Security Agent Based Distributed Authorization : An Approach

Vijay Varadharajan, Nikhir Kumar and Yi Mu

Distributed System and Network Security Research,
School of Computing and IT, University of Western Sydney, Nepean, Australia

Abstract

This paper considers a security agent based approach to authorization in a distributed environment. A security agent is used to capture the privileges and a part of security policy in distributed authorization. Principals make use of these agents to carry out their requests on remote hosts. Targets verify the authenticity of the security agent and its privileges and use them together with their local security policy to grant or deny the requests. The paper describes the operation of the authorization system using these security agents and how these agents can be used to support dynamic decision making. Finally, the paper also considers how the security agent based model answers the taxonomy questions that arise in the design of capability based secure systems.

1. Introduction

In a distributed system, when a request for a certain service is received by one principal from another, the receiving principal essentially needs to address two questions. Firstly, is the requesting principal the one it claims to be? Secondly, does the requesting principal have appropriate privileges for the requested service? These two questions relate to the issues of authentication and authorization. There are also other security concerns such as auditing, secure communication, availability and accountability. There have been several efforts in recent years which have addressed distributed authentication. However distributed authorization has not been considered to the same extent, even though there have been a large body of work in access control for stand alone systems over a number of years and relational databases. In fact, several open problems remain in the design and the implementation of secure authorization service for distributed applications, especially in large scale commercial systems.

The situation gets further complicated if the software programs in the distributed application are themselves mobile. Mobile code paradigms allow migration of code describing the service and possibly the associated state of execution to a different host. There are different types of mobile code paradigms such as code on demand, remote evaluation and mobile agent. The term “mobile agents” is an overused term at present with a variety of definitions ranging from autonomous agents to intelligent agents to co-operating and collaborating agents. Also there are different design choices as to which components migrate when a mobile agent moves. For instance, in Java [1] only program codes migrate whereas in Obliq [2] closures consisting of program code along with the environment that binds variables to values or memory locations also migrate. We will use the term mobile agent to refer to program codes along with the state containing data variables and its values along with program counter and recursion stack.

There are some security issues that are specific to a mobile agent system. These include the protection of host against the agents, the protection of agents from each other, the protection of the agent from the host

and the protection of the underlying network. The Java sandbox [3] security has, for instance, enabled developers to make some progress toward solving rogue agent problem. However there is no a clear solution at present to tackle the problem of an agent being attacked by the host computer. Some current agent systems offer basic privacy mechanisms such as a secure channel between machines via encryption of messages between agents while some others provide a certain level of authentication and integrity. Recently, though there have been several works addressing aspects of mobile agent security (see for instance [4,5,6,7]), several challenges remain and there has not been much work on developing an overall security model for mobile agent based systems. Furthermore, the "weakest link problem" of security is exasperated by mobile agent technology. For instance, when an agent migrates from a server in one company to another, the security is only as good as the weakest agent server against the strongest rogue agent. In [8], we consider some issues that need to be addressed when designing a security model for a general mobile agent based computing system. In this paper, we consider a slightly different situation and look at how the agent technology can be employed in a "capability" type manner to provide distributed authorization. In this context, a security agent is used to capture the privileges and part of security policies required in distributed applications which can then be used as a dynamic capability in providing distributed authorization.

This paper is organized as follows. In section 2, we consider some of the key architectural design principles involved in the design of a distributed authorization service. Section 3 outlines the agent environment and describes the security agent based authorization system. It describes the security management elements in the agent based architecture and defines the structure of the security agents. Section 4 considers the operations of the security agent based authorization system such as the creation of the security agent, the distribution and propagation of security agents, delegation and revocation issues. Finally, section 5 considers how the security agent based model addresses the taxonomy questions that arise in the design of capability based secure systems [12].

2. Authorization Systems

2.1 Architectural Design Principles

The design of any security service involves at least the following aspects: security information used in the provision of the security service, the security mechanisms that are required to support the service and the authorities involved in the management of the service. In designing an authorization service, the security information used can range from user identities to group identities, to role information, to location information, and to actions and parameters associated with the actions. In fact, from an architecture point of view, it is important to understand the characteristics of the different types of the authorization security information. Some security information are *generic* and *static* in nature. For instance, typically identity based information falls into this category. This is generic in the sense that whatever application the user may use, often the user's identity is the same and this remains more or less static. Then there are security information that are *specific* but still somewhat *static* in nature. For instance, role based access information falls into this category. Roles are specific to organizations and they are reasonably static in the sense that they are unlikely to change on a day to day or even on a monthly basis. In fact, one of the main benefits of access control system based on role based information is to reduce the effect of the changes in the user population on the management of access privileges. Then there are security information that are *specific* and *dynamic* in nature. These are *specific* in the sense they may relate to specific applications or parts of applications or files. They tend to be *dynamic* in the sense they are prone to state changes.

Such a classification of different types of authorization information in turn helps to understand better the requirements on their management. First, the authorities involved in the management of these different types of authorization information are likely to be different. Not only the strategies with respect to when the changes and updates to these information take place are likely to be different but also who are allowed to make these changes are likely to vary. For instance, the specification and changes to the role information in an organization will be the responsibility of a certain group of people who can be different to those responsible for setting the privileges for a specific file or application in a server.

From an architectural point of view, such a characterization also leads to some basic design principles. Static and generic authorization information can be "pushed" by the client principal to the target principal. In fact, static and specific information can also be pushed in a similar fashion. For instance, if we consider a capability based access control model, privileges and their validity can be “pushed” to the target. It is appropriate to store the domain specific and dynamic rights at or near the target, enabling the target system authorities to be involved in their management. Such specific and dynamic authorization information needs to be "pulled" at the time of the decision process. Based on these principles, we consider a security agent based authorization system. First let us briefly consider how such an agent based authorization relates to the traditional access control mechanisms.

2.2 Authorization Mechanisms

Authorization systems are concerned with making decisions on who can access what operations on which objects and how this can be achieved. Traditionally, the mechanisms that have be used in the provision of authorization service have included access control lists (ACLs) and capabilities. Access control lists associate lists of users to every type of access for a resource. The access types are typically actions such as read, write, execute, create, append, modify and delete. Any user on the list for a particular action is allowed to perform that action. ACLs are very common in file systems. The benefits and problems of ACLs are very well known. However it is important to note that ACLs are inherently resource based. Most systems today store the ACLs at the resource (e.g. within the application). A common nightmare with ACL based systems is the difficulty of their maintenance especially if the administrative tools are not common across the ACLs in all applications. If the authorization information itself is distributed which is the case in distributed applications, it makes answering questions about a user's access rights very difficult. For example, to remove an employee who has left the company from all ACLs usually requires the administrator to edit the ACLs in each application to determine if the user has access rights, and if so, remove those rights. This can be a very time consuming process. The other well known access control mechanism is the capabilities. A capability is essentially a ticket that provides the holder with some type of access right. A capability system requires the user to acquire this ticket before performing the access; the ticket is presented at the time of access. If all the information necessary for computing the access rights is available outside the application, then a capability system can work well. However in applications where some of the information necessary to make the access determination is inside the application, the capability system does not have all the information it needs. For instance, in an Automated Teller Machine (ATM) system, a capability system might be able to determine that the ATM card looks valid, the check digit on the account is correct, the bank information looks valid and so on, but only the bank applications know if the request to withdraw money is valid since only the bank applications know whether the account has enough money in the account and if the user is over his daily withdrawal limit. The decision to allow or deny the request must be made from within the application and is dependent on its state. It is clear that combining these two mechanisms can lead to a better access control system and there have been earlier work along these lines such as [9] and [10].

Let us now consider a variation on the access mechanism. Consider a security agent that acts in some sense like a capability in that it contains the privileges that can be used to perform certain actions at the target. But it is different from a traditional capability in that it may contain different types of identities, for instance, the identity of the principal who created the agent (creator) and the identity of the principal who is sending the agent. The agent is much more general than a traditional capability in that the security agent contains code that can capture certain access policies which can make the agent's behaviour to vary according to the circumstances, as specified by the creator of the agent. This can enable the security agent to take dynamic decisions which the static ticket type capabilities are unable to achieve. Hence such security agents can be used to support better dynamic access control policies. At the target, where the agent is executed, the agent's behaviour and access is constrained by the security policy at the target. In this sense, the authorization system has the characteristic of an access control list. Hence the security agent based authorization model has characteristics of both access control list and capabilities. Broadly speaking, the security agent captures part of the authorization policy (which is more than the traditional capability) and the target has part of the policy which controls the behaviour of the agent in its environment.

In a distributed environment, often the need for an entity to act on behalf of another arises [11]. This is particularly true in the case of mobile agents which often perform their actions on behalf of the sender. A delegation is a temporary permit issued by a delegator to a delegate that authorizes the delegate to act on its behalf in performing certain actions. In this case, the target needs to verify whether the delegator has the appropriate privileges, whether the delegator has actually transferred the privileges to the delegate and whether it is the delegate which is making the request. We need to consider mechanisms for providing secure delegation when the agent moves from one host to another performing actions on behalf of various principals. Delegation and revocation go hand in hand. One natural revocation mechanism is the use of timeout whereby the privileges that an agent has expires after a certain specified time. There may also be a need to revoke the privileges of the agent before the expiry time. Let us now consider the structure of the security agent and the agent environment.

3. Security Agents Based Authorization

3.1 Security Agents and Environment

One can view an agent as an entity that can be dispatched from one host to another and can be executed at the remote host. In our case, we assume that the agent migration involves transfer of program code as well as its data. It is considered to be autonomous as it has its own thread of execution after arriving at a host. The identity of the execution environments in the host is same as the identity of the host (e.g. the URL) along with an appropriate qualifier if there are more than one execution environment within the host. We assume that an agent executes in a secure environment in agent enabled hosts. Each agent enabled host has a security management component which is concerned with the security of the host and its execution environments. We refer to this principal as Security Management Component (SMC). An agent has a unique identity which is independent of the execution environment of a host. Agents communicate via messages. The collection of hosts which obey the same security policies are grouped together in a domain. Each domain has a security authority referred to as the Security Management Authority (SMA). The SMA principal interacts with the SMCs in the domain in the establishment and maintenance of security policies within the domain and it interacts with SMAs of other domains in inter-domain situations. In practice, we envisage the SMA's role to include some of the functions of a Certification Authority.

When an agent arrives at a host and runs within the execution environment offered by the host, security services are required to address at least the following aspects. On the one hand, the capabilities of the agent need to be restricted to ensure that the local resources of the host are protected from unauthorized actions by the agent. Note that our security agent contains not only the code (or the actions that it can do) but also some of the privileges and other security characteristics that are required to perform these actions. That is, a part of the security policy information has been incorporated within the agent. This information is used by the SMC at the target in making its access decision. For instance, the security policy within the SMC can be based on the privileges information of the security agent, the identity of the principal which created the agent in the first place, the client host principal which sent the agent and other characteristics of the agent itself such as the level of trust associated with it and its state as well as the state of the target application. For instance, an agent may have a method *withdraw* created by Bank-A sent by Customer-C_Bank-A allowing the agent to withdraw money from the account of Customer-C if state of the account meets certain conditions (e.g. > 0). On the other hand, the agent itself needs to be protected against unauthorized tampering by the host in which the agent is currently residing. At present, there are no easy practical solutions to this problem.

3.2 Security Agents based Authorization Model

Conceptually, the security agent based authorization model has the following elements: Security Management Authority (SMA), Security Management Component (SMC), Object Managers (OM) and Security Agents (SA). Each resource or host which needs to protect its objects has a SMC and each domain has an SMA. Both SMCs and SMA are trusted entities and they are involved in the creation and distribution of the SAs. As the name implies, OMs are responsible for the management of the objects that reside in the host. OMs use the SMC to generate and validate the SAs. The SMC also maintains security information such as public and private keys as well as a list of some of the commonly used certificates and name servers. In an inter-domain situation, the SMA in one domain will be involved in negotiating with the SMA in another domain concerning the validation of SAs that have originated from the other domain. We assume that each SMC has a public key and private key pair and so does each SMA.

OMs are persistent objects or kernel services or libraries. The OM defines the privileges of an object at the time of creation of the object and these are passed to the SMC which then encapsulates them in a SA and distributes it to the clients. We will see below that the OM also adds some default agent methods or code in the SA. Whenever a client principal wishes to access a target, it passes its request along with the SA. The OM at the target interacts with the SMC to verify the client principal, the SA and to determine whether the request is to be granted or not. The target SMC also maintains a duplication list which is used to detect unauthorized repeated use of the SA. We will return to this issue in section 4.2. As the SA is a full fledged object (program and data) and it has the ability to gather information relevant to its requests as it moves from host to host, it can use them to take dynamic decisions on behalf of clients. It is not necessary for the SA to return back to the sending client. There are several issues to consider regarding the SA. It is essential that a SA should be unforgeable. Also the SA should only have the capability to make those decisions for which it has been allowed to do so and it should not make any unauthorized decisions and requests without detection. The SA needs to prove to the target that it has not been compromised in any way. Furthermore, an additional issue arises as to where should the agent be when making such decisions. In general, the agent should move to a trusted or a neutral host as far as the sender is concerned when making the decisions. This is because in many practical situations, not all participating hosts trust each other. For instance, one host may try to change the state of the agent in an unauthorized manner thereby causing a different agent behaviour.

3.3 Security Agent Structure

The basic capability type operation using security agents is as follows. An OM principal creates a SA to protect a set of its objects in the host. That is, the OM is acting as the creator principal. The SA is then distributed to clients. Then a client sends the SA to the target host requesting to perform certain action. The structure of SA has several elements containing information about the privileges of the principal, on the validity of the privilege information as well as identity and other information gathered during its propagation. It contains the following:

- Identifiers: <SA Identifier,
  Creator-OM Identifier: Principal Certificate: <Principal | Group | Role>,
  Creator-SMC Identifier: SMC Certificate,
  Cloning-SMC Identifier: SMC Certificate, Timestamp, Lifetime>
- Privilege Tokens: {<Privileges, Nonce, Timestamp, Lifetime, Flag>}
- Delegation Tokens: {<Delegator, Delegate, Privileges, Nonce, Timestamp, Lifetime, Flag>}
- Agent Code: < Default Code, Creator-OM Code, Client Code>
- Rights Tokens: {<Principal, Rights, Timestamp, Lifetime>}
- Request: {Client Identifier: Principal Certificate: <Principal | Group | Role>,
  SMC Identifier: SMC Certificate, Operation, Timestamp, Lifetime}
- Data Store: <Data, Propagation List>
- Security Tags: <Security-Tag-OM, Security-Tag-S>

Identifiers: The SA identifier is a number associated to the agent by the creator OM. The Creator-OM identifier indicates the identity of the OM principal who created the agent. This is usually in the form of a principal certificate which contains the principal identity and may optionally have the role and/or the group the OM belongs to. The principal certificate contains the identity of the OM and the OM's public key signed by the SMC using its private key. As there is a single SMC per host, the SMC also identifies the host where the agent is created. The Creator-SMC identifier contains the certificate of the SMC signed by the SMA. The timestamp identifies the time at which the agent is created and the lifetime indicates the intended lifetime of the agent. In the case of cloning, it is necessary to include the identifier of the SMC where the cloning occurs; otherwise it is left empty. We will return to this when we consider the cloning method below in the description of default code.

Privilege Token: A Privilege Token specifies the set of privileges of the agent. These are specified as a set of methods or operations that are applicable to a set of objects in a particular class. The OM determines which privileges are to be accorded to a given agent. These privileges will be used in conjunction with the policy at the target host in determining whether a request by an agent is to be allowed or not. Each privilege token has a nonce which acts like an identifier, a timestamp which indicates the time at which this token was created along with a lifetime. The flag in the token defines whether the privileges can be delegated. If the principal wishes to allow another principal to delegate the privileges in the token, then this flag parameter is set to "delegate"; otherwise it is set to "final". The SMC at the host then carries out the appropriate cryptographic sealing which is discussed below.

Delegation Token: This token specifies a set of delegated privileges. Each delegated instance specifies the identity of the delegator, the identity of the delegate as well as the privileges that have been delegated along with a nonce, timestamp and a lifetime. Typically the delegator could be the principal who created

the agent (e.g. the Creator-OM). The delegate can make use of the delegated privileges in its requests when using this agent. Nonce acts as an identifier of the delegated instance, timestamp indicates the time at which this delegated instance was created and the lifetime indicates the time period this delegation is valid. The flag in the delegation token indicates whether the delegate can make further delegations. There is a default delegation method in the agent code which is described below.

Agent Code: The agent code is the core part of the SA. It consists of a set of executable methods or code. The creator code of the agent is specified by the OM. The default set of methods or code is automatically added when a SA is created by the SMC. Furthermore, the general model allows client code which may be added by principals other than the creator that may be used for specific decision making.

At present, the following methods in the Default Code are envisaged:

- Delegation method
- Current Request method
- Propagation List Creation method
- Cloning method
- Security Checksum method

If the token type flag is set to “delegate”, then the delegation method is activated. It is used to create the set of delegated privileges. It collects the privileges to be delegated which are either a subset or total of the original set of privileges specified, and inserts the nonce, timestamp and the lifetime. These are specified by the delegator principal. The lifetime can be used to control the period of delegation and the nonce and the timestamp can be used to prevent the same delegation token being used repeatedly. The nonce can also act as an identifier of the delegation token which is useful in revocation.

The current request method collects the client principal’s current request along with the client’s identity, a timestamp and a lifetime. This timestamp identifies the time at which the request has been generated and the lifetime indicates the time period for which the request is valid. The propagation list creation method creates the propagation list which consists of the principals the agent has visited.

The cloning method is used in the creation of a copy of the agent. When this happens, this method generates a different SA identifier. The Creator-OM and the Creator-SMC identifiers of the clone remain the same as the original. The Clone-SMC identifier is now specified which identifies the host where the agent is cloned.

The security checksum code is used to calculate the cryptographic checksums needed in the generation of the various security tags (see below).

The Creator-OM Code contains the agent code required to perform the required tasks. This includes the code required to perform the operations specified in the privilege tokens. For instance, consider an agent whose task is to determine the travel itinerary which involves booking of flights and hotels. In this case, the agent contains code to search for suitable flights and hotels and can include the preferences of the travelling principal. Consider another example where an OM at a bank server creates a simple agent to perform financial transactions such as withdraw and deposit from a certain class of accounts. That is, this agent may contain privilege tokens that allow a client to withdraw up to a specified amounts from a certain class of accounts say during a week.

In addition to the Creator-OM Code, the principal which sends an agent can specify additional methods. We will refer to such code as the Client Code. Considering the financial transactions security agent, the sending principal may be a client of the bank. When the agent requests to perform a certain transaction, the request may be granted or denied depending on the current state of the specific account at the bank server. For instance, the client might have already withdrawn a certain amount earlier in that week and that the current request exceeds the balance. Upon receiving the negative response, the client specified code in the agent may decide to request a smaller amount which can be granted by the target. That is, the client has specified methods that have allowed the agent to analyse the response from the target and to make modified requests. The situation described above is a simple one in that it was the client who had withdrawn some amount earlier and hence it should have been aware of this state change sometime in the past. However, there can be cases where the state changes occur at the target end which may not be visible to the agent.

Rights Token: The Rights Token specifies the set of rights that the target entities can have over the security agents. For instance, this includes the ability of the target to run the agent. In general, this token specifies the rights that different principals can have and the lifetime for which they are valid. So in principle, different SMC principals in different hosts may have different rights over the agent. However these rights are only for guidance or hints and cannot be enforced as the agents are under the control of the target's execution environment. Alternatively, the assumption needs to be made that an SMC is trusted to ensure that the rights specified in this token are enforced.

Request: This field specifies the client principal which is sending the agent, the operation that it is requesting, the time at which the request is being made and the time period for which the request is valid. The privileges associated with the request are determined using both the privilege tokens and the delegation tokens.

Data Store: The SA stores its data information in the data store. For instance, to begin with, this contains the state of the agent in terms of the data variables and the initial values. Subsequently, as the agent is executed, it also stores any results of operations and requests made by the SA. These results may need to be protected for confidentiality, integrity and origin authentication. We describe below in section 4.1.3 a way of achieving this using cryptographic techniques. The data store also contains a propagation list which consists of a list of identities of principals visited by the SA. For instance, this list may just contain the identities of the client host and the target host, if the client has directly sent the agent to the target; as the agent moves from host to host, the identity of each of the hosts will be added to the list. The propagation list is created by the default method propagation list creation mentioned above.

Security-Tag-OM: This security tag is created by the Creator-OM and it contains the hash value of the original SA signed using the private key of the OM. The hash value is calculated on the following: Identifiers, Privilege Tokens, Default Code and Creator-OM Code in the Agent Code, the Rights Token, the initial Data in the Data Store and Delegation Tokens where the delegator is the Creator-OM . When the target receives the SA, the SMC of the target verifies this hash value. In order to do the verification, the target SMC needs the public key of the OM principal that originally signed it. If the target SMC is different from the source SMC that signed it, then the target SMC can use the Identifiers in the SA to get the identity and the public key certificate of the OM that created it and use the source SMC certificate to validate it. In case the source and the target SMCs are in different domains, then SMA is used to get the public key certificate of the required SMC in another domain. In any case, an implementation of this model will require one or more certification authorities, but their role is perfectly standard.

Security-Tag-S : This security tag is generated by each sending client principal and there is one such security tag per sending principal. It contains a checksum containing the hash of any client appended code

and delegation tokens as well as the contents of the request. This security tag is signed using the sending client principal's private key. Even if the set of client appended methods is empty and that there is no delegation, this security tag is provided as it includes the signed request. The Security-Tag-S provides the proof to the target that the SA comes from the sending principal who it claims to be.

4. System Operation

4.1 Security Agent Life Cycle

4.1.1 Creation and Distribution of Security Agent

At the time of creation of an object, the OM generates the privileges in terms of a set of methods. These privileges along with target object identity are passed by the OM to the SMC. The SMC creates the SA by generating the appropriate privileges token, inserting the validity times for the SA, initializing the token type flag and appending the relevant agent code (both the Default Code as well as the Creator OM code) and the Security-Tag-OM. We assume that the OM and the SMC reside within the same host and hence these communications are protected using operating system security features rather than using cryptographic mechanisms.

Once the SA is created, there are two issues involved in their distribution. First issue is concerned with the decision as to whether a client can obtain a SA or not. This depends on the security policy which in our model is determined by the SMC and SMA in conjunction with the requirements from the OM which created the SA. The second issue is concerned with how the SA is distributed to clients. This can be achieved in two ways, either using a push model or a pull model. In the push model, the local SMC can send it directly to a set of clients or the local SMC can make use of the services of the SMA in the distribution of the SA. In particular, this may be useful if the SA is to be distributed to multiple domains. In the pull model, the client principal needs to obtain the SA either by contacting the SMA or the SMC of a host directly.

The transfer of the SA to the client principal needs to be protected for integrity and authentication and in some circumstances for confidentiality. Recall that the SA contains the security tags (Security-Tag-OM and Security-Tag-S) which provide origin authentication and integrity. If the SA needs to be protected for confidentiality, then it can be suitably encrypted using the public key of the client. The freshness of the SA is provided by the use of the nonce and timestamp in the SA included in the signatures in the security tags.

4.1.2 Propagation of Security Agent

In the simple case, the SA is passed by a client principal to a target. The OM at the target uses the SMC to verify the integrity of the SA. The Security-Tag-OM is calculated and checked to determine whether there has been any illegal modification in the original SA. Then the Security-Tag-S is verified in a similar manner using the public key of the sending client. By verifying this tag, the target is able to ensure that the request is currently made by the client and that the responses from the SA conform to those previously prescribed by the client. If all the checks are successful, then the request is evaluated and the results returned to the SA. The SA can use these results to invoke any relevant client specified methods and make further requests. When the transactions are completed, a copy of the results is stored in the Data Store in the SA.

4.1.3 Protection of Data in Data Store

Let us now consider the protection of the results stored in the Data Store. The target SMC produces a signed hash digest of the results along with a timestamp using its private key and this is also stored. This signed hashed digest provides integrity and origin authentication of the results. If confidentiality protection of the results is also required, then this can be achieved by the target SMC generating a secret session key and using this key to encrypt the results. The secret session key itself is then encrypted using the public key of the client. A pure public key based protection can also be used. In either case, with such confidentiality protection, once the results are stored in the Data Store in the SA, the SA as well as other clients are unable to read the results. The SA may return to the principal which sent it, at some later point in time. When this occurs, the principal can verify the signature of the target SMC on the results. This will provide the necessary guarantee to the principal that it was the target which generated these results. If confidentiality of results has been provided, then the results can be retrieved by first obtaining the secret session key and using it to decrypt the results. Alternatively, the SA might decide to reside within the remote host but pass back the results to the sending client. If this were to occur, once again protection of the results for confidentiality, integrity and origin authentication can be achieved as above.

Furthermore, the Propagation List Creation method is activated to modify the propagation list in the Data Store by adding the identity of the target principal to the propagation list.

Let us revisit our example above where a client is sending a security agent and a request to withdraw $10,000 from a cash management account. Assume that the client principal has also included methods to deal with some exception conditions. After verifying the SA and the client in the usual manner, the target creates appropriate runtime to enable the agent to withdraw from the specified cash management account object. Suppose that the current state of the account is such that a maximum of only $5000 can be withdrawn. The SA is informed that the request cannot proceed. The agent code records this response in the Data Store and uses its other specified methods to make a modified request to say withdraw $4000 which is in turn granted by the target. The response from this request is also recorded in the Data Store. As mentioned above, the contents of the results are transferred in a secure manner to the client. By examining the Data Store, the client principal is made aware of the transactions carried out by the SA on its behalf and how and why only $4000 has been withdrawn by the agent.

4.2 Delegation

In general, delegation refers to one entity acting on behalf of another. In a distributed system, this translates into a principal delegating some or all of its privileges to another principal to access some resources [11]. That is, principals may acquire privileges by virtue of co-operation with others. The delegation could be static which is predetermined or may be dynamic which occurs during system operation. By definition, an agent is acting on behalf of some principal. In this model, a security agent is created by an OM and is being sent by a client to act on its behalf. The actions that an agent can request to do are dependent on the OM specified methods and the privileges accorded to it by the OM as well as on client specified methods.

Consider, for instance, an OM principal A which has created a SA which has a Privileges Token PT_A which specifies a set of privileges PR_A. Let us assume that A wishes to delegate some of its privileges to a principal B. This is achieved by setting the token flag to delegate and using the default delegation method in the agent to specify the delegation token parameters such as the delegated set of privileges, their lifetimes along with nonce and timestamp. This can be represented using a Delegation Token as follows:

DT_AB = <A, B, PR_AB, R_A, Timestamp_A, Lifetime_A > Signed A

The Delegation Token includes the identities of the delegator (A) and the delegate (B) and the set of privileges delegated PR_AB (privileges delegated by A to B). The token also has a nonce R_A and can be used as the token identifier. The token has been generated by A at time Timestamp_A and has a finite lifetime Lifetime_A. These delegation parameters are included in the calculation of the signed cryptographic checksum in the security tag signed by the delegator OM principal A.

The principal B is able to read what privileges have been delegated to it and for how long, but is not able to modify them without avoiding detection. Now assume that B sends the SA with the delegation token to the target. By verifying Security-Tag-OM, the target is able to verify whether originally the OM principal A had created the agent and determine what are its privileges by examining the Privilege Token and whether A has actually delegated some of the privileges to B by examining the Delegation Token. Finally, the target can also identify whether it is B which is making the request by examining the Request and verifying the Security-Tag-S. When all these checks are successful, the target can evaluate the request.

Now consider the situation where a principal uses a SA repeatedly in an unauthorized manner. Recall that the delegator inserts a nonce and a timestamp in the delegated token and signs them using its private key. The target SMC can record the nonce, the timestamp as well as the identities of the delegator and the delegate and the SA identifier in a duplication list. This list can then be examined by the target to check whether the delegated SA is being used in an unauthorized manner.

Let us consider a simple scenario whereby an airline server creates an SA that allows clients to make airline ticket reservations. Assume that a client principal wishes to arrange a trip by booking her plane tickets using this security agent via a travel broker. First assume that the SA has been distributed to the client either via push or the pull model considered above. Note that the SA contains methods for booking tickets and has appropriate privilege tokens. The client principal requests the travel broker to make ticket reservations to destination X and requires that the fare be below $500. In this case, the original SA generated by the airline server is now augmented with a delegation token by the client where the delegated privileges include methods for checking the price limit. The client may also attach some additional methods to the SA specifying exception conditions and responses to them; for instance, if the difference between the fare for a direct flight and the fare for a flight with a stopover is less than $50, then the direct flight is to be preferred. After appropriate verification as described earlier, the travel broker sends the delegated SA along with its request to the airline server. The airline server is able to verify that originally the client had been given the privileges and that the client had delegated (some of) its privileges to the travel broker. Suppose now that the agent receives two fares for the plane ticket, one for $490 and the other for $450 though the latter involves a stopover. Now the SA can make the decision and choose the direct flight and make a ticket reservation. The data store in the agent now contains the information on the booking made. This part of the data store containing the information is protected in such a way that only the travel broker is able to read these information. For instance, the data is encrypted using a secret key K1 generated by an airline company and this secret key is encrypted using the public key of the travel broker. Note that in this case we have assumed that the SA makes its decision while it is residing at the airline host. This is acceptable if there is trust between the transacting parties involved. If this is not the case, then the agent needs to return to the sender host (the travel broker) or some suitable neutral host which is trusted (with respect to the transaction) to make its decision. In our model, we envisage one or more hosts within a domain which are trusted by an SA where the agent can migrate to make the decisions. The addresses and locations of these trusted hosts can be hardcoded into the agents or can be looked up by the agents using some form of directory service.

4.3 Revocation

In practice, just as much delegation is useful, revocation is equally necessary [11]. Often a principal may delegate privileges only for a certain period of time. For instance, when a department manager goes away on holiday, a subordinate project manager may be delegated a subset of her privileges; however when the department manager returns, the delegated privileges are to be removed from the project manager. Such simple revocations can be achieved using standard timeout schemes. The security agent structure proposed above has several timestamps and lifetimes associated with it. First, there is the lifetime associated with the agent itself. This lifetime indicates to any principal, in particular to the target which is executing the agent, the period for which the creator of the agent wishes the agent to be valid. The intention is that this lifetime will be taken into account by the SMC at the target in its decision making process. There are also timestamps and lifetimes associated with the privileges and delegation tokens. These lifetimes are on a token basis and they indicate the expiry times of the different privileges. So for instance, one can specify that the privilege to make a plane ticket booking to expire say at the end of the week, Friday 5 pm, 22 May 1998. This applies to both the privilege token specified by the OM as well as to the delegated privileges in the delegated tokens.

Apart from the timeout, there can be other reasons why revocation may be required in practice. For instance, consider the case where the agent itself has become corrupted and has become a rogue agent. In general, dynamic revocation is difficult to achieve because agents can be cloned and distributed to several places throughout the network. The first question to consider is how can we detect that the agent has become corrupted? In our model, the SMC of the host where the agent is being executed can use the security tags to detect some of these cases. For instance, corruption due to changes in the Creator Code and Default Code can be detected by checking Security-Tag-OM. In this context, it is important to note the fundamental trust assumptions on SMC. The SMC is trusted to ensure that a corrupted agent is not allowed to execute within its environment. When a SMC principal detects that an agent is corrupted, it can send a warning message to principals which initially sent the agent to it as well as to other principals to whom it sent the agent to. The agent can be identified using the elements in the Identifiers field. This warning message can be passed from one host to another along the propagation list. Such a method assumes that each SMC in the chain of hosts is trusted not to use the corrupted agent and to pass on the warning message. Of course, a problem with this approach is that although the originator might have trusted the intermediary principals when passing the agent, it might have ceased to trust them subsequently. In fact, it is this change in trust that may be one of the reasons for revocation. The SMC principals need to maintain a revocation list which identifies the revoked SAs using the identifiers, nonces in the privilege and delegated tokens as well as the identities of the delegator and the delegate and the request. Furthermore, it is necessary to assume that the SMC is trusted to ensure that a revoked SA is not able to execute within its environment.

5. Taxonomy and SA

In [12], Kain and Landwehr propose a design taxonomy for capability systems. They raise some 5 questions and the taxonomy is based on answers to these questions. In this section, we consider the answers to these questions based on our SA based authorization model.

1. What happens when a capability is created?

[12] considers two possibilities: (a) No privileges are inserted (b) Privileges are inserted. In our case, privileges are inserted in the Security Agent.

2. What happens to the capabilities describing a object if the security attributes of that object are modified ?

   [12] considers the following three possibilities: (a) Access privileges not changed upon attributes change. (b) Capability flagged for future change upon attribute change. (c) Access privileges updated upon attribute change. In our model, it is neither of the above. The target OM/SMC records this change with respect to the specific SA identifier in its revocation list and new policy interpretation is to decide on the revocation of the old SAs. For instance, when the SA is presented, it can be suitably modified or destroyed in conformance with the security policy interpretation by SMC/SMA.

3. What happens to the capabilities when a capability is copied ?

   [12] considers the following four possibilities: (a) Privileges are not changed. (b) Privileges are further restricted by context rules. (c) Privileges are set to the maximum consistent with the access rules set by the policy. (d) Access privileges guaranteed to be upgraded by the software. In our model, the cloning default method creates a SA copy which has a different SA Identifier and an additional Cloning-SMC Identifier, while the Creator-OM Identifier and the Creator-SMC Identifier remain the same as the original. Note that the SMC where the copying occurs is trusted to ensure this. Access is dependent upon the security policy in the target SMC/SMA with respect to this copied SA.

4. What happens when a capability is prepared for access?

   [12] identifies three cases: (a) Privileges not changed. (b) Privileges restricted by access policy. (c) Privileges set to the maximum consistent with the security policy in force. In our model, when the SA is prepared, privileges are not changed.

5. What happens when the client attempts to access an item?

   [12] identifies the following: (a) No checks are made. (b) The access is checked against the available privileges. (c) The maximum possible privileges are computed/read and the attempted access is checked against these computed rights. In our model, option (b) is chosen. The request is granted or denied depending on the available privileges.

6. Summary

In this paper, we have considered a security agent based approach to authorization in a distributed environment. The structure of a security agent which is used to capture the privileges and a part of security policy in distributed authorization is described. Principals make use of these agents to carry out their requests on remote hosts. Targets verify the authenticity of the security agent and its privileges and use them together with their local security policy to grant or deny the requests. The paper then described the operation of the authorization system based on these security agents and how these agents can be used to support dynamic decision making. Finally, the paper examined how the security agent based model answers the taxonomy questions that arise in the design of capability based secure systems.

References

1. Sun Microsystems, Java : Programming for the Internet, 1996.
2. L.Cardelli, “A Language with Distributed Scope”, Proc. of the 22nd ACM Symposium on Principles of Programming Languages”, pp286-298, 1995.
3. Gary McGraw and Ed.Felten : Java Security : Hostile Applets, Holes and Antidotes, Wiley and Sons, 1997.
4. J.Tardo and L.Valente, “Mobile Agent Security and Telescript”, Proc. IEEE CompCon96, IEEE Computer Security Press, 1996.
5. R. H. Campbell, T. Qian, W. Liao, Z. Liu. “Active Capability: A Unified Security Model for Supporting Mobile, Dynamic and Application Specific Delegation”. White Paper - University of Illinois. 1996. <http://choices.cs.uiuc.edu/liao/home.html>
6. W.M.Farmer, J.D.Guttman and V.Swarup, “Security for Mobile Agents: Issues and Requirements”, Proc.19th National Information System Security Conference, NISSC’96, 1996.
7. W.Farmer, J.Guttman and V.Swarup, “Security for Mobile Agents: Authentication and State Appraisal”, Proc. of the European Symposium on Research in Computer Security, 1996.
8. V.Varadharajan and Y.Mu, “On the Design of a Security Model for Mobile Agents”, Submitted for Publication.
9. P.A.Karger, “Improving Security and Performance for Capability Systems”, PhD Thesis, Cambridge University, 1988
10. L.Gong, “A Secure Identity-based Capability System”, Proc. of the IEEE Symposium on Security and Privacy, pp 56-66, 1989.
11. V. Varadharajan, P. Allen, S. Black. “An Analysis of Proxy Problem in Distributed Systems”. Proceedings of the Symposium on Security and Privacy, pp 255-275. IEEE, 1991.
12. R.Y.Kain and C.E.Landwehr, “On Access Checking in Capability-based Systems”, IEEE Transactions on Software Engg., Vol.SE13, No.2, Feb.1987.