrypted by the Certification authority's public key held by the Ewrapper. Next the state-based policies are checked.

5) Once **r** is legal, it is redirected to the appropriate host and executed. Upon returning a positive response, the current state of the Ewrapper is advanced.

A **mobile call** is performed as described by action sequences 6-7-8 or 6-9-10-11**.**

6) The MA is sent to a given host (Host 2 in this example) by the originating service or has migrated from another host, relocated and accepted by AP.

Then one of following actions can occur:

7) The MA perform the request **c** to contact the originating service and to obtain an additional certificate (for protected hosts, for example).

8) **C** is checked at the MA's state machine. If the response succeeds at the MA (processes by the MA code with the positive result), the current state of the MA state machine is advanced (for example, after the MA state machine `recognizes a valid certificate obtained from the originator).

Or:

9) The MA performs a local action A on the current host trying to access application resources (purchase a book or migrate, for example).

10) A is checked at the MA state machine.

11) Once this action is legal, the AP (Agent Platform - see below) contacts the Ewrapper to perform state-based check of A's legality. Certificates attached to A are decrypted by the Certification authority's public key held by the Ewrapper. Once A succeeds at the application, the response is sent back to the MA. If the response succeeds at the MA (processed properly by the MA code with the positive answer, the current states of both state machines (of the application and of the MA) are advanced and A is assumed to be legal.

During state-based checks the policies are fetched from the policy database (see thick arrow between Wrapper and Policy database).

Note that a mobile call described in steps 7-8 (contact to the originating service) is not aimed to access application resources on the current host. Only one state based check (at the MA state machine) is performed. In contrast, two state based checks are performed in a mobile call described in steps 9-11, and current states are advanced in a **transactional manner** (e.g. both states are advanced or no state is advanced). This solution is dictated by real situations – an action is considered to be successful if and only if the application and the MA have processed it properly. We also note that developers of mobile agents in this environment must be aware of the Wrapper architecture and the state-based policies, and design their agents to take advantage of these capabilities. One can think about specialized software engineering tools to help develop both Agents and Applications in this environment.

5.3 THE MOBILE AGENTS - IMPLEMENTATION ISSUES

We have implemented a prototype infrastructure to facilitate execution of mobile agents and the sample mobile agent executing the scenario of Section 5.1. We choose Scheme as the implementation language.

A MA in our infrastructure is represented as a structure consisting of the following fields:

1) The MA code (is represented as a Scheme list, but can be executed by an AP using Scheme *eval* primitive. *Code-data duality* of script languages is exploited here.

2) The representation of the MA state automaton (is represented as a Scheme list and compiled by an AP from a specification file before launching a MA).

3) The current state of the MA state automaton.

4) Host/port of the originating service.

5) The role ID given to MA by the originating service.

6) The additional parameters comprising the MA computational state (will be detailed further).

The MA code is a lambda-optional (i.e. function with optional number of parameters). It may contain calls to mobile primitives (**agent-jump, contact-to-originator, perform-local-action**) whose implementations are loaded into memory by an AP so that they can be executed when an AP evaluates the MA code.

The parameters described above carry all necessary information to resume the MA execution on the host the MA is launched to (by calling lambda-optional representing the MA code and passing role ID and optional parameters of **the agent-jump or agent-submit** calls as arguments). We will refer to them as *the MA computation state*.

The primitives aimed to facilitate mobile agent functionality in our infrastructure are specified as follows:

1) **agent-submit (target-host role-ID agent-code MA-automaton-file [opt-params])** - is called by the originating service in order to supply a MA with role **role-ID** (whose code is represented by **agent-code)** and to launch this MA to **target-host**.. The MA state machine is compiled by the AP from the file **MA-automaton-file.**

2) **agent-jump (target-host [opt-params])** – is called from within the executing MA code to stop the MA execution at the current host and resume it at **target-host** .

3) **contact-to-originator (value)** – is called from within the executing MA code to send **value** via TCP/IP to the originating service. The response is returned as the result of the call.

When the control is passed to **agent-jump** or the originating service performs **agent-submit** call, the AP on the current host sends the MA computation state (as was detailed above) to the target host host via TCP/IP.. The AP on the target host accepts this package, saves the MA computational state at the top level and executes the MA code via *eval* . During execution, the current state of the MA state automaton may change due to local actions on the application (**perform-local-action**), certificate exchange actions (**contact-to-originator**) or migration (**agent-jump**). An AP ensures all these actions do not bypass state-based checks on the MA automaton and all actions accessing application resources (migration and local actions caused by **perform-local-action** are checked at the application Ewrapper also). All computational state changes are reflected at the top-level. The parameters of the computation saved at the top-level facilitate execution of mobile primitives' implementations (**agent-jump** sends the MA computational state to the destination host via TCP/IP; **contact-to-originator** connects to the originating service using its host/port IDs contained in the MA computation state).

6. CONCLUSIONS

In this paper we introduced a Policy-based wrapper. The wrapper is a middleware that mediates the communication between the clients and the server application while enforcing the system security policies. The wrapper releases application programmers from dealing with security issues and policies that are basically provisioned by the system. Different policies and level of trust might be applied on the different agents.

Furthermore, the decoupling of the of security policies issues and the application issues provides an easy and modular maintenance of current design while also enables the assimilation of new and complicated policies into a system without requiring the modification of the applications themselves.

We then extended this framework to the Mobile world case and showed how various policies for the mobile world can be incorporated into the same framework where policies for governing the execution of mobile agents might be at various levels of complexity. Finally we discussed briefly a prototype implementation of the Mobile Ewrapper architecture using Scheme. We like to emphasize here that there is strong need for a general authorization model in the Mobile world which is a major subject for future research.

ACKNOWLEDGEMENTS

We like to thank Asnat Elichai for providing some references on Mobile agents and for her help in Figures 3 and 5.

REFERENCES

[1] Garfinkel S., G.Spafford. Practical Unix & Internet Security. Second ed. O'Reilly 1998.

[2] www.omg.org/technology/corba - CORBA Home Page.

[3] Tripathi T., Ahmed T., Kumar R.,Jaman S. Design of a policy-driven middleware for secure collaborations. In Proceedings of ICDCS 2002, 393-400.

[4] Ribeiro, C., Zuquete A. and P. Ferreira, SPL: An access control language for security policies with complex constraints. In Proceedings of the Network Security Symposium, (NDSS'01), San Diego, California, February 2001.

[5] Jean Bacon, Ken Moody, Walt Yao: A model of OASIS role-based access control and its support for active security. In ACM transactions on Information systems security (TISEC :(4)5 492 -540, 2002

[6] J.Biskup, T.Leineweber. About the Enforcement of State-Based Security Specifications. In Proceedings of IFIP WG11.3 Conference on Database Security, VII. Elsevier (North-Holland), 1994

[7] Konstantin Beznosov, Yi Deng, Bob Blakley, C. Burt, John F. Barkley: A Resource Access Decision Service for CORBA-Based Distributed Systems. In Proceedings of ACSAC 1999: 310-319.

[8] M.Olivier, E.Gudes. Wrappers - A Mechanism to Support State-Based Authorization in Web Applications. In Proceedings of IFIP WG11.3 conference, Amsterdam, 2000.

[9] N.Damianou. A Policy Framework for Management of Distributed Systems.Ph.D Thesis, Imperial College of Science, University of London, April 2002.

[10] R.S.Sandhu, E.J.Coyne, H.L.Feinstein, C.E.Youman. Role-Based Access Control Models. IEEE Computer, Feb.1996.

[11] R.Oppliger, Authorization Methods for E-Commerce Applications. In Proceedings of the International Workshop on Electronic Commerce, 19-22.10, Lausanne, 1999.

[12] Cambell R., Sturman D., Tock T. Mobile computing, security and delegation,, Int. workshop on Mobile computing, Japan, 1994.

[13] Blaze m., Feigenbaum J. Lacy J, Decentralized trust management, In Proceedings of 17th IEEE symposium on security and privacy, 1996, pp. 164-173.

[14] Woo, T.y and S. Lam, Designing a distributed authorization service, In Proceedings of IEEE INFOCOM 1998.

[15] Greenberg, M.S, Byington, J.C, Harper D.G, Mobile agents and security, IEEE Communications magazine, July 1998

[16] Hohl, F. A model of attacks of malicious hosts against mobile agents, In Proceedings of 4th ECOP workshop on mobile object systems, 1998.

[17] Elichai A. and D. Raz, The security computing tradeoff in mobile computing, technical report, Ben-Gurion university, 2002.

[18] F. Dignum, D. Morley, E. A. Sonenberg and L. Cavedon, "*Towards socially sophisticated BDI agents*". In Proceedings of the ICMAS 2000, pages 111-118, 2000.