The following paper was originally published in the
Proceedings of the Third USENIX Conference on Object-Oriented Technologies and Systems
Portland, Oregon, June 1997

Resource Access Control for an Internet User Agent

Nataraj Nagaratnam, Dept. of ECE
Syracuse University
Steven B. Byrne, JavaSoft, Inc.
Sun Microsystems

For more information about USENIX Association contact:

1. Phone: 510 528-8649
2. FAX: 510 548-5738
3. Email: office@usenix.org
4. WWW URL: http://www.usenix.org

Resource Access Control for an Internet User Agent

Nataraj Nagaratnam, *Dept. of ECE, Syracuse University*
Steven B. Byrne, *JavaSoft, Inc., Sun Microsystems*
email: { *nataraj@cat.syr.edu, sbb@eng.sun.com*}

Abstract

The rapid increase in the Internet's connectivity has lead to proportional increase in the development of Web-based applications. Usage of downloadable content has proved effective in a number of emerging applications including electronic commerce, software components on-demand, and collaborative systems. In all these cases, Internet user agents (like browsers, tuners) are widely used by the clients to utilize and execute such downloadable content. With this new technology of using downloadable content comes the problem of the downloaded content obtaining unauthorized access to the client's resources. In effect, granting a hostile remote principal the requested access to client's resources may lead to undesirable consequences. Hence it is important for the browsers to provide a framework such that the user can fine tune his system according to his trust relationship with the content authors. Currently available systems either do not allow the downloaded content to access any of the local resources or allows all the contents to have the same privileges. In this paper, we present the design and implementation of a model that provides resource access control of a finer granularity for an user agent. Using our model, the client will be able to selectively grant access to resources based on a trust relationship with the principal, who has certified the authenticity of the contents.

1 Introduction

The ever-expanding nature of the Internet and the World Wide Web poses new problems such as scalability, standard naming scheme, and security. Nowadays it is becoming increasingly common to download some active content over the untrusted Internet and execute it on a client machine. This downloadable content can be Java [1] applets, Castanet [3] channel's contents or component objects like JavaBeans [2], and other executables. With the wide acceptance of object-oriented technology in every aspect of engineering, it is also common to envision all such content on the Web to be objects, accepting messages and providing the necessary services. Designing a scheme to protect client machines from hostile applets and components has become a necessity. Such a scheme should also provide the user the ability to selectively allow *trusted* contents to be downloaded and executed.

Protecting the client machine from hostile applets can be considered equivalent to providing a controlled access to a (client) system's resources. Devising such scheme calls for defining whom the client trusts, to identify the source of such downloadable content, verifying that the principal certifying the content (identity) is the same as it claims to be. This scheme should be flexible enough for the user to customize it to his security needs. The flexibility is now a requirement given the classifications of the network as Internet, corporate Intranet and Extranet (a domain consisting of an Intranet and multiple trusted client sub-domains). As more Intranet applications are developed it is common to assume that all such applications and downloadable content originating within the Intranet domain can be equally trusted. In this paper, we present the design and implementation of such a scheme for usage in user agents like browsers such that the restriction of access to resources in the name of high-security does not prohibit the users from using downloadable contents such as applets.

1.1 Motivation

The Internet has proved to be an effective data distribution medium, especially for software. The concept of downloadable content, where the software component (or the software itself) can be downloaded on-demand from the provider's (server) machine and executed on client machine, adds a flavor to this medium. From the point of view of software distributors, a new version of the software can just be installed on a server from where a client can download it or, in the case of Castanet [3], the tuner will automatically download updates in channels from transmitters. At the same time, a client can be assured of obtaining the latest version. The important aspect that affects such a developer-client

relationship over such an open medium like Internet is the varying trust relationships between any such pair. One way to protect client's resources is to prevent any (and all) downloadable content from accessing any of the client's resources. This is exactly the default policy enforced by the Java runtime to protect client machine from being attacked by all applets, the so called "sandbox security model." The user-agents (like browsers, tuners, etc.) incharge of downloading the content over the net and executing it on the client machine, widely adopt this default policy and hence restrict any applet from accessing the client's resources. Though this provides a solution for preventing hostile applets from attacking a client machine, its inflexibility prohibits the client to grant access to *trusted* applets. Hence, there is a high demand for a flexible mechanism for user agents (*browsers*) to serve the entire spectrum of trust relationship, varying from completely trusted Intranet contents to highly-untrusted Internet contents. Such a demand has been the motivation behind the modeling of the flexible system described in this paper.

1.2 Infrastructure

Our model is general enough to be applied to any environment of user agents and downloadable contents. Our implementation is tailored to the Java environment, as it is becoming the *de facto* standard for deploying Web-based applications. The basic infrastructure over which our system is built is the security framework of the JavaSoft's JDK1.1 release. We have taken advantage of digitally signed applets (which establishes an identity to base our trust on), the public-key key cryptography based mechanism for such exchange of contents, and the ACL (Access Control List) framework to associate a list of trusted identities with any object the client is trying to protect and Java's Sandbox security model.

The rest of the paper is organized as follows. Section 2 describes related work. An overview of the Java's sandbox security model is provided in Section 3. Definition of the components in our model is provided in Section 4. The specification details (identities, groups) required in our model is covered in Section 5. Description of our access control model is in section 6 followed by details of our trust policy over which we base our decision and the way in which we specify such policy, in section 7. Section 8 concludes this paper.

2 Related Work

Netscape Navigator 3.01 prohibits any applet downloaded over a network from accessing local files. Only those applets that reside on client machine which are accessible through the CLASSPATH (i.e. the content server is the client itself) can access the local files. The applets loaded over a network can reestablish network connections to only the site from which they were downloaded. Any other network connection to other sites is prohibited. This inflexible mechanism does not provide a way for users to fine tune their browser to allow trusted applets to access the local resources.

The HotJava1.0prebeta1 web browser provides a little flexibility to users in controlling accesses to local files. HotJava has encapsulated many parameters as *properties* ($< name, value >$ pairs) that can be configured by the user. These properties take effect when the browser is first invoked and changes to certain properties will be dynamically absorbed. Among those, properties of interest are the *acl.read* and *acl.write*. The value these properties take is a list of file (or directory) names. Specifying file (or directory) names indicate to the system that any applet run by the browser can read the files (or directories) listed in the *acl.read* and write to those listed under *acl.write* property. This means, either all applets read/write to a file/directory or none of the applets can.

Safe Tcl [4] consists of two interpreters: trusted interpreter and untrusted interpreter. The trusted interpreter provides access to the client machine's resources whereas the access is prohibited in the untrusted interpreter. The idea behind the SafeTcl is to run trusted code in trusted interpreter and untrusted one in the other. It lacks authentication and so all content is to be assumed to have been downloaded from untrusted sources.

The Telescript engine [5] uses *credentials* and *permits* for access control. The credentials establishes the identity of the principal responsible for the creation of the downloadable content. The permit is like a capability which grants access rights to other (including downloading client) principal's resources whereas the client can deny the right granted by the permit. Also permits do not have the scope of resource restrictions that we provide.

Cryptolopes (cryptographic envelopes) [6] provide a mechanism for protecting the content from hostile hosts. The client negotiates to access the content with the server. It helps providing security to the content whereas our system protects the client from the content.

Abadi et al. [7] present a calculus for access control from logical perspective. They have provided a logical language for access control lists and theories for making decisions on granting access requests. They have dealt with roles, by treating roles as a

composite principal which acts "as" the role (usually with reduced rights). In our system, we have not dealt with roles at this point. We plan to work on these extensions in future.

Jaeger et al. [10] describe an architecture for access control of downloaded content. Their architecture allows access of resources by downloaded content in a controlled manner. They map a remote principal to a principal group and determine the access rights. The four categories of principals they consider are: downloading principals, remote principals, applications developers and system administrator. Individual principals are aggregated into a principal group if they have the same rights. Such group rights are used to determine the rights of each individual principal. In our design, we define access to a resource using an ACL. This ACL is a set of $< principal, permissions >$ pairs. Thus in our method, same ACL can be used to define access rights for other resources. This increases the flexibility and reusability of ACL definitions.

3 Java Security Model

The implementation of our access control model is for a Java-enabled user agent like HotJava. Understanding the underlying security model is necessary to successfully augment advanced security features. As we are concerned with the security of a client system's resources against a downloaded content (applets), we will describe the Java's sandbox security model on which our implementation is based.

3.1 Security Reference Model Specifications

The Java Security Reference Model [13] defines an applet to be an executable Java program that is downloaded from the server. Also, applet loading and security is under the control of the application. Hence, defining our security policy for the browser is needed to provide necessary security enhancements as far as downloaded applets are concerned. The model defines a set of security interactions between Java components namely applet, application, Java virtual machine (JVM), client-server platforms and the server itself. Among those interactions, the *Applet Access Device Attempt(AADA)* is important to our model. According to AADA, an applet may attempt to call a method within the application (browser), such as an access to the local file system, or display. The application's Security Manager policy mediates the requested access. The invariant in the AADA is that the application always calls the Security Manager object to see if the requested access is permitted. This model (the sandbox model) in which access to resources goes through a security manager object of the application, helps to run untrusted code in a trusted environment and still ensure that the applet cannot damage the local machine.

3.2 Sandbox Security Model

Users can import and run applets from the Web or an intranet without damaging the client machine. Such an applet's actions are restricted to the "sandbox", which is an area dedicated by the browser to that applet. The applet cannot access any resources beyond the sandbox. This helps users ro run any (even untrusted) code and still ensure protection of their resources from attacks. The scope of such a sandbox is left to be defined by the browser. Our work presents a model to expand this sandbox to an extent user desires i.e., user should be able to define the scope of this sandbox depending upon the remote principals certifying downloadable contents.

According to the sandbox model, a security manager object serves as an access-approving authority within the application. Any attempts to access to resources, go through the security manager which in turn grants or denies access. The security manager is an object that is a subclass of the class *SecurityManager*. When an access is denied, the security manager throws a security exception.

Currently, the existing browsers don't have the flexibility to selectively allow access to selected applets. Our work fills this gap by defining a model to specify trust, resources that can be accessed on a per-applet basis, and necessary extension of the SecurityManager class to achieve the desired flexibility.

3.3 Establishing Trust

A mechanism to authenticate applets is necessary to define trust based on where the applet comes from. Digital signatures [11, 12] based on public-key cryptosystems come to the rescue. If an applet author can sign his applet, then the client can verify his signature and take necessary action: either deny or allow resource access requests. The JDK provides necessary framework for signing class files. To sign an applet, the author can bundle all Java code (class files) into a single Java archive file called a JAR file. Based on his private key and the contents of the JAR file, the author generates a digital signature block. On the client side, the security manager can resolve authentication issues by using the digital signature mechanism. Once the code is authenticated, then it can take the right decision based on user's access control specification.

Java provides the sandbox security model along with the mechanism for authentication using digital signatures. Our design is based on these available facilities in the Java security framework. In the next subsequent sections we will describe how we use this framework to help user specify trust, resource access control and how these specifications are interpreted by our security manager in effectively controlling access to the resources.

4 Model Components

Protecting client machines from malicious downloadable content is the objective of our system. Before we model a system that would achieve this, we need to define what we are trying to protect (resources) and from whom (principals certifying the contents). In this section we will define the granularity of such resources and varied categories of principals.

4.1 Resources

The resources on the client machine that need to be protected includes from files, directories on local disk, network connections, CPU usage, memory usage and access to the display. The resources can also be extended to non-physical components like remote objects, components like Java beans, etc. In our implementation, we will illustrate the protection of files and restricting network connections from Java applets. Effective application of remote objects [9, 8] may involve method invocations by downloadable content on objects residing in a client machine. In such cases, the user might be interested in protecting the object from being invoked or accessed by other hostile objects/applets.

4.2 Principals

When the issue of access control is raised, along with that comes the question of whom to trust. Implicitly there is an association of contents to some *principal* responsible for (creation or certification by digitally signing) that content. Such a principal can be another user, a company, a host, or a group of such entities. In an open distributed environment like the Internet, it is not impossible to impersonate other principals. This will lead to the user giving access to his resources to a principal, who is actually not whom he claims to be. Strong authentication is necessary in such an environment. The basic requirement for authentication is to define who principals are and how they can be authenticated.

In our model, principals can be individual users, companies, or hosts. With each of these principals, there should be a $< public, private >$ key pair associated. Using public key crypto techniques we can authenticate the principal. The notion of a principal can further be extended to groups of such principals. Assuming the existence of a name space to resolve identities of these principals, we can confirm their identities. Those principals can establish their identity along with the content they have developed, by signing them using digital signatures. We will describe the syntactic specification of resources and principals, as in our model, in the next section.

5 Specification of Principals

A standard format is required to specify principals and resources in any of the configuration files. Resources need to be specified only when its corresponding access control list (ACL) is configured. An ACL is a list of $< principal, permissions >$ pairs. Hence, principals need to be specified during formation of ACLs. Principals in an ACL can be individual users, hosts or group of these. Individual principals can be specified using their associated names (eg., *Nataraj, syrResearch, SyrUniv* or *diamondsTeam* for identities and *ratnam.cat.syr.edu or cat.syr.edu* for host name or domain name specification). Identity names are unique (we assume the existence of a global name space) and can be specified as such. Hence, a identity name can be name of individual identity like *Nataraj* or an identity of a team like *syrResearch* or a company or a body of companies and so on. In these cases, even though an identity might be a collection of other individual entities, it by itself is considered an identity. This notion of an entity representing a set of identities is different from *groups* of identities. A group is a set of identities sharing some common property. Each of those identities are called members of that group. A member can represent a group by signing **for** the group. But in the above case like *researchTeam*, though its a set of individual identities, it is a principal by itself having its own key pair. For such principals, other authorized principals can sign **as** the group.

Groups are sets of principals. Principals can be identities, hosts or other groups. Groups are specified separately in our system. They are specified in the format

```
<groupName>=<identityName>[,<identityName>]*
```

hence, following is an example of valid group specification:

```
syrResearch=Nataraj,Doug,Paul
diamondsTeam=Gary,Doug,Nataraj
syrHosts=cat.syr.edu,ece.syr.edu
catHosts=ratnam.cat.syr.edu,lynx.cat.syr.edu
```

The underlying *ACLParser* object (an instance of the sun.hotjava.security.ACLParser, responsible for parsing the ACLs specified in the predetermined format) parses this information and populates the *ACLManager* (an instance of the sun.hotjava.security.ACLManager, which maintains ACLs, policy database and the decision making authority for granting access). This is stored effectively like a lookup table as illustrated in Table 1. Any of the specified principals in an ACL should be registered in the user's identity database. An identity DB is created using the *javakey* utility of the JDK1.1. This utility helps specify the identity and how it is trusted and so on. This utility manages a database of entities (people, companies, etc.) and their keys (public and private) and certificates. This tool also generates signatures for JAR files and verifies those signatures [14]. It can be used by a client to declare whether or not it trusts certain entities.

The principals can also be specified using regular expressions. Hence, the following is also a valid specification.

```
allSyrHosts=*.syr.edu
```

Using a combination of regular expressions and other groups, new groups can be formed with ease. This makes the specification of principal groups easier.

6 ACL-based Model

Our model is based on associating access control list (ACL) with resources. We create named ACLs and associate them with resources. An ACL is associated with each resource to guard it and ACL itself is independent of the resource it guards. So a $< resource, ACL >$ pair means that the principals have the corresponding permissions on the resource. An ACL can be (re)used to guard more than one resource. The relationship between resources and ACLs are depicted in Figure 1. In this design, the key is the resource name. When a principal tries to access a resource, the system consults the configuration and obtains the ACL associated with the resource. It then checks the ACL to see if the principal under consideration has the required permission. If so, the system allows the principal to access the resource. Otherwise, it denies the attempted access.

In this system, the Java VM traps any access to the system's resources. The request is funneled through the security manager. The security manager is responsible for checking if the access is authorized by the user. The $< resource, ACL >$ association is formed during the start up of the application (which is executing downloaded content) and hence, its basically an associative lookup for ACL during the runtime. The user is also given the flexibility to add new $< resource, ACL >$ entries at runtime. The configuration is then dynamically updated and so is the database.

7 Trust Policy

In this section, we will describe the semantics of the decision to grant/deny access based on the specification of ACLs and policy. We will first understand the logic behind decisions taken by the security manager. We will then provide the format specification of ACLs and policies.

7.1 Access Granting Policy

Each user maintains a local security database containing the trust policy information. It consists of

- a database of principals (and keys) created using the *javakey* utility
- a specification of groups, formed by a set of principals
- a set of access control lists, containing $< principal, permissions >$ pairs
- a list of $< resource, ACL >$ (policy) pairs defining the trust relationship

The specification of the ACLs and associating resources with ACLs together form the trust policy database. When a request for an access to a resource is submitted, the application consults (through our enhanced security manager) this database to make a decision based on the ACL guarding the resource.

The Figure 2 depicts the decision flow in granting permission for a downloaded content to access a resource. The default policy is to deny any access unless the user explicitly grants access. If the identity is given access or denied access through explicit specification by the author, then the decision is based on that specification only. If there is no explicit individual specification of permission, the same check is carried out for all of the groups that the identity is a member of. Even if one of the groups is explicitly denied access, then the principal is denied access by the security manager. If all the groups are explicitly given access, then the principal is granted the request to access the resource. If no explicit permission is specified either as an individual identity or as a member of any of the group, then by default the access is denied. The browser then dynamically queries the user if he would like to allow the principal who has signed the applet to access the resource. Depending on user's input, the

GroupName	Principals	Principal Type
syrResearch	Nataraj, Doug, Paul	Identities
diamondsTeam	Gary,Doug,Nataraj	Identities
syrHosts	cat.syr.edu, ece.syr.edu	Hosts
catHosts	ratnam.cat.syr.edu,lynx.cat.syr.edu	Hosts

Table 1: A Group Table

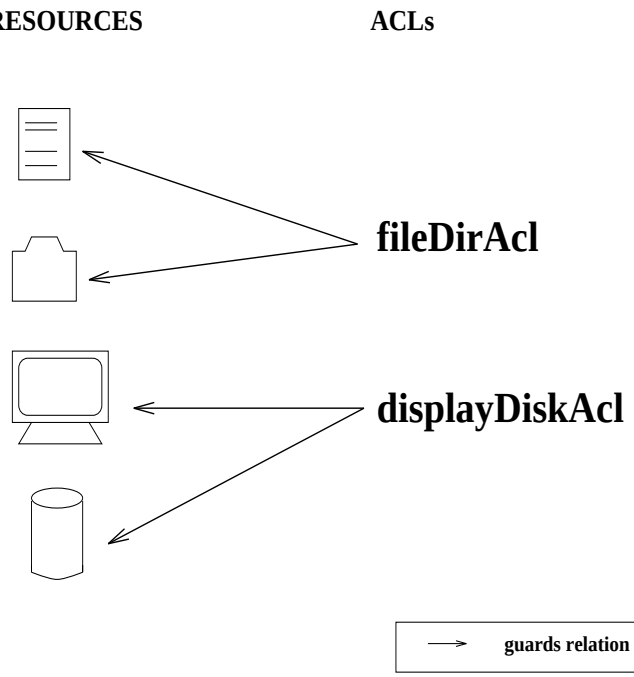

Figure 1: Sample Relation between Resources and ACLs

database as well as the runtime are dynamically updated. It is also possible for the user to specify negative permissions. This flexibility will be useful when specifying exceptions to group access permissions. For example, user might want to give access to all the members of a group except one member. In such a case, instead of forming a new group (a subgroup of original group), it is easier to grant permissions to the group and specify the member to be an exception. All combinations of principals can thus be accommodated in permission specification using the flexible format. The specification format for ACL and policy are given below.

7.2 ACL and Policy: Specification Format

An ACL relates principal to permissions. We use the ACL framework in JDK1.1 [14]. The principal is the key field in the ACL database. Given a principal name, one can obtain all the permissions associated with the principal. The format is as follows:

```
[+/-]{User|Group}.{Identity|Host}.
  <PrincipalName>=<setOfPermissions>
```

where,
the first field (optional) specifies whether the ACL specifies a granted permission or an exception. A − in that field indicates that it is an exception, i.e. the principal of given *PrincipalName* is explicitly denied the specified *setOfPermissions*. A + in the first field (or even if nothing is specified in that optional field) indicates that the principal is given the specified set of permissions. The key word *User*, in the second field, specifies that the principal name associated in this ACL is an individual principal whereas the keyword *Group* specifies that the principal name is actually the name of a group of principals. This resolves the *PrincipalName* specified in the fourth field to be either an individual principal or a group. The third field indicates that the specified principal

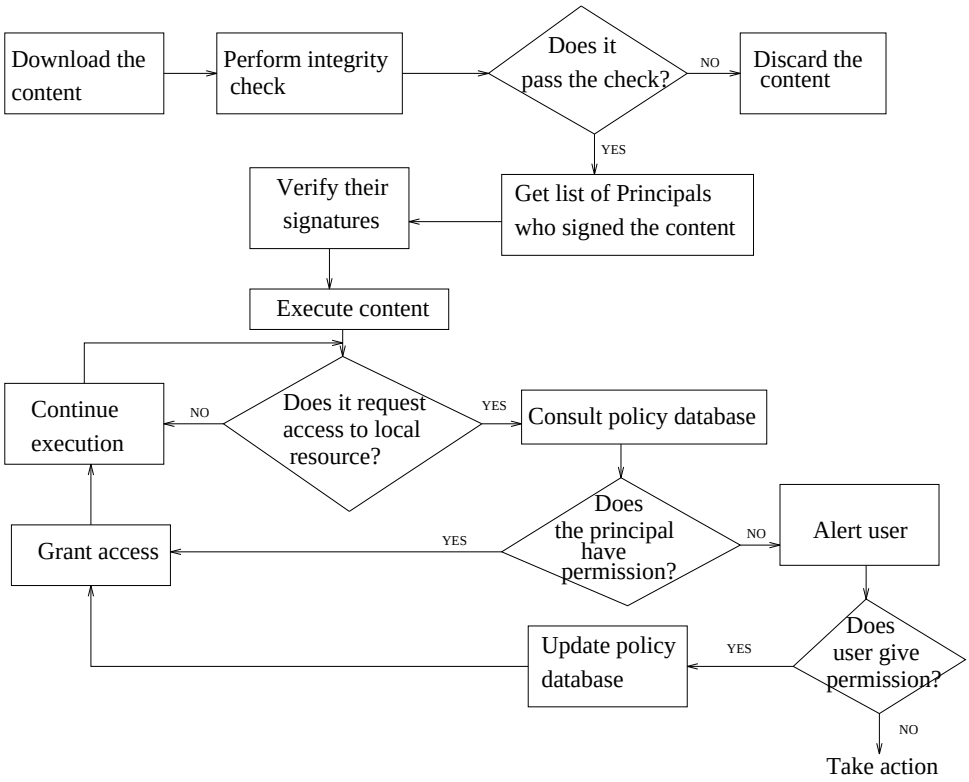

Figure 2: Decision flow for access control

is an Identity (name of a person, company, team, etc) or a Host (hostname, domainname, etc). The *setOfPermissions* is a list of permissions separated by a ','. Following example is a valid ACL specification.

```
+User.Identity.SyrUniv=FileRead, FileWrite
+Group.Host.catHosts=FileRead, FileWrite
-User.Host.ratnam.cat.syr.edu=FileWrite
```

In the above ACL (say it is named *acl1*), the principal SyrUniv has the FileRead and FileWrite permissions granted to it. All the host principals in syrHosts have the permission to FileRead and FileWrite except for the individual host cat.syr.edu (presumably a member of syrHosts group), which has been denied FileWrite permission. So effectively, the host cat.syr.edu has the permission FileRead through its membership in the syrHosts group but does not have the permission FileWrite even though all the other group members have the permission. For instance let the policy specification contain

```
/hostA/users/nataraj/javaWork/*=acl1
```

In this case, the ACL named acl1 guards the directory /hostA/users/nataraj/javaWork. So contents certified by SyrUniv can read and write to that directory. Also all the syrHosts can read and write to that directory with the exception of the host, ratnam.cat.syr.edu which cannot write to it.

8 Implementation Status

We have implemented our model for the HotJava browser. The browser allows selective access control of resources by the user. The user interface has been designed so that an end-user need not deal with the ACL or policy specification format details. The user can use the interface to specify the group and access configuration. The internal format and storage details are taken care of by our system.

We have built a security manager class BrowserSecurityManager which subclasses sun.applet.AppletSecurity. An instance of the BrowserSecurityManager is created when the browser is initialized. This security manager object has an ACLManager object which acts as an interface to the ACLs and the policy database. Whenever an access is attempted, the sandbox security model funnels the request to our security manager object. This object consults the ACLManager object to see if the remote principal (responsible for the applet which originates the access request) has the necessary permission(s) to perform the operation. On re-

Principals	Principal Type	Permissions	Permission Type
SyrUniv	Identity	FileRead, FileWrite	Grant
syrHosts	Hosts	FileRead, FileWrite	Grant
ratnam.cat.syr.edu	Hosts	FileWrite	Deny

Table 2: An ACL declaration

ceiving a *grant* message from the ACLManager object, the access is permitted. A security exception is thrown, otherwise.

The group and access configuration of the client system are read during initialization of the ACLManager object. With the assistance of ACLParser, the group specification and access specification are read and the ACLManager object is populated with this information.

We have provided implementations of the java.security.acl.Permission, java.security.acl.AclEntry and java.security.acl.Acl interfaces to serve our purpose. The BrowserAclImpl is responsible for both adding ACL entries (after reading the policy database) and for checking permissions during runtime. The class relationship is depicted in Figure 3 (for clarity, we have omitted method names from the class diagram).

The interface to the policy database is provided through the ACLManager object. In turn, the ACLParser object has the permission to read the policy database. The BrowserAclImpl object can add new entries at runtime (if the user wishes to add a trusted principal) and can update the policy database. Thus all these objects co-operate to control an access to client's resources. Currently, we have implemented our system to provide access control for the HotJava browser based on who has certified (signed) the applet and/or the source host of the applet (the host from where the applet is downloaded). In future, we plan to extend the communication through the Secure Socket Layer (SSL) and to provide flexible delegation mechanism, once the necessary delegation framework is in place.

9 Conclusion

We have presented a model to control the access to resources in an open distributed environment like the Internet. This model has been designed to provide advanced security features to user agents like browsers enabling them to selectively trust and grant access permissions to principals in such an open environment. This flexibility is critical not only for development of applications for the open untrusted Internet but for any trusted Intranet. The need for such a flexible security framework still exists. The public key cryptography techniques have been put to effective use in establishing authenticity over the network. Using our model, we have implemented a solution for the browsers to download contents over untrusted network and execute them in a trusted environment without damaging the client machines.

We have modeled our system with the flexibility of dealing with any kind of resource. Especially this will be useful with the current state of object-oriented technology where distributed objects cooperate to achieve their goal. Systems based on intelligent agents or distributed objects providing different services are being built. These objects may communicate either through remote method invocations like the RMI package [8] provides or by the another remote object mechanism proposed by us [9]. Given the Web's heterogeneous nature and Java's suitable positioning as an object-oriented platform for the Internet, extending our model to distributed objects can be easily done.

In a distributed environment, rights of a principal may be delegated to other principals. Thus the access control model needs to be extended to accommodate such delegated rights. As more distributed object-based systems are evolving and with speed in which Web-based applications are being deployed, the need for such a framework is necessary to provide a secure environment for remote accesses. In the future, we plan to develop a practical delegation model for secure distributed computation over the Internet.

References

[1] K. Arnold, and J. Gosling , "The Java Programming Language," *Addison-Wesley,* 1996

[2] "Java Beans: A Component Architecture for Java," *Online document at http://www.javasoft.com/beans,* December '96.

[3] "Castanet Whitepaper," *Online document at http://www.marimba.com,* October 1996.

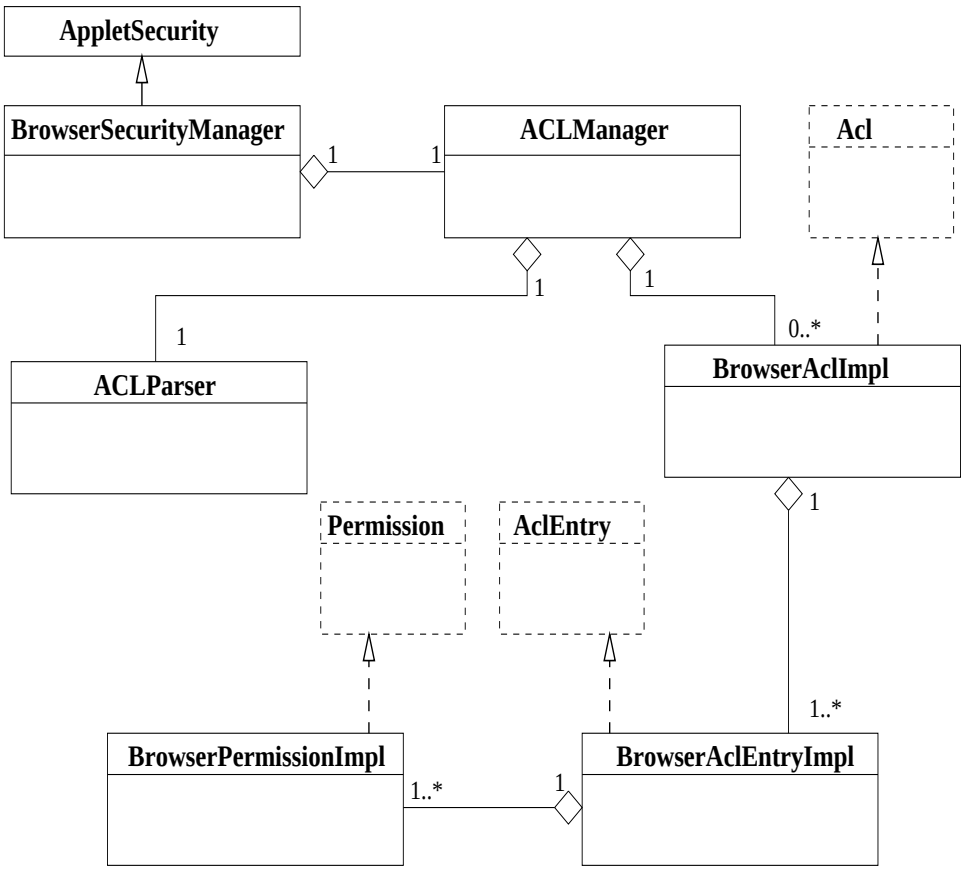

Figure 3: HotJava Resource Access Control Class Diagram

[4] J. Levy, and J. Ousterhout, "Safe Tcl: A Toolbox for Constructing Electronic Meeting Places," *First USENIX workshop on Electronic Commerce,* 1995.

[5] J. E. White, "Telescript Language Reference Manual," *General Magic Inc.,* October 1995.

[6] "Cryptolope Container Technology: A White Paper," *online documentation at http://www.cryptolope.ibm.com,* September 1996.

[7] M. Abadi, M. Burrows, and B. Lampson, "A Calculus for Access Control in Distributed Systems," *ACM Transactions on Programming Languages and Systems,* Vol 15 September 1993, pp706-734.

[8] A. Wollrath, R. Riggs, and J. Waldo, "A Distributed Object Model for the Java System," *Proceedings of the USENIX Conference on Object-Oriented Technology and Systems '96,* July 1996.

[9] N. Nagaratnam, A. Srinivasan, and D. Lea, "Remote Objects in Java," *Proceedings of the IASTED Intl. Conference on Networks,* January 1996.

[10] T. Jaeger, A. Rubin, and A. Prakash, "Building Systems That Flexibly Control Downloaded Executable Content", *Proceedings of the Sixth USENIX Security Symposium,* July 1996.

[11] B. Schneier, "Applied Cryptography," *Wiley and Sons,* 1994.

[12] R. Rivest, A. Shamir, and L. Adleman, "On Digital Signatures and Crypto Systems," *Communications of the ACM,* 1978.

[13] M. Erdos, B. Hartmann and M. Mueller, "Security Reference Model for the Java Developer's Kit 1.0.2,", *On-line document,* http://java.sun.com, 1997.

[14] Sun Microsystems, "Security in JDK1.1," *Java Documentation,* http://java.sun.com, 1997