Entegrity® AssureAccess®

Java™ Standards

This paper provides an overview of current Java™ standards relating to authorization and their limitations, and discusses some new approaches to Java authorization.

Entegrity Solutions® Whitepaper

ENTEGRITY Solutions®

Table of Contents

Executive Summary

Entegrity® AssureAccess® is an industry leading access management product, which provides a rich set of authorization services in a variety of application server environments. Entegrity is strongly committed to standards and has always supported relevant industry standards in all of its products. AssureAccess is implemented entirely in Java and targets Java server environments, such as Servlets, Java Server Pages and Enterprise Java Beans (EJB) for delivery of its services. Yet AssureAccess, like its competitors, does not provide implementations of major Java authorization standards, such as the Java Authentication and Authorization Service (JAAS) or the J2SE permissions model.

A complete explanation of why this is the case requires a detailed analysis of the relevant standards. This paper provides just such a point-by-point description of the existing Java Authorization standards. In summary the reasons boil down to two, one or both of which apply in each case.

1. The standard does not specify in sufficient detail how it can be used in a server. There is no explicit provision for the identity to reflect the user making a remote request.
2. The standard can only be implemented by a vendor providing a complete application server, not by an independent security provider.

Java provides a rich set of security standards. These include cryptography services, authentication methods and authorization based on the origin of the code being executed. This paper concentrates on authorization services that are based on user identity or network connection characteristics, as these are the services relevant to access management. Even with this narrowed focus, there are four major models to discuss: J2SE, JAAS, Servlet and EJB. The discussion is simplified by the fact that these models can be presented cumulatively. J2SE defines a based set of classes and methods, which the other three models are consistent with, but specify additional details.

The cornerstone of the J2SE authorization model is the permission class. A permission object is a way of recording the right to perform some action in a Java environment. By extending the permission class in various ways it is possible to obtain considerable flexibility in the semantics of permissions and the relationship between more general and more specific capabilities. However, the J2SE model does not specify how the identity of a remote user is represented, in order to make access decisions.

JAAS defines a model of client-side authentication that allows a user to authenticate and retain credentials that apply to multiple environments. A subject class is defined to hold the principals and credentials associated with the various authentications. In order to access different applications, the appropriate principal is presented. Unfortunately, there is no way to use this scheme as a server-side pluggable authentication system without providing the entire application server or using a scheme defined by a particular application server vendor. The authorization component of JAAS is now identical to J2SE and suffers the same limitations.

The J2EE environment consists of the Servlet (and JSB) environment and the EJB environment. Both use Deployment Descriptors to define the semantics of permissions. In Servlet, permissions are mapped to URLs. In EJB, permissions are mapped to Bean methods. Servlet also supports controls based on the security properties of the network connection and the use of a request filter, which can be used for various purposes, including security. J2EE does

encompass the notion if identifying the remote user's identity, but the means of making that identity available for access control is left to the application server vendor. AssureAccess 2.0 makes use of the Servlet filter to apply policy controls in a manner similar to the filters in Web servers.

Recognizing the limitations of the various Java authorization models, Entegrity worked through the Java Community Process, beginning in 2000 to develop a standard that would allow access management vendors to plug in to the Java environment. The intention was to preserve the strengths of the Java models and allow competition based on the unique characteristics of distinct security products. This effort led to JSR 115, which represents the combined efforts of application server vendors, access management vendors, systems integrators and Sun™. However, JSR 115 will not be available in products for some time and it may prove too complex for customers to use.

Entegrity has therefore incorporated into AssureAccess, two further methods of performing access control in a Java server environment in addition to using the EAA API directly. The first is a library of JSP Tags, which both simplify development and reduce vendor lock-in a JSP environment. The second is the revolutionary Universal Java Plug-in, based on Tangosol technology, which provides access to the full power of the EAA dynamic policy capabilities, with no programming at all. Once an administrator has identified the methods to be protected, Tangosol automatically inserts the required controls. Because no programming is required, time and money are saved, which customers can better apply to application functionality. The elimination of programming also means that customers are not required to invest in a vendor-specific API.

Introduction

This paper discusses methods of performing authorization on Java, with particular emphasis on authorization within servers of requests made by remote users. This is the type of authorization provided by Entegrity AssureAccess and other access management products. In distributed environments, it is difficult to protect resources on client systems. Therefore the general practice is to concentrate them in servers. Authorization in non-server contexts tends to be useful in two cases: applets and imbedded systems where the OS does not provide access control.

Java provides a large number of security related services in its standard APIs. One such set of classes and methods is called the Java Cryptography Extension (JCE). It provides low-level cryptographic functions that can be used in a variety of ways. These capabilities are not discussed in this paper.

Java also supports a variety of popular authentication methods. These include username/password, PKI and Kerberos. Facilities are also provided for extension to cover other methods. This paper only discusses the aspects of authentication that directly relate to authorization.

Java provides a rich set of authorization APIs. While they share certain common elements, there are at least four distinct variants. Each applies in different environments. Unfortunately each also has limitations to it use.

The initial Java security model only distinguished between local (installed) and remote (downloaded) classes. Remote methods were very limited in the functions they could perform. This model was later extended in two dimensions. First, remote code was distinguished by where it was downloaded from and who, if anyone, had signed it. Second, a much finer degree of control was introduced, depending on the code source. Specific operations could be allowed or prohibited.

The ability to make access control decisions based on the properties of the local or remote user was also introduced. The current models allow decisions to be made based on both code-source and user, but in practice usually only one or the other is used. With Applets, the user is trusted and the system is being protected from hostile code. In installed-code environments, the code is trusted and the goal typically is to differentiate the capabilities of different users.

Since the focus of this paper is server-based authorization, the facilities based on code source will not be discussed. Note that the remaining capabilities are based exclusively on user permissions that have been granted and do not consider factors such as date/time, network location, or application parameters. In this respect they are essentially equivalent to Access Control Lists (ACLs) in functionality. The Servlet environment also specifies access control based on the security properties of the communications channel, however this feature is not present in the other models.

Most of the complexity in the Java authorization models comes from two general requirements. First, it is necessary to allow application components to be deployed in different environments in conjunction with other components that may have been developed independently. This requirement is met by introducing various levels of indirection into the models. For example, EJB applications use role references, which can be mapped at deployment time to specific roles,

thus avoiding potential name conflicts. The use of permissions to map between identities and actual resources is another example.

The other general requirement is the need to allow application developers to protect resources that may have different properties from those provided in the standard libraries. The mechanism for doing this is to create new Permission classes.

This paper begins by describing the current authorization models. For each, the key classes and methods are described. Then the limitations to the specification are described. The paper then describes the approach being taken by JSR115 to eliminate some of the restrictions of the current specifications. Finally, the paper describes two approaches to the problem taken in AssureAccess. While these approaches are not a part of any Java specification, they do enable the full power of the AssureAccess dynamic policy model while not locking the customer into Entegrity or any other vendor.

Java Authorization

The Java authorization standards are not written with the idea that there is an access management provider distinct from the container provider. In a number of areas, the standards only specify the APIs visible to applications and their semantics, but key aspects of the implementation are left up to the application server vendor who implements the container. This means an Access Management vendor, like Entegrity who does not, as a matter of business strategy, wish to provide an application server faces several obstacles in implementing these specifications.

First, the security vendor is dependant on the application vendor to implement the necessary mechanisms and to make the information about how they work available. Even if the application server vendors are willing to do this, most likely each will do it in a different way. This means the security vendor will be forced to provide a distinct implementation for each supported application server. It also means that a security vendor will only provide implementations for a few application servers, based on perceived demand. This will tend to create an additional barrier to entry to newer and smaller application server vendors.

This situation is illustrated by the current support by various application servers by AssureAccess. We have close integration with BEA, some integration with Borland and no product-specific integration with others, such as IBM. This is a consequence of the amount of information the vendors have provided about the implementation details not specified by Java standards.

It might be thought that application server vendors would suppress this information to avoid competition from security vendors. However, the general feeling among application server vendors is that they are required to provide a large number of complex services as a part of their product. It is not possible for them to maintain expertise in all areas. Security is a complex field that the access management vendors focus on exclusively. The ability of an application server vendor to exclude competition from security vendors is offset by the disadvantage of an application server competitor providing integration with a superior third party security product. This view is supported by the strong participation in the JSR 115 expert group by key application server vendors and key access management vendors.

There are many legitimate reasons why application server vendors do not wish to document their product's internal mechanisms. They may not wish to expose their technology to other

application server vendors. They may wish to maintain the flexibility to change their implementation in the future. They may simply wish to avoid the cost of creating documentation in a form suitable for external publication.

The next sections of this document describe the four Java authorization models. Only the key classes and methods of each are presented. Many additional details are required for a comprehensive understanding of all the possible conditions of use. The models are cumulative in the sense that the J2SE model is basic and the other three models build on it. Keep in mind that the behaviour described here is in addition to the code-base checks mentioned previously.

J2SE Authorization

The J2SE authorization model is in the package java.security. These are the most important classes.

Permissions

The **Permission** class is the cornerstone all the Java authorization models. A **Permission** is a way or recording the right to access some resource or set of resources. By defining new sub-classes, it is possible to specify the protection of different kinds of resources with different characteristics.

The **Permission** does not access the resource, it simply is used to keep track of who is allowed to do what. Users can be granted sets of permissions and code that needs to make access control decisions can make queries to determine if some particular **Permission** has been granted.

A **Permission** usually has a Name, which can correspond to something fairly specific, like a file name, or something more abstract like modifyThread. In some classes, like **FilePermission**, the name can represent a wildcard. In contrast, the **AllPermission** class does not have a name. The name is set by the constructor and can be accessed via the **getName** method.

Permissions can also have an Action. This can be something abstract, like read or write or it can refer to something concrete such as a method name or protocol verb. Action is also set by the constructor and accessed via the **getAction** method.

When an access control decision is to be made the procedure is to create a **Permission** of the appropriate type, name and action. This permission is then passed to the **implies** method of some permission or set of permissions that has been granted. The **implies** method determines if the permission it encapsulates implies the passed permission and returns true or false. In the simplest case, this might be a check to see if they are the same, i.e. that the Names and Actions match. However, more complex processing is possible. For example, if the granted permission has a wildcarded name, the **implies** method would determine if the passed permission's name matched the wildcard. Another example would be a classification hierarchy, where the permission to read Top Secret documents implies the ability to read Secret and Confidential documents.

The package also contains four sub-classes of permission. **AllPermission** is like Root. Its **implies** permission always returns true. In other words, it implies all other permissions. **BasicPermission** is a class that only has a Name, not an Action. **SecurityPermission** is a sub-class of **BasicPermission** that is used to control access to critical security functions.

Normally Permissions are not used individually. A **PermissionCollection** is a set that contains a number of permissions of the same class. The **Permissions** object is a set of

PermissionCollection objects, which therefore can be of different classes. Both implement the same three key methods. The **add** method adds a permission. The **element** method allows the items to be iterated. The **implies** method invokes the implies methods of the **Permission** objects that are encapsulated and relays the result.

These objects can be used in a variety of ways to keep track of what is available in a security domain, what has been granted to a user or what is required for some set of operations.

Policy

The J2SE security model specifies a single **Policy** object that encapsulates all the current information relating to permissions. The reference implementation reads information about permissions from a text file, however it is envisioned that other implementations would use LDAP or some other repository.

The **Policy** object can be queried about available permissions by means of the **getPermissions** method. Access control checking can be done by invocating the **implies** method, which in turn invokes the **implies** methods of all the encapsulated permission containers. The Policy object also implements a **refresh** method with unspecified semantics. The intent is that the policy object will update its contents, if possible, when it is called.

AccessController

An **AccessController** object is a machine for making access control decisions. It encapsulates the principal identity of the current user and all the granted permissions. By creating an appropriately named permission of the correct class and calling the **checkPermission** class method, a piece of code can decide whether or not to allow some operation. The current state of the **AccessController** can be preserved in an **AccessControlContext** object by invocating the **getContext** method. This is useful when the thread of execution transitions to a different principal or code base. The saved context can be restored later. It is also possible to invoke the **checkPermissions** method directly on an **AccessControlContext**.

ProtectionDomain

A **ProtectionDomain** is intended to encapsulate the properties of a principal. It can also be associated with a set of classes with the same code base and granted the same permissions. The **getPermission** method is used to obtain all the encapsulated permissions. The **implies** method allows some specified permission to be checked against the permissions within the **ProtectionDomain**.

Limitations

The most important thing to understand about the principal-based aspect of the J2SE authorization model is that it does not specify the means for a remote user's identity to be represented. But in a server, it is the remote identity that is needed to make access control decisions. For a server process, the local identity would typically be some kind of special administrator account and not appropriate for deciding if access should be allowed.

A further limitation of the model is that many of the details of how principals come to be represented in **AccessController** objects and objects are unspecified. Each application server provider is free to use any **ProtectionaDomain** implementation they choose. Also the method

for propagating remote identity and making it available for access control decisions is not fully specified. If it is done at all the means is proprietary.

For these reasons it is not feasible for an independent access management product to provide server authorization by means of the J2SE authorization model.

JAAS

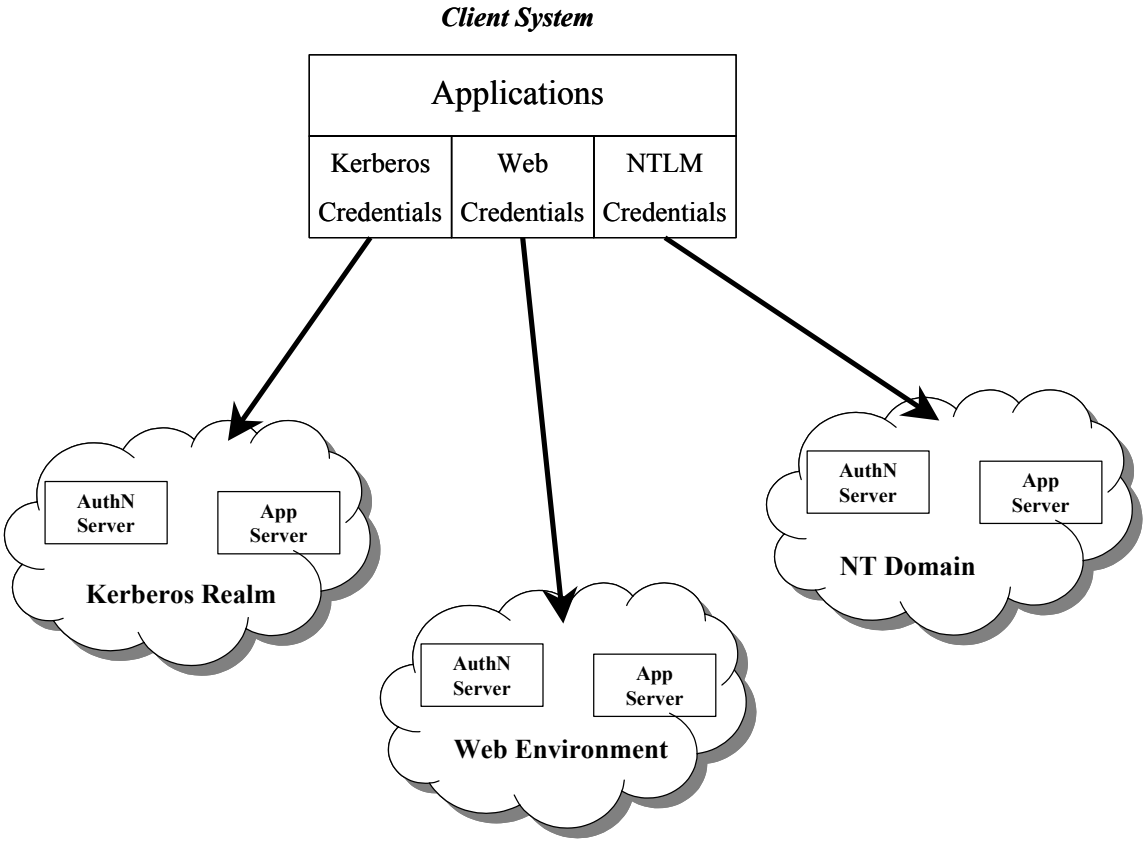

Figure 1: Client-based Single Sign-On

The Java Authentication and Authorization Service (JAAS) was developed independently of the J2SE model, using some of the same concepts. It was primarily intended to enable a pluggable authentication service which implements a client-side form of single signon. What this means is that a client system is able to login to multiple authentication systems and preserve the identities or credentials associated with each. The client then uses the appropriate information when making network requests to different servers. This is hidden from the user, thus providing a form of single signon. JAAS also allows an administrator to specify that one of several authentications must succeed or that two or more must all succeed. By definition, this type of capability requires a "heavy client" system to support this functionality. This architecture is illustrated in the diagram below.

In contrast, access management systems provide server-based single sign-on. In this environment, a standard desktop, such as a web browser is used. The user first communicates with some authentication service, which may support multiple methods of authentication and multiple user repositories. After authenticating, the client is provided with information it can

use to access application servers, such as a cookie or Kerberos ticket. By presenting this information, the client can prove it has successfully authenticated. In this model, a heavy client is not required. The servers are responsible for authentication and single sign-on rather than the client. This architecture is illustrated in Figure 2 below.

In J2SE 1.3, JAAS was included as an optional component, with a **Policy** object that replaced the standard one. In J2SE 1.4, JAAS is a standard component (although the package names were not changed), but the JAAS **Policy** object has been deprecated in favor of the J2SE one.

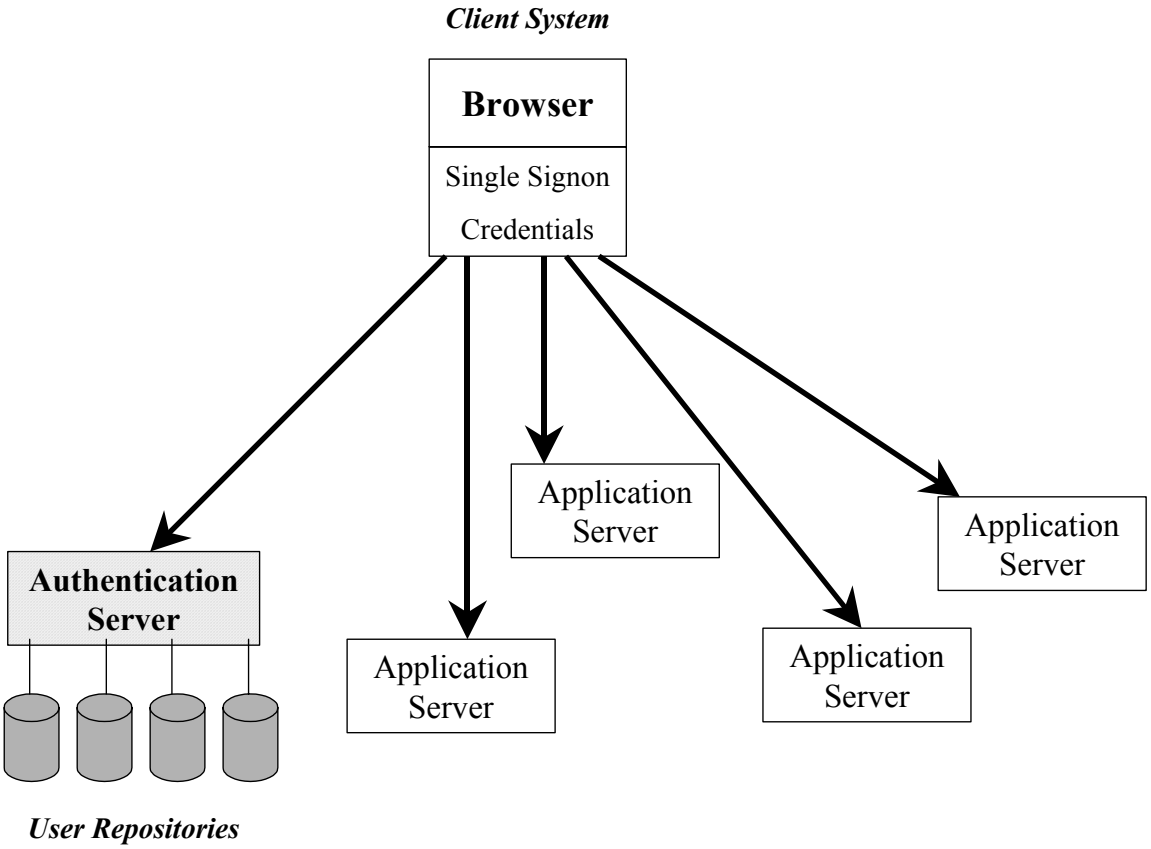

Figure 2: Server-based Single Sign-On

Pluggable Authentication

JAAS Authentication is contained in three packages. The javax.security.auth package contains the **Subject** class. A Subject can contain a number of principals as well as a number of credentials. Credentials can be public, such as a username or Kerberos ticket or private, such as a secret key or password. The **Subject** class **doAs** method allows code to be executed under a different identity. This is an example of when an **AccessControlContext** is used to preserve the previous information. The **getPrincipals** method can be used to obtain the principals contained in a particular **Subject** object.

There is also a **SubjectDomainCombiner** class that updates the **ProtectionDomain** with the Subject's Permissions.

The javax.security.auth.login package contains the **LoginContext** class. It is responsible for finding all the login modules, (based on configuration info) calling each in turn and running the two phase commit protocol based on the success or failure of each authentication and the specified logic. It implements the **login**, **logout** and **getSubject** methods, which do the obvious things.

It is intended that the javax.security.auth.spi package be provided by an independent vendor of authentication software. Each **LoginModule** class must implement authentication via the **login** method, which is called in the first phase and the **commit** and **abort** methods, one of which will be called in the second phase. It also implements the **logout** method.

In addition to these primary packages, the javax.security.auth.callback package provides definitions for classes used to callback from a login module to prompt the user for inputs or report errors. The javax.security.auth.kerberos and javax.security.auth.x500 packages provide key elements of the implementation of Kerberos and PKI login modules, respectively.

Authorization

The original JAAS **Policy** object, which drives the authorization process, has been deprecated. The J2SE **Policy** object is used instead. The **Policy** object is called with an implicit access control context or protection domain, each of which has associated principals.

Limitations

Like the J2SE model, JAAS is intended to make decisions based on the identity of the local user. There is no explicit provision for representing the identity of a remote caller. If the authentication is performed at the client system, there is no specified means for conveying this information. Therefore a security provider is dependant on the application server vendors to provide this capability and document its internal mechanisms.

In principal the JAAS authentication API could be remoted, but this is not feasible for a number of reasons:

- The security vendor would be required to provide a heavy client, an approach the market has rejected,
- There would still be no way to associate the subject with application requests,
- While a login proxy scheme could work for username/password, it is not possible for protocols such as Kerberos or SSL, and
- The client would need the keys and certificates to implement SSL just to protect the RMI calls.

The authorization portion of JAAS is now the same as J2SE, so exactly the same limitations apply to each of them.

J2EE Authorization

The J2EE security models do relate to server environments, specifically to Enterprise Java Beans (EJB) and Servlet, which includes Java Server Pages (JSP). Therefore there is support in J2EE for the notion of the caller's identity. Like JAAS, the J2EE authorization model builds on the constructs of the J2SE model.

Common

Certain concepts are used consistently across the Servlet and EJB environments.

Roles

Conceptually, Roles are a user attribute intended to control what that user is allowed to do. Users and Groups are assigned roles and that determines what URLs or EJB methods they are permitted to access. J2EE defines the Roles and their relationships to EJB methods or URLs. However, J2EE does not specify the way that Roles are granted to users or groups. In addition to automatic enforcement, based on role, APIs are provided to allow applications to get the Role of the current caller.

Role Reference

When independently developed components are deployed in the same container, it is possible that they might inadvertently use Roles with the same name, however the deployer might want to assign the distinct Roles to different groups of users. For this reason, the API calls that refer to role, actually use what is called a Role Reference. At deployment time, the Role References used in different components are mapped to the actual runtime Roles.

Deployment Descriptor

The way Roles, Role References and many other attributes of components are specified is by means of Deployment Descriptors. Deployment descriptors are stored in XML format and processed by the deployment tool when a component is deployed. Deployment descriptors describe Roles and the resources they are required for. They also describe the mapping between Roles and Role References. The security features specified via Deployment Descriptors are called the declarative security model.

Servlet Authorization

The Servlet environment features a number of specialized security features.

User Data Constraints

User Data Constraints specify the security properties required of the communications channel over which requests are made and responses received. The alternatives are NONE, INTEGRAL which means protected from modification and CONFIDENTIAL which means protected from being read. User Data Constraints are specified in a Deployment Descriptor, but the implementation is up to the application server vendor. User Data Constraints are applied to some (possibly wildcarded) set of URLs known as a web resource collection.

Auth-Constraints

The Auth-Constraint Deployment Descriptor specifies the Roles required to access a web resource collection. The application server is expected to enforce these constraints, but again, the implementation is unspecified.

HTTPServletRequest

In addition to declarative security, the Servlet environment also features programmatic security features. These are access via the **HTTPServletRequest** interface, which is defined in the javax.servlet.http package. The **getRemoteUser** method returns the name of the current user associated with the request. The **getUserPrincipal** method returns a **Principal** object with the same name. The **isUserInRole** method allows an application to determine if the caller is in a

particular role. The container can use this call to implement the specified Auth-Constraints, but is not required to.

Servlet Filter

The J2EE Servlet environment now provides the capability of installing a request filter. This is similar to the web adapter filters used by most Web servers, including iPlanet Web Server, Microsoft Internet Information Server and Apache Web Server. A web server filter operates by implementing methods that are called at defined points during the processing of requests and responses. Web filters and Servlet filters can be used for many purposes, including enforcing authorization rules. Entegrity AssureAccess version 2.0 provides a Servlet Filter to enforce security policies in a Servlet environment.

EJB Authorization

The Enterprise Java Bean environment also supports declarative and programmatic security. As with Servlets, declarative security is defined by Deployment Descriptors. The EJB environment does not currently define User Data Constraints, but equivalent semantics can be specified at deployment time and enforced at the Inter-ORB Request level. Roles and Role References are defined in Deployment Descriptors. However, Roles are mapped to sets of Permissions instead of web resources.

Permissions

Permissions are defined in a Deployment Descriptor. A Permission can be mapped to any combination of Beans, Methods or Method signatures. In effect there is a many to many mapping between Methods and Permissions and between Permissions and Roles. As with Servlet, Roles can be granted to users or groups, but the method is not specified. A user granted a role can execute any method associated with a permission contained in that role. Permissions can also be granted to users directly. The method of implementing Permissions is also unspecified, but presumably would involve **Permission** classes.

EJBContext

Programmatic security is provided via the **EJBContext** interface. The **getCallerPrincipal** method returns a Principal object representing the caller making the current request. The **isCallerInRole** method determines whether the current caller is in the specified role.

Limitations

Unlike the J2SE and JAAS models the J2EE models at least encompass the notion of a remote caller. Unfortunately so many aspects of the implementation are unspecified. There is no way for a security provider to operate without a lot of knowledge of the proprietary implementation techniques used by the application server. Application servers are supposed to implement single signon across Servlet and EJB environments, but again the method to be used is unspecified.

Many aspects of the semantics are also unspecified, for example, whether role and permission definitions must be consistent or not. Servlet roles and EJB Roles are similar, but different. User Data Constraints are not available in the EJB environment.

JSR 115

Java Specification Request (JSR) 115 is an effort under the Java Community Process (JCP) to define a Service Provider Interface (SPI) Contract, using existing APIs, that will allow Access Management products, such as Entegrity AssureAccess to plug into the Java environment. JSR 115 is taking the approach of defining a contract between an application server and a security provider. In principle, this should allow a single security provider implementation to work with any application server and vice versa.

Security providers will implement the API calls that check Roles and Permissions. By calculating the grants to users on the fly, customers will be able to take advantage of the dynamic policy capabilities of modern access management products. For example, a user might only be granted a specified role at certain times of the day or week. Or a certain permission could require access from a specified part of the network, or require a specified user attribute value.

JSR 115 proposes to make no changes to existing APIs. It attempts to include all the features of the four models and resolve the inconsistencies between them. It allows J2SE Principals or JAAS Subjects to be used. It does not specify any new management APIs.

New Permissions

JSR 115 remains consistent with both the J2SE and J2EE models by explicitly defining five permission types, WebResourcePermission, WebUserDataPermission, EJBRoleRefPermission, EJBMethodPermission and WebRoleRefPermission. Considerable effort is made to preserve the existing semantics of Servlet Roles and EJB Permissions. The Deployment Tool is still responsible for parsing the Deployment Descriptors, but the Security Provider is responsible for building the Policy Object, populating it with the appropriate permissions or providing equivalent functionality by other means.

Limitations

Since JSR 115 is not finished yet, it is too soon to be certain what its exact form will be, however some observations can be made. First, whatever its merits, it will be a year or more before it is supported by significant numbers of application servers. Obviously, users will not receive the benefits of this approach until it is being delivered in products.

A possible criticism of the JSR 115 approach is its complexity. To the existing scheme of roles, role references, permissions, implies methods and Deployment Descriptors will be added the features of third party authorization policy models. This will permit a great deal of flexibility. However some users my find it confusing to understand.

Finally, there are limitations to what JSR 115 proposes to accomplish. It will not define server-based pluggable authentication or a standard single signon scheme. It will not define management interfaces to manipulate permissions or roles.

AssureAccess JSP Tag Library

Version 1.5 of Entegrity AssureAccess introduced a JSP tag library to facilitate application development in a JSP environment. Simple, macro-like tags allow users to easily incorporate Authentication, Authorization, Audit and other dynamic policy capabilities in their JSP applications. While these tags are not a standard, they do provide customers with measure of investment protection. If customers need to change the underlying technology used for access management, it would be possible to simply redefine the Tags. This comparatively simple task would allow all existing applications that used the Tags to be instantly converted without having to recode any application.

Universal Java Plug-In

Entegrity AssureAccess version 2.0 now introduces the most revolutionary technology available anywhere for Java Access Management. The Universal Java Plug-in, based on Tangosol technology gives users the full power, flexibility and scalability of AssureAccess, **with no application programming required**. No advance specification or coding of any kind is required. Existing applications can be protected as easily as new ones. Even when no source code is available, the Universal Java Plug-in can protect any method of any object, whether called locally or remotely. All that is required is for an administrator at the AssureAccess management console to specify what methods are to be protected with what policies and the Plug-in does the rest.

This is how it works. Tangosol technology, the product of extensive research and development, understands the Java bytecodes that are the executable form of every Java program. Tangosol locates the portions of the application to be protected and inserts protective code exactly where it is needed. The Universal Java Plug-in works on any Java program, even with obfuscated code. It is safe for the future, as it only depends of the definition of the Java bytecode language, which has been stable for many years and is not expected to change.

The Universal Java Plug-in provides the most efficient security programming model possible – none. It provides AssureAccess customers with complete investment protection. There is no proprietary API whatsoever. The Universal Java Plug-in eliminates the need to do any security coding, but at the same time does not interfere with the use of any of the other Java security standards discussed in this paper.

Summary

Entegrity is fully committed to providing its customers with the most functional and easiest to use security capabilities for the Java server environments. Entegrity also has a long history of implementing all applicable standards in its products. Entegrity has been frustrated with the current infeasibility of implementing various Java authorization standards by an independent security provider. Entegrity drove the process in the JCP that led to JSR 115 and is active in that effort (in addition to other standards efforts). Entegrity will fully support JSR 115 when it is complete. In addition, Entegrity AssureAccess provides customers with a variety of alternatives designed to minimize development effort and maximize customer investment protection. These include a Servlet filter, JSP Tag library and the revolutionary Universal Java Plug-in.

About Entegrity Solutions

For more information about Entegrity Solutions, please contact us at info@entegrity.com or visit www.entegrity.com.

Entegrity Solutions Offices

West Coast:
2077 Gateway Place, Suite 200
San Jose, CA 95110
Tel: 408.487.8600 ext. 161

East Coast:
410 Amherst Street, Suite 150
Nashua, NH 03063
Tel: 603.882.1306 ext.2701

Mid-Atlantic:
10500 Little Patuxent Parkway, Suite 550
Columbia, MD 21044
Tel: 410.992.7600 ext. 3012

Europe:
Gainsborough House
58-60 Thames Street
Windsor
Berkshire SL4 1TX
United Kingdom
Tel: +44(0) 1753 272 072