Whatever Happened to JAAS?

Introduced in 1995, Java has firmly established itself as a mature mainstream programming language for enterprises. The Java platform security model has evolved over the years to meet new requirements, and today enterprise Java developers have a large number of APIs and services to choose from to fulfill their security needs.

Originally touted as a secure runtime environment for downloadable executables (applets), Java platform security received a lot of attention early on and the rather inflexible security model was quickly identified as a weakness in the system. With the Java 2 Platform, Enterprise Edition (J2EE), Sun revamped the Java platform security model and introduced a fine-grained, flexible, and extensible security model for code-based security. This new model has largely been a success but it was restricted to code-based security. This makes sense for browser-based deployment environments but is not as relevant for server-side deployments. To address this gap, Java Authentication and Authorization Service (JAAS) was introduced as an optional package for Java Development Kit (JDK) 1.3.1, and subsequently integrated into JDK 1.4.

Sun officially announced Java Platform, Enterprise Edition (Java EE) 1.0 in 2000. Just as Java became the mainstream programming language of choice, Java EE has been widely adopted as the primary application platform for enterprises. But the security needs of enterprise applications are quite different from those of downloadable executable code, so Java EE defined its own security model that is declarative, consistent, and portable across Java EE implementations.

JAAS was included officially as part of the Java EE 1.3 specification. Unfortunately, Java EE 1.3 did not attempt to resolve the differences between the Java EE security model and JAAS, which made it more difficult to deploy off-the-shelf JAAS login modules in some vendors' implementations.

State of Java EE Security
In this section, we'll provide a brief overview of the main players in the Java EE security landscape.

Java EE Declarative Security
Java EE defines a declarative and portable security model that applies to both Web and Enterprise JavaBeans (EJB) modules and applications. The main goal of the declarative security model is to decouple security concerns from business application logic, so enterprise application developers can focus on the main business functions and services provided by the applications without worrying about security.

The Java EE declarative security model applies to the servlet container and the EJB container. The servlet container security model is based on URL patterns, whereas the EJB security model is method-based.

For security-aware applications, Java EE also defines a programmatic API to handle more advanced security needs.

JAAS

Introduced as an option for JDK 1.3.1 and incorporated into J2EE 1.4, today JAAS has been quite widely adopted by commercial and open source vendors as the primary pluggable authentication framework for SE and EE applications alike.

JAAS defines a framework for subject-based authentication and authorization in a pluggable manner, decoupling applications from underlying security implementations.

Key components of JAAS include:

- The definition of *subject* (*javax.security.auth.Subject*)
- An authentication API (*javax.security.auth.login.LoginContext*) that supports pluggable and stacked authentications
- An authentication SPI (*javax.security.auth.spi.LoginModule*) for pluggable authentication mechanisms
- A configuration contract for configuring and associating login modules with applications
- A typesafe callback contract (*javax.security.auth.callback*) for services to communicate with applications
- The definition of SubjectDomainCombiner (*javax.security.auth.SubjectDomainCombiner*) for dynamically updating the protection domains with the principals from the subject, for integration with the J2EE security model

Figure 1 illustrates the JAAS authentication architecture.

The JAAS authentication framework has been stable since JAAS 1.0 - there were some very minor changes (the introduction of a *LoginContext* constructor that takes a *configuration* as an argument for dynamic configuration of login modules, for example), but in general applications written on top of JAAS 1.0 (JDK 1.3.1) continue to work without modification today (JDK 6.0). Perhaps more remarkable, login modules written to the JAAS 1.0 SPI can be plugged into today's applications, which reflects well on the overall design of the API.

Figure 2 illustrates the typical authentication call sequence.

Note that the application developer is largely decoupled from underlying login module implementations - especially if the login module implementation only uses the standard callbacks.

When integrating JAAS authentication with Java EE implementations, however, a number of thorny issues arise.

- Although JAAS uses *subject* to represent a user, Java EE uses *principal*. A *subject* can - and usually does - contain multiple (custom) *principal* instances, so there must be a mechanism whereby a container can determine the *principal* instance representing the caller or user.
- The standard JAAS callbacks are useful and valuable for Java Platform, Standard Edition applications, but they do not cover typical enterprise deployment scenarios. For instance, there is no standard definition of *HttpServletRequestCallback* or *HttpServletResponseCallback*, so if a login module provider needs to access the *HttpServletRequest* object (say to retrieve a HTTP header value), the
  login module provider must resort to a vendor-specific API, compromising portability.

Java Specification Request 196

When this was written, Java Specification Request (JSR) 196 was slated to be included as part of Java EE 6.0.

Figure 3 depicts the JSR 196 generic message processing model and the four interaction points.

This generic model applies to any message-processing runtime that integrates with JSR 196. The reader is encouraged to peruse JSR 196 for more details.

As part of JSR 196, a servlet container profile is defined, clarifying how servlet container implementations can integrate with this contract. When this was written, this profile is under consideration as a Java EE 6.0 requirement.

JSR 196 defines a standard SPI for authentication providers. Therefore, the interfaces introduced by JSR 196 are primarily used by Java EE container vendors. Consequently, Java EE application developers are not directly affected by a container vendor's uptake of this SPI.

On the other hand, JSR 196 directly addresses a few long-standing problems in the JAAS/Java EE landscape, namely determining a standard way to obtain user principals and group principals.

JAAS, being a flexible, pluggable standard, allows any login module to define its own custom principal type(s) and populate the custom principal instance(s) into the subject. But because the custom principal type(s) are not known to the container, the container can only determine which principal represents the caller's identity by introducing *CallerPrincipalCallback*. JSR 196 addresses this issue. The *ServerAuthModules* can indicate to the container which principal instance represents the *caller*, and you can ensure that *getUserPrincipal* and *getRemoteUser* return the correct and expected value instead of leaving it to the mercy of the vendor's implementation details.

The need to obtain group principals is an equally fundamental issue. This is addressed by introduction of *GroupPrincipalCallback* in JSR 196.

Conclusion
JAAS has come a long way since it was an option for JDK 1.3.1. With the introduction of JSR 196 in Java EE 6.0, many of the thorny issues that exist today will finally be addressed. JAAS has certainly evolved, and it looks to be well positioned to have a place in the enterprise marketplace for years to come.

• • •

SOAP Box: The Talented Callback Handler
One of the most versatile components introduced by JAAS is the typesafe *callback* contract (*javax.security.auth.callback*). We assume the reader is already familiar with the commonly used *NameCallback* and *PasswordCallback* through which the security services (login modules) obtain user name and password information. However, the callback contract is actually a generic two-way communication model that can be applied in many circumstances, both security-related and not.

For instance, JSR 196 takes advantage of the *callback* contract by defining a *CallerPrincipalCallback* whereby a security service (such as *ServerAuthModule*) can communicate with the container (or application) that the *principal* instance represents, solving a long-standing issue. JSR 115 defines a similar *callback* contract (*javax.security.jacc.PolicyContextHandler*) that enables the container to communicate additional contextual information to the policy provider to support advanced security policies such as the instance-based security model. In fact, with the callback model, notions such as XACML-styled obligations can be supported as well, without affecting existing APIs.

Finally, JSR 196 creates a bridge to JAAS login modules, helping to preserve existing investments in JAAS. It also properly decouples JAAS login modules from any protocol-specific processing. In the past, attempts to create a JAAS login module to integrate with a single sign-on solution failed because there was no portable way to access the *HttpRequest/Response* objects. JSR196 solves that issue - not by making

HttpRequest/Response objects available to JAAS login modules, but by cleanly separating the component that deals with protocol-specific processing from the component that deals with credential validation (the JAAS login module). We believe this is the right approach.

When this was written, Glassfish was the only major Java EE provider that supports JSR 196, but the servlet container profile was slated to be part of Java EE 6.0, so you can expect major Java EE vendors to support this contract as well.

• • •

SOAP Box: Why JSR 196?
The benefits of JSR 196 include:

- It is sufficiently rich to enable implementations of complex authentication protocols
- It is portable across component containers and servers
  -WORA capabilities for server-side authentication modules
  -Application-level binding supports application-level authentication configurations and mechanisms
  -Security-aware enterprise applications can remain portable
- It properly delegates security processing to ServerAuthModules, which are registered/configured at the container level and can be associated with applications
- It supports and preserves the existing declarative Java EE security model, auth-constraint processing, and the like
- It defines a servlet container profile and a SOAP profile (with Java Message Service, Internet Inter-ORB Protocol, and the like) to be defined in the future

By supporting JSR196, Java EE vendors benefit by leveraging the available *ServerAuthModule* implementations for competitive advantages.

How about Servlet Filters? Servlet filter is a very useful mechanism, but servlet filters must be configured as part of the application (via web.xml) and are considered part of the application. In fact, by the time a servlet filter is invoked, the authentication has already happened and auth-constraints have already been processed. That does not mean we cannot use servlet filters for security needs, but there are some undesirable tradeoffs with this approach.

How about Spring Security? Spring is perhaps the most important de facto application framework in use today, and Spring security (formerly ACEGI security) is the main security framework provided by Spring. It takes advantage of Spring-styled configuration, aspect-oriented programming, and inversion of control technologies. Not surprisingly, Spring security uses a filter-based approach. Users must configure a series of filters as part of the application, and then Spring security effectively replaces Java EE container security. To compensate, Spring security provides its own security model based on URL patterns, and provides features typically found in servlet container implementations. The features offered by Spring security are indeed valuable and should not be underestimated - and aside from servlet filters, there are few other options available today. However, there are inevitably tradeoffs involved when this method is employed, and users should be aware of them. In fact, when JSR 196 becomes readily available, we expect some of the features implemented via servlet filters be folded into *ServerAuthModule* implementations.