Using JAAS and Sun Java System Access Manager to authenticate federally-identified users of a web-application

A Case Study

David Bullock

Australian Java User's Group

<db@dawnbreaks.net>

ABSTRACT

From the perspective of applications that use authentication services, federated identity is similar to more established forms of centralized account-management facilitating single-sign-on (SSO), except that an application accepting connections from federally-identified users no longer directly receive user credentials.

Existing Java APIs for access-control in both JAAS (`checkPermission()`) and J2EE (`isCallerInRole()`) already provide near-transparent authentication, and are well-placed to exploit federated identity without significant application changes.

The push for federated identity is likely to increase awareness of opporunities that centralized account management afford, because it requires less trust in applications. Centralized account management is an enabler of central management of access-policies. The most useful policies can evaluate the permissions of a user with respect to specific *application resources*. However, to allow external evaluation, applications must code policy-enforcement points without assuming any particular policy framework.

Using JAAS in combination with Sun Java System Access Manager it is possible to achieve enforcement of policy without the application assuming anything about how a user comes to have a permission, thereby allowing use of policy frameworks such as Role Based Access Control. However, the programming contract of J2EE ironically interferes with protection of application resources in an enterprise environment, even though that environment is where browser-based federated identity makes sense.

We consider how we would approach coding an access control point for p-Contact, a web-based contact-management system which allows users to define, use and share mailing-lists according to a fine-grained access-control scheme that supports conformance with Australian privacy law.

1. Federated versus Centralized Identity

Federated identity changes where a user places their trust. In centralized schemes, the user trusts the application, whereas in federated schemes, the user trusts the federator. Figures 1 and 2 depict the roles played by each software actor in centralized and federated schemes, respectively.

In each case, the application ends up with the notion that some user, identified by a *principal*, 'is logged in'.

Federated authentication protocols are not new. Every security technology supporting single-sign-on uses the basic federation idea.

The current rush towards 'federated identity' also seeks to ascribe to the federation service management of sensitive information other than login-credentials (such as contact information), to apply the technique brazenly across organizational boundaries, and to achieve it using an unmodified HTTP user agent [13,14].

2. Java Authentication and Authorization Service

The Java Security Architecture initially had support only for enforcing access-control based on the location from which the currently-executing code was obtained [1] (a necessary feature given the dynamic late-binding facilities of Java). Later, support was added for enforcing access-control based on the identity of the user accessing the code, as well as an authentication system similar to the Linux Pluggable Authentication Module (PAM) to authenticate

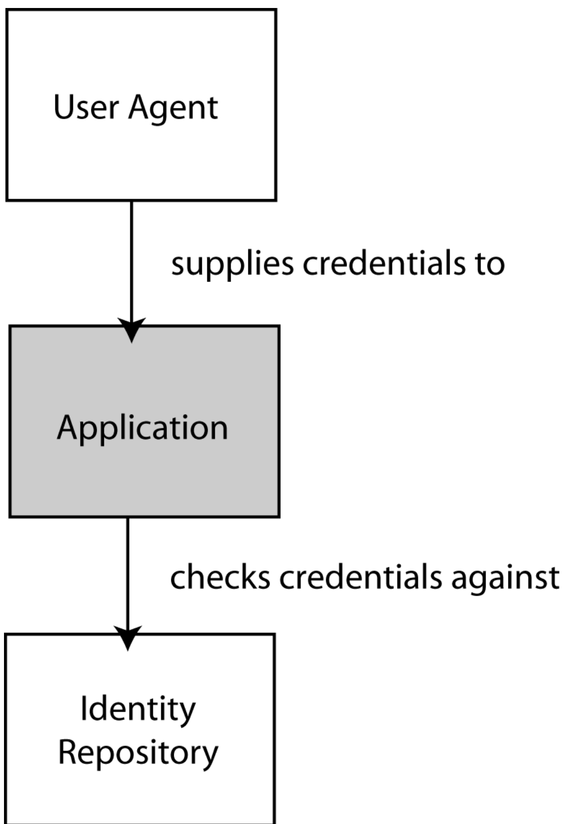

Figure 1: Centralized Identity

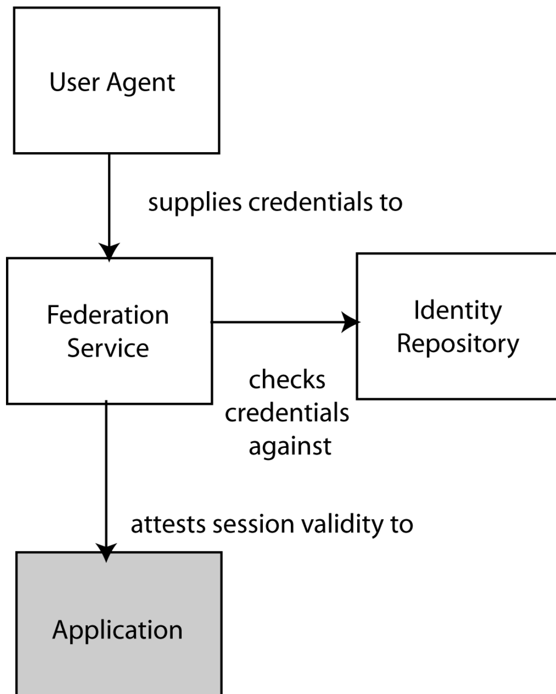

Figure 2: Federated Identity

users. The Java Authorization and Authentication System (JAAS) [2] was integrated into the platform with JDK 1.4.

2.1 Authorization

Authorization in the unified security model is performed by a `Policy` object which tests whether the current `AccessControlContext` is associated with a `Permission` due the location of its codebase and/or the identity of the user associated with the current thread.

The following example adapted from the Java Security Architecture specification (3.1.18) uses a concrete sub-class of the abstract `java.security.Permission` class to represent an operation or *action*, `watch`, on some system resource or *target*, `channel-5`, in a television (where a Java Virtual Machine may be running):

```
public void watchChannel5(){
 TVPermission tvperm =
    new TVPermission("channel-5", "watch");
 AccessController.checkPermission(tvperm);
 /* ... code being guarded */
}
```

The `AccessController` will throw an `AccessControlException`, unless the permission had previously been associated with the execution context. The `AccessController` delegates ultimately to the configured `Policy` object.

The default `Policy` object evaluates permissions based on statements in an administrator-configured policy file. The following statement restricts watching `channel-5` to code downloaded from Sun:

```
grant codeBase "http://java.sun.com/" {
  permission com.abc.TVPermission
        "channel-5", "watch";
}
```

Or to specify that only users having the principal "channel-surfer" are allowed to watch the channel, we could alternately specify:

```
grant Principal "channel-surfer", codeBase
"http://java.sun.com/" {
  permission com.abc.TVPermission
        "channel-5", "watch";
}
```

Some points to note:

- Although the default Policy implementation requires Permissions to take String argument because it must construct them from string-literals in the policy file, there is no fundamental restriction on what objects a Permission associates together.
- It is not essential that policies be stored in a file. An alternative `Policy` object (configurable by the JVM administrator) could fetch policies from a database, or delegate the decision to an external system. This is exactly the approach taken by Sun Java Identity server to provide centralized management of access-control policies.
- The class of the permission and the arguments it takes are not fixed in advance. Provided the default Policy object is able to construct a `Permission` from information in the policy file, and to use its `equals(Object)`

or implies(Permission) methods, any kind of information we desire may qualify the permission.

- The principal can be a role if desired, as is the case here with "channel-surfer". The default Policy object relies on authentication having associated the role-principal with the JAAS Subject. (This is not the only way to evaluate permissions – the default Policy object simply happens to find this convenient).

2.2 Authentication

JAAS gives us a consistent interface to any number of login modules which may have been configured by an administrator. Briefly the login code is like:

```
LoginContext lc =
  new LoginContext(
    "authenticationDomain",
    new CustomCallbackHandler()
  );
lc.login();
Subject s = lc.getSubject();
```

The callback interface implemented by CustomCallbackHandler here allows the login-modules to query the application for information such as username, password, challenge-response, etc, which the application will collect from the user as necessary.

After successful login, we now have a Subject which encpasulates the identity of a user, as authenticated by whatever login modules the administrator thought appropriate to use.

2.3 Associating Users to AccessControlContexts

To associate a particular user (an application may be interacting with more than one) with the current AccessControlContext, an application must:

```
PrivilegedAction privilegedAction =
  new PrivilegedAction() {
    public Object run() { watchChannel5(); }
  }
Subject user = getUser(); // authenticate
Subject.doAs(user, privilegedAction);
```

Here, the doAs(Subject, PrivilegedAction) invocation temporarily associates permissions held by the user with the current AccessControlContext, while the run() method of the PrivilegedAction is executing, and dissociates them immediately when the run() method terminates (either naturally, or because of a SecurityException or other unchecked Exception or Error).

2.4 Resource Collections

One of the resources protectable by the Java Security Architecture are files. Clearly, it would be tedious to specify in the policy file the precise permissions of each file with respect to each code-base location and user-role combination.

For some types of resources (such as files on file-system), it may be appropriate to have a 'super-permission' which implies permissions on portions of a resource hierarchy. For example, for the following permissions below,

```
Permission p1 =
  new FilePermission("file:/tmp/*", "read");
Permission p2 =
  new FilePermission("file:/tmp/jones",
                                        "read");
Permission p3 =
  new FilePermission("file:/*", "write");
```

the implies(Permission) method of FilePermission can be overridden such that:

```
(p1.implies(p2) == true) &&
(p1.implies(p3) == false) &&
(p2.implies(p1) == false) &&
(p2.implies(p3) == false) &&
(p3.implies(p1) == true) &&
(p3.implies(p2) == true)
```

This facility is useful both for specification of permissions in the default policy file (since everything must be represented as strings there anyway).

It is also useful because representing a Subject's permissions over the entire file-system does not require a Permission instance for each file in the file-system to be carried by the Subject.

On this latter point, it appears to have been an early design approach that concrete Permission instances were available for evaluation via the implies method. For the purposes of protecting access to system resources, which is easily satisfied by local evaluation of the policy, this approach was adequate.

However, since JDK 1.4 there is also an implies method on the Policy class. This allows the policy to be evaluated remotely, without transporting perhaps hundreds of Permission classes on the wire.

This is one tool which we can use when enforcing access-control to application resources, since the sheer number of those resources could often be an obstacle to centralized access-control.

3. Policy Evaluation

When we guard a code-block with AccessController.checkPermission(Permission), and associate a Subject with an

`AccessControlContext`, our application code assumes that the installed `Policy` object will know how to associate the user with the permission, if indeed they do have it.

It is imperative that applications be agnostic about how users acquire permission, to allow organisations to externally implement whatever security policy they like, whether it be Role Based Access Control [4] or something more substantial [12].

We really do want

```
{
  Permission perm =
    new Permission(resource, action);
  checkPermission(perm);
  // guarded code
}
```

and not

```
if (lookup(getPrincipal(), resource,
                                 action)) {
  // guarded code
}
```

as the latter gives no opportunity for centralized policy evaluation, which may take other factors into account.

3.1 Exporting Application Attributes for Centralized Policy Evaluation

p-Contact must enforce a policy rule that only authorized users may send a message to a distribution list.

To evaluate policy for the p-Contact permission 'can post mail to distribution-list', Identity Server must have some notion of the distribution-list and its characteristics (at a minimum, the ID of the list).

This question of how to partition the access-policy evaluation between applications and middleware is investigated more fully in [10, 11]. However two main strategies present themselves in our case:

1. The `Policy` object is further customized to make decisions based on local application data, as well as policies stored in Identity Server; *or*
2. A relevant subset of application data is exported to Identity Server in a structure that Identity Server tools can work with. This puts some obligation on the application to use Identity Server APIs to export the data, but facilitates the use of out-of-the-box tools when defining policy.

It should be noted that directory servers (on which Identity Server is based) are especially adept at storing attribute information, and that the second approach is more useful if policy evaluation must include *other* applications, and makes less-technical demands on deployers.

4. How JAAS Participates in Single Sign On with the Sun Java System Identity Server

`ISPolicy` provides an implementation of `javax.security.auth.Policy` to which JAAS `AccessControlContext`s ultimately defer policy decisions. The `ISPolicy` object in turn defers policy decisions to the `Policy` service of Identity Server. The `ISPolicy` implementation is able to enforce the same policy-decisions as the default JDK `Policy` object, as well as decisions regarding `ISPermissions`.

The constructor for an `ISPolicy` is as follows:

```
/**
 * Constructs an ISPermission instance,
 * with the specified service name,
 * resource name and action name.
 */
 ISPermission(
   java.lang.String serviceName,
   java.lang.String resourceName,
   java.lang.String actions,
   java.util.Map envParams
 )
```

The `serviceName` relates to a service in Identity Server.

4.1 Coding a p-Contact Access Rule with JAAS and Sun Java System Identity Server

Our main obligation when coding flexible access-control enforcement points is to put an AccessController.checkPermission() to guard the message-sending code block. To write it naturally in JAAS:

```
public void sendMessage(Message m,
                            ContactList l){
  PContactPermission perm =
    new PContactPermission(l, "post");
  // throws exception if no permission
  AccessController.checkPermission(perm);
  /* ... code being guarded */
}
```

However, unlike the default `Policy` object, which would compare the `Permission` it construted from the policy file, with the one we supply to it programmatically, the `ISPolicy` specifically expects an `ISPermission` if it is to involve Identity Server in the evaluation. For other permissions, `ISPolicy` behaves equivalently to the default `Policy` provided by the JDK.

Fortunately, `ISPermission` suits our needs (having both a *target* and an *action* field), and we can write equivalently:

```
public void sendMessage(Message m,
ContactList l) {
  ISPermission perm =
   new ISPermission("eval", l.getId, "post",
null);
  AccessController.checkPermission(perm);
  /* ... code being guarded */
}
```

Someplace else in our code, we must authenticate our users, and associate them with an `AccessControlContext` using `Subject.doAs()` but these obligations are easily met.

5. J2EE Shortcomings

Java 2 Enterprise Edition (J2EE) extends the statndard Java edition as a platform for multi-user applications. J2EE's per-user security mechanisms predate JAAS by a few years, and has so far taken an approach which does not expose JAAS to developers.

Instead of AccessControl.checkPermission (Permission), in J2EE, we have a method:

```
boolean isCallerInRole(String role)
```

Reflecting the composite nature of J2EE, this method is named slightly differently depending on whether the enquiry is about a `ServletRequest` (web-tier), or `EJBContext` (EJB tier).

```
ServletRequest.isUserInRole(String)
EJBContext.isCallerInRole(String)
```

We may think of the container has having acquired the JAAS subject, and executing all code touched by the remote entry point inside an extended `Subject.doAs()` construction on our behalf. The effects achieved are the same, and we are relieved of the need to setup the security context – we only have to enforce the decision points using `isCallerInRole()`.

Additionally, we no longer have an API requiring us to take explicit action to authenticate the user `(LoginContext.login())` ... the J2EE container is assumed to have done this transparently.

Allowing the J2EE container to compute whether the caller actually has the role or not is consistent wiht our aims to keep policy-evaluation out of our application domain.

However, there are downsides.

EJB does not allow us to specify a target resource – only a 'role'. We might think to overload the role, such that `TVPermission("channel-5", "watch")` becomes the 'role' `"tv:watch:channel-5"`. After all, it is of little consequence to the policy evaluator if it has to do a little extra parsing.

Unfortunately, we are unable to use our overloaded role string, since the roles we must query against come from a fixed set of roles defined in a J2EE deployment descriptor!

We can only hope that this situation changes soon.

Finally, wheras in JAAS, we had static access to `AccessController.checkPermission()`, we now must have a reference to the `ServletRequest` or `EJBContext` at the point in code where we wish to enforce access policy. This makes significant intrusions into our code, requiring us at least to wrap, and then pass-around the reference to the object which can answer our question. It is possible this restriction is inherent to the J2EE resource management model, but it does seem that it could have been done differently.

5.1 JACC

The Java Authorization Contract for Containers (JACC) [3] seeks to unify the JAAS and J2EE approaches by requiring (web and EJB) containers to defer to the JAAS subsystem for policy evalution.

The Servlet or EJB container converts the role into a `Permission` object, and using an `AccessControlContext` which is aware of the principal that represents the user to the application-server, invokes `AccessControl.checkPermission(Permission)` on our behalf.

JACC specifies 5 permisions that containers will instantiate and test with a call to `Subject.doAs()` on the component-provider's behalf:

Declarative security in ejb-jar.xml

```
EJBMethodPermission
  * String ejbName
  * String methodName
  * String interfaceName
  * String[] methodParams
```

EJB code that uses isCallerInRole(String)

```
EJBRoleRefPermission
  * String ejbName
  * String roleRef
```

Declarative security in web.xml, *without* reference to transport guarantees

```
WebResourcePermission
  * String urlPatternSpec
  * String[] httpMethods
```

Declarative security in web.xml, *with* reference to transport guarantees

```
WebUserDataPermission
  * String urlPatternSpec
  * String[] httpMethods
  * String transportType
```

Servlet code that uses isUserInRole(String)

```
WebRoleRefPermission
  * String servletName
  * String roleRef
```

Unfortunately, the contract stops short of requiring containers to make the JAAS Subject available, or export the `Subject`'s principals and credentials to an `AccessControlContext`.

5.2 Sun Java System to the Rescue?

Sun Java System Identity Server [6,7] gives us a (proprietary) way to escape the limitations of the J2EE container.

A J2EE-container is first of all protected by a 'Policy Agent' [5] which acts as an interceptor of client requests, and ensures the federated identity authentication procedure is followed, and gives us access to the single-sign-on (SSO) token assigned to the caller.

The presence of the Policy Agent facilitates discovery of the SSO token associated with a (web or EJB) request. Once we have the token, we can use (non-JAAS) policy APIs [8,9] to evaluate a policy using similar arguments as encapsulated by our `PContactPermisssion`.

```
import com.sun.identity.agents.filter.\
                             AmFilterManaer;
import com.iplanet.sso.SSOToken;
import com.iplanet.sso.SSOTokenManager;
com.sun.identity.policy.client.\
                     PolicyEvaluatorFactory;
com.sun.identity.policy.client.\
                            PolicyEvaluator;


AmSSOCache ssoCache =
   AmFilter.getAmSSOCacheInstance();
String ssoToken =
ssoCache.\
       getSSOTokenForUser(getEJBContext()));

SSOTokenManager mgr =
   SSOTokenManager.getInstance();
SSOToken token =
   mgr.createSSOToken(ssoToken);
PolicyEvaluatorFactory f =
   PolicyEvaluatorFactory.getInstance()
PolicyEvaluator evaluator =
   factory.getPolicyEvaluator("");

boolean allowed = evaluator.isAllowed(
  token,
  "distribution-list-21", // resource
  "post",                 // action
  Collections.EMPTY_MAP
);

if (allowed) (
  // privileged block
}
```

So it seems that Identity Server may have somthing to offer where J2EE takes away. Most of the dirty work requiring 'foreign' APIs could be delegated to helper methods, giving us perhaps:

```
boolean allowed = MyAccessController.check(
   ejbContext,
   list.toString(),
   "post"
);
```

6. Further Work

We have only looked at a fairly trivial permission, involving only an entity in our system, and an entity in the organisation's user-directory.

We did not consider how application information is exported to Identity Server for the purposes of external policy evaluation, not did we explore the capabilities of Identity Server in this regard.

We only looked at the API for Sun Java System Identity Server, and did not show actual configurations or policy definitions, or test actual operation of our ideas (partly owing to the non-trivial systems-administration required to setup Identity Server).

We did not consider the possibility of ignoring J2EE container security, and performing JAAS login ourselves. We suppose for now that this might break integration with components that did use the container's facilities, and possibly break clustering.

We did not consider how `GuardedObject` in the Java Security Architecture might be applied to our situation.

We did not consider whether it might be possible to expose some notion of application resource identity in the URL (since JACC permissions constructed by the container can include URLs). Policy could be enforced based on extracting application-respource ids from the URL. This intrudes upon the web-application's architecture a little, but might be useful in some scenarios.

Using JAAS alone has some shortcomings when it comes to fine-grained access control. Consider the following example, which requires evaluation of access-control policy on each iteration of a loop (the example filters out distribution lists a user is not allowed to view):

```
public HTML display(List<ContactContainer>\
                             contactLists) {
  Subject.doAs( getSubject(), new
         DisplayListsAction(contactLists) );
}
public class DisplayListsAction implements
                          PrivilegedAction {
  private List<ContactContainer>\
                               contactLists;
  public
   DisplayListsAction(List<ContactContainer>
                                    lists) {
   this.contactLists = lists;
 }
```

```
 public Object run() {
   HTMLBuilder builder = new HtmlBuilder();
  for (ContactList list : this.contactLists)
{
     try {
     Permission seePermission = new
Permission( Helper.getObjectId(list), "see"
);

AccessController.checkPermission(seePermissi
on);
     builder.add(list);
     } catch (SecurityException ignore) {
     }
   }
   return builder;
 }
}
```

There are at least 3 reasons for not doing this with JAAS:

1. we create a new `Permission` each time through in the loop;
2. `checkPermission` is a potentially expensive call (especially it it consults a remote server);
3. generating and catching an exception on failure is also quite expensive.

Policy API on Sun's Access Manager is in some ways better:

```
PolicyDecision getPolicyDecision()
boolean isAllowed()
```

These don't throw exceptions, and `getPolicyDecision()` can evaluate many things in one remote call, but we could explore these questions further.

7. Conclusion

Coding flexible access-control enforcements points in a way that allows external policy-evaluation is possible in both JAAS and J2EE. In both cases, it makes no difference if identity is truly federated or merely centralized. However, not only can JAAS code not be used in a J2EE container, but the J2EE container prohibits reference to application resources when evaluating policy, even though the container may use JAAS itself.

Sun Identity Server provides services that facilitates the use of both JAAS and J2EE API's, and centralized policy evaluation. To evaluate policy with respect to application resources in the J2EE case, we can use Identity Server specific APIs.

The manner in which application-resource attributes are made accessible to the external policy-evaluator (and the programming obligations thereof) need further exploration, as do a few more ideas for escaping from the limitations of J2EE.

Disclosures

David Bullock is a freelance Java Programmer and director of Thrive Online `http://www.thriveonline.com.au`. He has no association with Sun Microsystems, and in no way profits from the sale of Sun Java System Identity Server.

References

[1] JavaTM 2 Platform Security Architecture, Version 1.2. Li Gong. http://java.sun.com/j2se/1.5.0/docs/guide/security/spec/security-spec.doc.html

[2] User Authentication And Authorization In The Java(TM) Platform. Charlie Lai, Li Gong, Larry Koved, Anthony Nadalin, and Roland Schemers. Proceedings of the 15th Annual Computer Security Applications Conference, Phoenix, AZ, December 1999. http://java.sun.com/security/jaas/doc/acsac.html

[3] Java Authorization Contract for Containers Specification 1.0, Final Release. Java Community Process. http://java.sun.com/j2ee/javaacc/

[4] R. S. Sandhu, et al. "Role-Based Access Control Models", IEEE Computer 29(2): 38-47, IEEE Press, 1996. http://citeseer.ist.psu.edu/sandhu96rolebased.html

[5] Sun ONE Identity Server Policy Agent 2.1 J2EE Agents Guide http://docs.sun.com/db/doc/816-6884-10

[6] Sun Java Enterprise System 04Q2 http://docs.sun.com/db/prod/entsys.04q2

[7] Sun Java System 04Q2 Identity Server. Sun Microsystems. http://docs.sun.com/db/coll/IdentityServer_04q2

[8] Sun Java System Identity Server 2004Q2 Developer's Guide. http://docs.sun.com/source/817-5710/index.html

[9] Sun Identity Server 04Q2 Developers Guide, Chapter 8 - Policy API. Sun Microsystems. http://docs.sun.com/source/817-5710/prog_policy.html#wp19788

[10] Konstantin Beznosov, Object Security Attributes: Enabling Application-Specific Access Control in Middleware http://citeseer.ist.psu.edu/661542.html

[11] Access Policies for Middleware. Ulrich Lang. http://citeseer.ist.psu.edu/576094.html

[12] On the Role of Roles: from Role-Based to Role-Sensitive Access Control. Xuhui Ao, Naftaly H Minsky. SACMAT'04, June 2–4,

2004, Yorktown Heights, New York, USA. http://citeseer.ist.psu.edu/692537.html

[13] BBAE – A General Protocol for Browser-based Attribute Exchange. Birgit Pfitzmann and Michael Waidner. http://citeseer.ist.psu.edu/pfitzmann02bbae.html

[14] Federated Identity-Management Protocols – Where User Authentication Protocols May Go. Birgit Pfitzmann and Michael Waidner. http://citeseer.ist.psu.edu/670544.html