

# User Authentication and Authorization in the Java™ Platform

Charlie Lai Li Gong

Larry Koved Anthony Nadalin

Roland Schemers

Sun Microsystems, Inc.  
charlie.lai,li.gong@sun.com

International Business Machines, Inc.  
koved,drsecure@us.ibm.com

onebox.com  
schemers@onebox.com

## Abstract

*Java™ security technology originally focused on creating a safe environment in which to run potentially untrusted code downloaded from the public network. With the latest release of the Java™ Platform (the Java™ 2 Software Development Kit, v 1.2), fine-grained access controls can be placed upon critical resources with regard to the identity of the running applets and applications, which are distinguished by where the code came from and who signed it. However, the Java platform still lacks the means to enforce access controls based on the identity of the user who runs the code. In this paper, we describe the design and implementation of the Java™ Authentication and Authorization Service (JAAS), a framework and programming interface that augments the Java™ platform with both user-based authentication and access control capabilities.*

## 1 Introduction

The Java™ technology [8, 12] emerged in 1995 with a prominently stated goal of providing a safe programming environment. This means that Java security must provide a secure, readily-built platform on which to run Java enabled applications. It also means that Java security must provide adequate and extensive security tools and services implemented in Java technology that enable independent software vendors (ISVs) to build a wider range of security-sensitive applications, for example, in the enterprise world.

The latest release of the Java platform (Java 2) introduces a new security architecture [7] that uses a security policy to decide the granting of individual access permissions to running code (according to the code's characteristics, e.g., where the code is coming from and whether it is digitally signed and if so by whom). Future attempts to access protected resources will invoke security checks that compare the granted permissions with the permissions needed for the attempted access. If the former includes the latter, access is

permitted; otherwise, access is denied.

Such a code-centric style of access control is unusual in that traditional security measures, most commonly found in sophisticated operating systems, are user-centric in that they apply control on the basis of who is running an application and not on the basis of which application is running. One major rationale behind code-centric access control is that when a user uses a web browser to surf the net and runs executable content (e.g., mobile code written in Java) as needed, the user variable remains essentially constant. On the other hand, the user may trust one piece of mobile code more than others and would like to run this code with more privileges. Thus it is in fact natural to control the security of mobile code in a code-centric style.

Nevertheless, it is obvious that Java is becoming widely used in a multi-user environment. For example, an enterprise application or a public Internet terminal must deal with different users, either concurrently or sequentially, and must grant these users different privileges based on their identities. The Java Authentication and Authorization Service (JAAS) is designed to provide a framework and standard programming interface for authenticating users and for assigning privileges. Together with Java 2, an application can provide code-centric access control, user-centric access control, or a combination of both.

The rest of the paper is organized as follows. Sections 2 and 3 introduce the basic concepts used by JAAS. Section 4 describes the authentication model implemented by JAAS. Section 5 describes the authorization framework for JAAS, and is broken up into several subsections. Section 5.1 defines the JAAS user-based security policy, Section 5.2 covers the JAAS access control implementation, and Section 5.3 discusses scalability issues regarding the security policy. Section 6 discusses the issue of logging into the Java virtual machine. Section 7 follows with a summary.

---

<sup>0</sup>Published in the Proceedings of the 15th Annual Computer Security Applications Conference, Phoenix, AZ, December 1999

## 2 Subjects and Principals

Users often depend on computing services to assist them in performing work. Furthermore services themselves might subsequently interact with other services. JAAS uses the term, *subject*, to refer to any user of a computing service [9, 17]. Both users and computing services, therefore, represent subjects. To identify the subjects with which it interacts, a computing service typically relies on names. However, subjects might not have the same name for each service and, in fact, may even have a different name for each individual service. The term, *principal*, represents a name associated with a subject [11, 17]. Since subjects may have multiple names (potentially one for each service with which it interacts), a subject comprises a set of principals. See Figure 1.

```
public interface Principal {
    public String getName();
}

public final class Subject {
    public Set getPrincipals() {}
}
```

**Figure 1. Subject Class and Principals**

Principals can become associated with a subject upon successful authentication to a service. Authentication represents the process by which one subject verifies the identity of another, and must be performed in a secure fashion; otherwise a perpetrator may impersonate others to gain access to a system. Authentication typically involves the subject demonstrating some form of evidence to prove its identity. Such evidence may be information only the subject would likely know or have (a password or fingerprint), or it may be information only the subject could produce (signed data using a private key).

A service's reliance on named principals usually derives from the fact that it implements a conventional access control model of security [10]. This model allows a service to define a set of protected resources as well as the conditions under which named principals may access those resources. Recent studies (PolicyMaker [4] and SPKI [5]) have focused on the limitations of using conventional names in large distributed systems for access control, and note that public keys, instead, provide a more practical and scalable name representation. JAAS, and SPKI as well, do not impose any restrictions on principal names. Localized environments that have limited namespaces, or that do not rely on public key cryptography, may define principals that have conventional names. Large-scale distributed systems may use principals that allow the principal name to be a public key (encoded as a hex string, as in PolicyMaker).

## 3 Credentials

Some services may want to associate other security-related attributes and data with a subject in addition to principals. JAAS refers to such generic security-related attributes as *credentials*. A credential may contain information used to authenticate the subject to new services. Such credentials include passwords, Kerberos tickets [16], and public key certificates (X.509 [9], PGP [21], etc.), and are used in environments that support single sign-on. Credentials might also contain data that simply enables the subject to perform certain activities. Cryptographic keys, for example, represent credentials that enable the subject to sign or encrypt data. JAAS credentials may be any type of object. Therefore, existing credential implementations (java.security.cert.Certificate, for example) can be easily incorporated into JAAS. Third-party credential implementations may also be plugged into the JAAS framework.

JAAS credential implementations do not necessarily have to contain the actual security-related data; they might simply reference the data. This occurs when the data must physically reside on a separate server, or even possibly in hardware (private keys on a smart card, for instance). Also, JAAS does not impose any restrictions regarding credential delegation to third parties. Rather it allows each credential implementation to specify its own delegation protocol (as Kerberos does), or leaves delegation decisions up to the applications.

JAAS divides each subject's credentials into two sets. One set contains the subject's public credentials (public key certificates, Kerberos tickets, etc). The second set stores the subject's private credentials (private keys, encryption keys, passwords, etc). To access a subject's public credentials, no permissions are required. However, access to a subject's private credential set is security checked. See Figure 2.

```
public final class Subject {
    ...
    // not security checked
    public Set getPublicCredentials() {}

    // security checked
    public Set getPrivateCredentials() {}
}
```

**Figure 2. Subject Class and Credentials**

## 4 Pluggable and Stackable Authentication

Depending on the security parameters of a particular service, different kinds of proof may be required for authentication. The JAAS authentication framework is based on

PAM [18, 20], and therefore supports an architecture that allows system administrators to plug in the appropriate authentication services to meet their security requirements. The architecture also enables applications to remain independent from the underlying authentication services. Hence as new authentication services become available or as current services are updated, system administrators can easily plug them in without having to modify or recompile existing applications.

The JAAS *LoginContext* class represents a Java implementation of the PAM framework. The *LoginContext* consults a configuration that determines the authentication service, or *LoginModule*, that gets plugged in under that application (See Figure 3). The syntax and details of the configuration are defined by PAM.

```
public final class LoginContext {
    public LoginContext(String name) {}
    public void login() {} // two phase process
    public void logout() {}

    // get the authenticated subject
    public Subject getSubject() {}
}

public interface LoginModule {
    boolean login(); // first phase
    boolean commit(); // second phase
    boolean abort();
    boolean logout();
}
```

**Figure 3. LoginContext Class and LoginModule Interface**

JAAS, like PAM, supports the notion of stacked LoginModules. To guarantee that either all LoginModules succeed or none succeed, the *LoginContext* performs the authentication steps in two phases. In the first phase, or the *login* phase, the *LoginContext* invokes the configured LoginModules and instructs each to attempt the authentication only. If all the necessary LoginModules successfully pass this phase, the *LoginContext* then enters the second phase and invokes the configured LoginModules again, instructing each to formally *commit* the authentication process. During this phase each LoginModule associates the relevant authenticated principals and credentials with the subject. If either the first phase or the second phase fails, the *LoginContext* invokes the configured LoginModules and instructs each to *abort* the entire authentication attempt. Each LoginModule then cleans up any relevant state they had associated with the authentication attempt.

In addition to JAAS, the Generic Security Services Ap-

plication Programmer’s Interface (GSS-API) and Simple Authentication and Security Layer Application Programmer’s Interface (SASL) [13, 14] define frameworks that provide support for pluggable authentication. However, the GSS and SASL authentication frameworks are designed specifically for network communication protocols and, as such, provide additional support for securing network communications after authentication has completed. While JAAS does accommodate general network-based authentication protocols (including Needham-Schroeder and EKE [15, 2]), it also focuses on addressing the need to support pluggable authentication in stand-alone non-connection oriented environments.

## 5 Authorization

Once authentication has successfully completed, JAAS provides the ability to enforce access controls upon the principals associated with the authenticated subject. The JAAS principal-based access controls (access controls based on who runs code) supplement the existing Java 2 codesource-based access controls (access controls based on where code came from and who signed it).

### 5.1 Principal-Based Access Control

As stated earlier, services typically implement the access control model of security, which defines a set of protected resources, as well as the conditions under which named principals may access those resources. JAAS also follows this model, and defines a security policy to specify what resources are accessible to authorized principals. The JAAS policy extends the existing default Java 2 security policy, and in fact, the two policies, together, form a single logical access control policy for the entire Java runtime.

Figure 4 depicts an example codesource-based policy entry currently supported by the default policy provided with Java 2. This entry grants code loaded from *foo.com*, and signed by *foo*, permission to read all files in the *cdrom* directory and its subdirectories. Since no principal information is included with this policy entry, the code will always be able to read files from the *cdrom* directory, regardless of who executes it.

```
// Java 2 codesource-based policy
grant Codebase “http://foo.com”,
    Signedby “foo” {
    permission java.io.FilePermission
        “/cdrom/-”, “read”;
}
```

**Figure 4. Codesource-Based Policy Entry**

Figure 5 depicts an example principal-based policy entry supported by JAAS. This example entry grants code loaded from *bar.com*, signed by *bar*, and executed by *duke*, permission to read only those files located in the */cdrom/duke* directory. To be executed by *duke*, the subject affiliated with the current access control context (see Section 5.2) must have an associated principal of class, *bar.Principal*, whose *getName* method returns, *duke*. Note that if the code from *bar.com*, signed by *bar*, ran stand-alone (it was not executed by *duke*), or if the code was executed by any principal other than *duke*, then it would not be granted the *FilePermission*. Also note that if the JAAS policy entry did not specify the *Codebase* or *Signedby* information, then the entry's *FilePermission* would be granted to any code running as *duke*.

```
// JAAS principal-based policy
grant Codebase "http://bar.com",
    Signedby "bar",
    Principal bar.Principal "duke" {
    permission java.io.FilePermission
        "/cdrom/duke", "read";
}
```

**Figure 5. Principal-Based Policy Entry**

JAAS treats roles and groups simply as named principals [10]. Therefore access control can be imposed upon roles and groups just as they are with any other type of principal. See Figure 6.

```
// an administrator role can access user passwords
grant Principal foo.Role "administrator" {
    permission java.io.FilePermission
        "/passwords/-", "read, write";
}

// a basketball team (group) can read its directory
grant Principal foo.Team "SlamDunk" {
    permission java.io.FilePermission
        "/teams/SlamDunk/-", "read";
}
```

**Figure 6. Role-Based and Group-Based Policy Entries**

For flexibility, the JAAS policy also permits the *Principal* class specified in a grant entry to be a *PrincipalComparator* (the class implements the *PrincipalComparator* interface). The permissions for such entries are granted to any subject that the *PrincipalComparator* *implies*. See Figure 7.

Figure 7 demonstrates how *PrincipalComparators* can be used to support role hierarchies [19]. In this example assume that an administrator role is senior to a user role and, as such, administrators inherit all the permissions granted

```
public interface PrincipalComparator {
    boolean implies(Subject subject);
}

// regular users can access a temporary
// working directory
grant Principal bar.Role "user" {
    permission java.io.FilePermission
        "/tmp/-", "read, write";
}
```

**Figure 7. PrincipalComparator Interface and Example Policy Entry**

to regular users. To accommodate this hierarchy, *bar.Role* must simply implement the *PrincipalComparator* interface, and its *implies* method must return, *true*, if the provided subject has an associated "administrator" role principal. Note that although the JAAS policy supports role hierarchies via the *PrincipalComparator* interface, administrators are not limited by it. JAAS can accommodate alternative role-based access control mechanisms (such as that defined in [6]), as long as the alternative access controls can be expressed either through the existing Java 2 policy or the new JAAS policy.

## 5.2 Access Control Implementation

The Java 2 runtime enforces access controls via the *java.lang.SecurityManager*, and is consulted any time untrusted code attempts to perform a sensitive operation (accesses to the local file system, for example). To determine whether the code has sufficient permissions, the *SecurityManager* implementation delegates responsibility to the *java.security.AccessController*, which first obtains an image of the current *AccessControlContext*, and then ensures that the retrieved *AccessControlContext* contains sufficient permissions for the operation to be permitted.

JAAS supplements this architecture by providing the method, *Subject.doAs*, to dynamically associate an authenticated subject with the current *AccessControlContext*. Hence, as subsequent access control checks are made, the *AccessController* can base its decisions upon both the executing code itself, and upon the principals associated with the subject. See Figure 8.

To illustrate a usage scenario for the *doAs* method, consider when a service authenticates a remote subject, and then performs some work on behalf of that subject. For security reasons, the server should run in an *AccessControlContext* bound by the subject's permissions. Using JAAS, the server can ensure this by preparing the work to be performed as a *java.security.PrivilegedAction*, and then by in-

```

public final class Subject {
    ...
    // associate the subject with the current
    // AccessControlContext and execute the action
    public static Object doAs
        (Subject s,
         java.security.PrivilegedAction action) {}
}

```

**Figure 8. Subject doAs Method**

voking the doAs method, providing both the authenticated subject, as well as the prepared PrivilegedAction. The doAs implementation associates the subject with the current AccessControlContext and then executes the action. When security checks occur during execution, the Java 2 SecurityManager queries the JAAS policy, updates the current AccessControlContext with the permissions granted to the subject and the executing codesource, and then performs its regular permission checks. When the action finally completes, the doAs method simply removes the subject from the current AccessControlContext, and returns the result back to the caller.

To associate a subject with the current AccessControlContext, the doAs method uses an internal JAAS implementation of the *java.security.DomainCombiner* interface, newly introduced in version 1.3 of the Java 2 SDK. It is through the JAAS DomainCombiner that the existing Java 2 SecurityManager can be instructed to query the JAAS policy without requiring modifications to the SecurityManager itself. Details of the interaction between the Java 2 SecurityManager and DomainCombiners are documented in the javadocs for the *java.security.DomainCombiner* interface.

### 5.3 Scalability of the Access Control Policy

The JAAS principal-based access control policy was intentionally designed to be consistent with the existing codesource-based policy in the Java 2 platform. The default policy implementations provided with both Java 2 and JAAS reside in a local file, and assume that all policy decisions can be defined and made locally. Obviously, this design does not scale beyond small localized environments. KeyNote [3] and SPKI both address the limitations of such access control designs, and discuss alternative solutions that enable the delegation of policy responsibilities to certified 3rd parties. By delegating policy-making responsibilities, access control policies can easily scale to serve larger systems.

To improve scalability, both the Java 2 and JAAS file-based policy implementations can be replaced with alternative implementations that support delegation. This is achieved by specifying the alternative implementations in

the *java.security* properties file located in the lib/security subdirectory from where the Java runtime environment was installed. The designs of potential alternative implementations are beyond the scope of this paper.

## 6 Logging in to the Java Virtual Machine

With support from the JAAS framework, the Java virtual machine (VM) can be augmented to provide a general login facility for users. This would enable the VM itself to impose access controls based on who logged in. In fact, [1] investigates and describes the constructs necessary to support a multi-user environment within a VM. In such an environment, individual users log into the VM and are each given an execution shell in which to launch commands and applications (similar to Unix). The VM imposes access controls based on the identity of the user, and special UserPermissions may be granted to code running as a particular user to permit access to particular resources.

JAAS can serve as the underlying authentication architecture for such a system. Also, the environment described in [1] focuses on user-based authentication and access control from the point of view of the Java virtual machine. The JAAS framework supplements this environment by providing the support necessary for developers to build the same user-based authentication and access control capabilities into their own applications.

## 7 Summary and Future Directions

In this paper, we have outlined the design and implementation of the *Java™ Authentication and Authorization Service (JAAS)*, a framework and programming interface that augments the Java™ platform with both pluggable authentication and principal-based access control capabilities, without requiring modifications to the Java 2 core. Although individual pluggable LoginModules can be written in native code, the basic JAAS framework can be written entirely in Java. A prototype implementation of the framework has been developed, and is currently packaged as a Java 2 standard extension consisting of approximately 25 classes partitioned into four packages.

As Java technology is used to construct not just a single desktop but a full-fledged distributed system, a whole new range of distributed systems security issues (such as those we touched upon in the Introduction chapter) must be tackled. For example, additional mechanisms are needed to make RMI secure in the presence of hostile network attacks. For Jini, service registration and location must be securely managed if the environment contains coexisting but potentially mutually hostile parties. There is a full set of higher-level concepts and services that must be secured, such as

transactions for electronic commerce. There are also many lower-level security protocols that we can leverage on, such as the network security protocols Kerberos and IPv6. JAAS is a critical building block for all these issues.

## 8 Acknowledgements

We are grateful to Bob Scheifler for his comments and feedback on the JAAS architecture. We also thank Bruce Rich, Kent Soper, Anat Sarig, Maryann Hondo, and David Edelson for their work in helping to define JAAS' functional requirements, and for their assistance in testing and documenting JAAS' features. Whitfield Diffie, Gary Ellison, Rosanna Lee, Jan Luehe, Peter Neumann, Jeff Nisewanger, Jerome Saltzer, Fred Schneider, Michael Schroeder, Scott Seligman, and Rob Weltman all contributed to early JAAS designs. Maxine Erlund provided management support for the JAAS project. Sriramulu Lakkaraju and Narendra Patil wrote product tests for JAAS. Scott Hommel helped edit this paper.

## References

- [1] D. Balfanz and L. Gong. Experience with Secure Multi-Processing in Java. In *Proceedings of ICDCS*, May 1998.
- [2] S. Bellovin and M. Merritt. Encrypted Key Exchange: Password-Based Protocols Secure Against Dictionary Attacks. In *Proceedings of the IEEE Symposium on Research in Security and Privacy*, May 1992.
- [3] M. Blaze, J. Feigenbaum, and A. Keromytis. Keynote: Trust Management for Public-Key Infrastructures. In *Proceedings of the Security Protocols International Workshop*, 1998.
- [4] M. Blaze, J. Feigenbaum, and J. Lacy. Decentralized Trust Management. In *Proceedings of the IEEE Conference on Security and Privacy*, May 1996.
- [5] C. M. Ellison, B. Frantz, B. Lampson, R. Rivest, B. M. Thomas, and T. Ylonen. SPKI Certificate Theory. Internet Engineering Task Force, November 1998. Internet Draft.
- [6] L. Giuri and F. U. Bordoni. Role-Based Access Control in Java. In *Proceedings of the 3rd ACM Workshop on Role-Based Access Control*, 1998.
- [7] L. Gong, M. Mueller, H. Prafullchandra, and R. Schemers. Going Beyond the Sandbox: An Overview of the New Security Architecture in the Java™ Development Kit 1.2. In *Proceedings of the USENIX Symposium on Internet Technologies and Systems*, December 1997.
- [8] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. Addison-Wesley, Menlo Park, California, August 1996.
- [9] R. Housley, W. Ford, T. Polk, and D. Solo. Internet X.509 Public Key Infrastructure Certificate and CRL Profile. Internet Engineering Task Force, January 1999. Request for Comments 2459.
- [10] B. Lampson. Protection. *ACM Operating Systems Review*, 8(1):18–24.
- [11] B. Lampson, M. Abadi, M. Burrows, and E. Wobber. Authentication in Distributed Systems: Theory and Practice. *ACM Transactions on Computer Systems*, 10(4):265–310.
- [12] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, Menlo Park, California, 1997.
- [13] J. Linn. Generic Security Service Application Program Interface, Version 2. Internet Engineering Task Force, January 1997. Request for Comments 2078.
- [14] J. Myers. Simple Authentication and Security Layer (SASL). Internet Engineering Task Force, October 1997. Request for Comments 2222.
- [15] R. M. Needham and M. D. Schroeder. Using Encryption for Authentication in Large Networks of Computers. *Communications of the ACM*, 21(12):993–999.
- [16] B. C. Neuman and T. Ts'o. Kerberos: An Authentication Service for Computer Networks. In *IEEE Communications*, volume 39, pages 33–38.
- [17] T. Ryutov and B. C. Neuman. Access Control Framework for Distributed Applications. Internet Engineering Task Force, November 1998. Internet Draft.
- [18] V. Samar and C. Lai. Making Login Services Independent from Authentication Technologies. In *Proceedings of the SunSoft Developer's Conference*, March 1996.
- [19] R. S. Sandhu, E. J. Coyne, H. L. Feinstein, and C. E. Youman. Role-Based Access Control Models. *IEEE Computer*, 29(2):38–47.
- [20] www.opengroup.org. X/Open Single Sign-On Service (XSSO) - Pluggable Authentication. In *Preliminary Specification P702*, June 1997.
- [21] P. Zimmerman. *PGP User's Guide*. MIT Press, Cambridge, 1994.