Security Aware Programming in Tuscany

Security Aware Programming in Tuscany.

Introduction to Java Security

As you develop software for the **Apache Tuscany** project, it is important to be aware and maintain the security features of the Java platform. Java's success is promoted by its security architecture and relative absence from security problems. Similarly, Tuscany will benefit as a platform if the developers take care to keep it secure and free from ill-intentioned users.

This article focuses on the **security manager** and **access control** mechanisms in the Java platform and the Tuscany code base. The focus here is not on SCA applications and samples, but rather contributions that affect the Tuscany runtime including any contributions or additions such as new bindings, implementations, or data bindings. Taking advantage of the security manager and limiting access to secure areas of the code base are the primary defense against unsafe operation of Tuscany's SCA platform. And since Tuscany code is available for anyone to read, it is important to prevent any unprivileged access to the system on which Tuscany is running. This requires that developers understand application runtime and server policies and how to guard the system resources. Here is a good overview of Java Security.

By default, Java application code and the Tuscany code base run in an unsecure environment with no security manager. This gives the Java application access to all system resources. The application may read and write all system properties, open and read any system files, and do all sorts of unprotected actions. All Tuscany code will run unhindered in this environment. All malicious Tuscany users will also run unhindered in this environment. Users may not welcome this and we need to therefore think about how to provide secure code to block malicious usage.

You may turn security on by running your Tuscany application with the `-Djava.security.manager` option on the command line. The default security manager delegates access control decisions to `java.security.AccessController`. The `AccessController` determines access authority for your Java code by consulting the permissions in a `java.security.Policy` class usually specified in the default `security.policy` file.
who are you talking to here? Is this the extension developer or user running an app on Tuscany?

There is only one `Policy object` installed into a Java runtime at any given time. The default behavior for Java is to load the authorization data from one or more security policy files, but Tuscany users may add to or replace the policy by running with additional policy information on the command line. For instance `"-Djava.security.manager -Djava.security.policy=tuscany.policy -Dpolicy.allowSystemProperty=true"` will add the permissions in the tuscany.policy file to the default Java permissions. If you specify `"-Djava.security.policy==tuscany.policy"` you replace the default policy with those specified in the Tuscany policy file. When Tuscany is run by an application server (whether it be WebSphere, Geronimo, or other), the policy of the server will form the starting point for Tuscany's security policy.
What is a policy object?
Again, who is the 'tuscany user'? I am asking this because in the 2nd parag. you mentioned this is written for extension developers and not users

Each policy file will contain a list of grant statements. A grant tells the runtime where the code came from (a URL specifying the code base), who signed the code (a list of signer certificates), and what permissions are given. The permissions can be read/write permissions to the file system, access to system properties, or class loading privileges.
An example of a granting all permission to an unsigned Tuscany code base is given here:

security.policy example

```
grant codeBase "file:$/{{user.home}}/tuscany/java/sca/-" {
  permission java.security.AllPermission;
};
```

This example grant statement is quite a broad bludgeon. Namely it says that all Tuscany code has been granted all permissions. This is not much different that running without a security manager. In practice, a user policy will want much finer-grained permissions towards the Tuscany code and allow only specific pieces of the code to have privileged access. The remainder of this article shows how to locate those permissions and how to enable secure checks for them.

In summary, with security on, Tuscany class XYZ can access a secured resource only if the Is this Java security manager security manager's `AccessController` has determined the proper permissions are available in the security policy. Tuscany code that calls a protected Java API will only work with an `AccessController doPrivileged` block and the proper permissions in place. Otherwise, the Tuscany code will not run properly with security on, and it will throw `SecurityExceptions` left and right. Many times these `SecurityExceptions` will be passed back to the Tuscany runtime and then be wrappered in a `ServiceRuntimeException` and presented to the user. Not good.

Identifiying the common Java APIs That Require Security Enablement

What are some of the Java APIs that might cause your Tuscany code to produce a `SecurityException`? In the Java API Reference, any Java API that throws a `SecurityException` is a candidate.
For instance, notice that `java.lang.System.getProperty(String)` may throw a security exception. With security on, you are allowed to read some `System` properties (such as `java.version` or `os.name`) but by default you will get a `SecurityException` on other properties (such as `user.home` or `java.home`). In general, this makes sense because we do not want any intruders to have a map of the file system. A concise list of APIs with security checks is located at Methods and the Permissions They Require.

Whenever you use one of these security enabled Java APIs (any API with a `"throws SecurityException"` suffix), you must do two things to ensure your code will run with security on:

1. Add the required permission to your Tuscany or application sever security policy.
2. Add an `AccessController.doPrivileged` call block around your code.
   This section discusses identifying any necessary security permissions, and the next session shows how to write `AccessController` call blocks.

To be a little more concrete for Tuscany developers, let's go through some common API groups that they are likely to use.

1. `System.getProperty(String)`. This is quite a common item to check to make operating system or Java version specific code. If you use this API you need to add `java.util.PropertyPermission` to the security policy. For example, this permission allows the Tuscany code base to read two system properties:

   security.policy property read example

   ```
   grant codeBase "file:$/{{user.home}}/tuscany/java/sca/-" {
     permission java.util.PropertyPermission "user.home", "read";
     permission java.util.PropertyPermission "java.home", "read";
   };
   ```

2. File IO. If you create a file stream, or attempt to check if a `File` exists or is a directory, or can read or write, you will have to add `java.io.FilePermissions`.
3. ClassLoader. Anytime you access the ClassLoader via `getClassLoader()` (which is common if you want to load a resource), you will have to add a `java.lang.RuntimePermission`.
4. Introspection. Whenever you do any sort of dynamic class loading and execution, say you want to check for methods in a business interface, or you want to instantiate a data object, you are going to add a `java.lang.RuntimePermission`.
5. URLS and sockets. Any sort of network access via URL streams or sockets will have to be checked by the security manager. You will have to add `java.lang.RuntimePermission` or `java.net.SocketPermission`.
6. Threads. Any starting or stopping or checking for threads is going to be guarded and you will have to have the proper `java.lang.RuntimePermission`.
7. The Security stack. Need it be said, but if you attempt to access the `SecurityManager`, `AccessController`, `Policies`, or other parts of the security architecture you are going to need `java.security.SecurityPermission`. In other words, hands off the security system.

This list is just a general overview. Once again, any API that throws a `SecurityException` is something that your Tuscany code should be privileged and permitted to use.

How to Write Access Control Blocks.

The final step of making your code security enabled is to add `AccessController` privileged blocks around your enabled code. You must be careful here. You cannot put a privileged block with a wide scope. That is similar to running with security turned off. You also must be careful not to grant privilege code that a malicious user could use. Be very careful of granting privilege to any public API that a user can use with malicious parameters. Here is a more in-depth article.

The type of coding pattern you use is based primarily on three considerations:

- the return type of the checked call
- the input parameters to the checked call
- the exceptions thrown by the checked call.

In the simplest case, let's say you use a checked Java API that throws no exceptions other than `SecurityException`. (Recall that `SecurityException` is thrown by all of these APIs.) For instance, your brand new Tuscany "Blatz" extension needs to load the "blatz" library before it runs.
This is how you security enable the loading of the library.

AccessController doPrivileged block - No parameters

```
// ... unchecked code here...
AccessController.doPrivileged(new PrivilegedAction<Object>() {
    public Object run() {
        System.loadLibrary("blatz");
        return null;
      }
});
// ... more unchecked code here
```

Notice that the code is using the Java type safe syntax to ensure the proper casting of the method run return type. The class in the <class specification> should always match the return type class type.

In the next example, let us examine how a Java API with a boolean return type is handled. Notice that we are using the autoboxing feature of Java to convert from `boolean` to `Boolean`.

AccessController doPrivileged block - With Return Type

```
boolean isDirectory = AccessController.doPrivileged(new PrivilegedAction<Boolean>() {
        public Boolean run() {
                return rootFolder.isDirectory();
        }
});
```

Using this succinct anonymous class, you must make sure any input parameters are final instances. This is a requirement of the Java syntax. If you have a parameter that has undergone multiple assignments, you can easily create a final version of that input parameter and pass it into the block.

AccessController doPrivileged block - Input Parameters

```
URL constructedURL = protocol;
constructedURL += pathName;
constructedURL += parameters;

final URL xsdURL = constructedURL;
URL definitionsFileUrl = AccessController.doPrivileged(new PrivilegedAction<URL>() {
    public URL run() {
        return getClass().getClassLoader().getResource(definitionsFile);
     }
});
```

If your checked Java API throws an exception, you have two choices: you may catch the security exception (PrivilegedActionException) and recast it to a more domain specific exception, or you may incorporate the security exception into the signature of your method.

This example shows a recasting from the security exception (PrivilegedActionException) to a local exception such as the `IOException` shown here:

AccessController doPrivileged block - Recast Exception

```
InputStream is;
try {
is = AccessController.doPrivileged(new PrivilegedExceptionAction<InputStream>() {
    public InputStream run() throws IOException {
       return url.openStream();
   }
});
} catch ( PrivilegedActionException e ) {
   throw (IOException) e.getException();
}
```

This example shows incorporating the security exception into an existing signature of a local method. Note that the security exception (PrivilegedActionException) is not recast, but it will just be thrown along with any other exception from this method under the guise of general java.lang. Exception. If your method already has a number of more explicit exceptions, you may simply add the PrivilegedActionException to the list.

AccessController doPrivileged block - Exception thrown by method

```
 public void startBroker( Broker jmsBroker ) throws Exception {
     AccessController.doPrivileged(new PrivilegedExceptionAction<Object>() {
         public Object run() throws Exception {
             jmsBroker.start();
             return null;
         }
     });
}
```

This article shows just a few techniques that cover many of the security issues that arise in the Tuscany code base. If you are aware of the Java security architecture and how to control access to guarded resources, you can guard against malicious users who might use the Tuscany code base.