WHITE PAPER

Java

Java 2 Security

By Brian Maso

This paper explores some of the internals of Java 2 security. We'll take an inside-out approach to exploring the important concepts and APIs: First we'll look at the high-level concepts that Java 2 security is based on, and then we'll examine the Java class internals that encode these concepts. The Java classes are a highly optimized implementation of the relatively simple concepts on which Java 2 security is based. The high optimization level means that a bare reading of the source code is confusing and at times misleading, so a good understanding of the concepts and how the Java security classes map to those concepts is essential when pursuing a strong understanding of Java 2 security.

This paper wraps up with a discussion of advanced Java security techniques. We'll see how to impose Java security on non-Java interpreted scripts, how to create logical threads of execution taking into account Java 2 security, as well as other techniques.

Java 2 Security Concepts and Implementation Classes

Java 2 security is a two-part mechanism for creating security-aware Java methods. Security-aware methods can only be completed by threads that have sufficient permissions to complete the actions of the method.

The first part of this mechanism is a very simple API to create security-aware methods. A security-aware method completes some potentially dangerous operation, and so requires "guard" conditions to ensure only trusted or permitted threads are allowed to use the method. A method is made security-aware by including a preamble "guard" block. For example, consider the security requirement of a FileOutputStream object. Through its methods, a FileOutputStream object can be used to truncate and modify any file in the local filesystem. Obviously, this is a powerful object, which can have serious security implications if used maliciously. To guard against malicious activity, the FileOutputStream constructor includes a small guard condition that can be written like this:

```
public FileOutputStream(String name)
{
  SecurityManager sm = System.getSecurityManager();
  if(sm != null)
  {
    FilePermission perm = new FilePermission(name, "write");
    sm.checkPermission(perm);
  }
  ...
}
```

For historical reasons the preamble of the FileOutputStream class constructor doesn't look exactly like this. However, the code in its constructor does perform the same security check, although indirectly. If you follow the trail of execution in the FileOutputStream constructor, you would see that eventually the SecurityManager.checkPermission() method is called is the same manor indicated in the sample code above.

The checkPermission method call in the constructor returns benignly if the current thread has the given FilePermission, which represents permission to open and write to the given local filesystem file with the given name. If the current thread does not have that permission, then checkPermission throws an AccessControlException--an unchecked exception derived from RuntimeException. Thus, the guard block guarantees the calling thread can only create and use the FileOutputStream object if it has sufficient permissions. Threads without necessary permissions receive an exception, so they are unable to perform this potentially dangerous operation.

The second part of this mechanism, really the guts of Java 2 security, defines how threads are associated with Permissions. Protection domains are the key concept of this facility. A protection domain is an execution context with a set of associated permissions. When a thread enters a protection domain, the Java VM maintains an internal association between the thread and the domain. The list of all protection domains associated with a thread at a given point in time is known as the thread's access control context. Thus, the access control context is basically a collection of the permissions available to a thread at a given point in time.

The AccessController.getContext() method returns an object representing the current thread's access control context. getContext in fact returns an instance of the java.security.AccessControlContext class:

//...get the current thread's access control context from the VM... AccessControlContext acc = AccessController.getContext(); The checkPermission method in the FileOutputStream constructor code example works by accessing the current thread's access control context–the list of protection domains associated with the calling thread–and running a check to make sure the required FilePermission is a member of that context. If it is, then the checkPermission method returns without doing anything. If the required permission is not a member of the calling thread's access control context, then checkPermission throws an AccessControlException.

An AccessControlContext object is an opaque data structure. While internally the object maintains a reference to a list of ProtectionDomains, this list is not available through the context object's public interface. The context object does, however, provide its own checkPermission method that consults the permissions associated with each ProtectionDomain in the context's list. So, in theory, the SecurityManager.checkPermission method might be coded like so:

```
public void checkPermission(Permission perm)
{
 AccessControlContext acc = AccessController.getContext();
 acc.checkPermission(perm);
}
```

That's not exactly correct. The SecurityManager.checkPermission implementation actually calls AccessController.checkPermission, which in turn executes code similar to the two lines listed above. Through this short series of indirections the current thread's list of ProtectionDomains is accessed to see if the permission required by the FileOutputStream guard block is indeed a member of the current thread's access control context. The actual order of method calls is not really important, and in fact may change as the Java Core API evolves. The salient features of the Java 2 security design are:

- Security-aware classes and objects use a guard block before executing potentially dangerous code. Without this guard block, a method is security-agnostic, able to be executed by any client code at any time.
- The Java VM maintains an internal list of ProtectionDomains associated with each thread. (The next section details how this association is maintained.)
- Each ProtectionDomain includes its own list of Permissions.
- The SecurityManager.checkPermission method accesses the current thread's ProtectionDomains, represented by an aggregating AccessControlContext object, and ensures the required permission is available to the thread's ProtectionDomains.

Managing a Thread's Protection Domains

As stated above, the Java VM internally maintains a list of ProtectionDomains associated with each Java thread. The mechanism of this association is implemented natively by the VM implementors. This association happens automatically, magically, outside the Java Core classes.Each domain, in turn, includes a list of Permissions available to threads in that domain. The list of domains associated with a thread--the thread's access control context--is built by default from the set of classes the thread is currently executing within.

In Java 2, each class is associated with a single ProtectionDomain. This isn't a well-documented part of the Java VM specification--in fact, this association doesn't appear as part of the VM specification--but a little spelunking in the Core API source reveals the association. Java classes are loaded into a VM through the ClassLoader.defineClass method, whose code looks like this:

```
protected final defineClass(String name, byte[] bytecode,
                            int offset, int len)
{
 //...call overloaded defineClass with a null ProtectionDomain...
 defineClass(name, bytecode, offset, len, null);
}

protected final defineClass(String name, byte[] bytecode,
                            int offset, int len,
                            ProtectionDomain domain)
{
```

That is, no matter whether or not a ProtectionDomain is provided with a class' bytecodes, each class is assigned to at least the default ProtectionDomain.
At a point in time all you would need to do to compile a thread's access control context is to take a snapshot of the thread's stack; each class that thread is currently executing has an associated ProtectionDomain. The combined set of all ProtectionDomains is the thread's current access control context.

That is essentially what the AccessController.getContext() method does. It compiles the list of classes the current thread is executing in and from each pulls out the associated ProtectionDomain--defined at class load time. This list of ProtectionDomains is stored in a new AccessControlContext object. See Figure 1 below. In the AccessControlContext object's private ProtectionDomain[] member named

"context", though that's just an implementation detail of the AccessControlContext class that may change in later versions of Java.

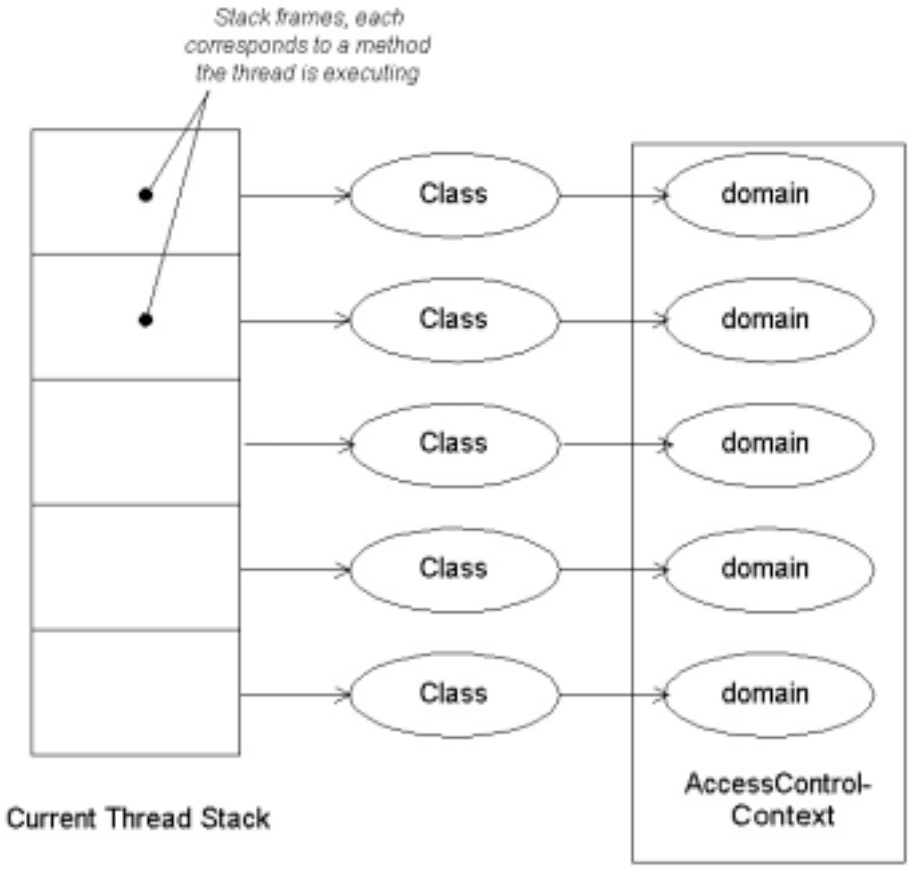

Figure 1

Execution Context + Assigned Context = Access Control Context

The access control context concept isn't that hard to understand at the level I've presented it. Basically, this context is a collection of ProtectionDomains, representing all the "domains"--security-relevant contexts--the current thread is associated with, built from the classes a thread is currently executing in.

The low-level Java security APIs and infrastructure are built around a slightly more complicated model of the access control context, though. Now we're going to flesh-out the full access control context model by illuminating some of its more complicated internal aspects, in order to understand the Java APIs that are based on the more complicated model.

The access control context is actually a composite context, the sum of two distinct contexts associated with a thread. Both of these member contexts have their own set of ProtectionDomains.

The Execution Context

I'm going to call the first member of a thread's composite access control context the **execution context**. There is no official name for this context. I use execution context because it's unambiguous and indicates how this context's ProtectionDomains are defined. This is the set of ProtectionDomains associated with all classes the thread is currently executing in, as described above. This context dynamically changes, with ProtectionDomains constantly being added and removed from it. Whenever a Java thread enters a new domain (by invoking a method of a class), the Java VM adds a

ProtectionDomain object associated with that domain to the thread's execution context. When the thread later leaves the domain (by returning from the method invocation), the Java VM automatically removes the associated ProtectionDomain from the thread's execution context.

The Assigned Context

The other member context of the composite access control context is the **assigned context**. The assigned context's ProtectionDomains are static and are not changed by the Java VM. The assigned context may be modified programmatically only, using the AccessController.doPrivileged API. This method temporarily modifies the calling thread's assigned context for the duration of a PrivilegedAction.run method call. When a thread enters a doPrivileged call, a new assigned context is associated with the thread until the method call exits. The previous assigned context is remembered when the thread enters doPrivileged, and re-assigned to the thread when doPrivileged exits.

This does not imply that a thread has no initial assigned context. Initially, before any call to doPrivileged in a thread, a thread inherits a default value to be used as an initial assigned context. This default assigned context value is known as the thread's inherited context. Upon creation, every Java thread is assigned an inherited context defined as the full (composite) access control context of the thread that created it. Defined when the thread is created, a thread's inherited context does not change during the thread's lifetime. Basically a thread's inherited context is that thread's default assigned context. When the thread is involved in a doPrivileged call, this default is overridden. Outside any doPrivileged call, this default (inherited) context is used as the assigned context. Figure 2 is a sequence diagram illustrating when a thread is associated with an inherited context and how its assigned context is defined at different points during its lifetime.

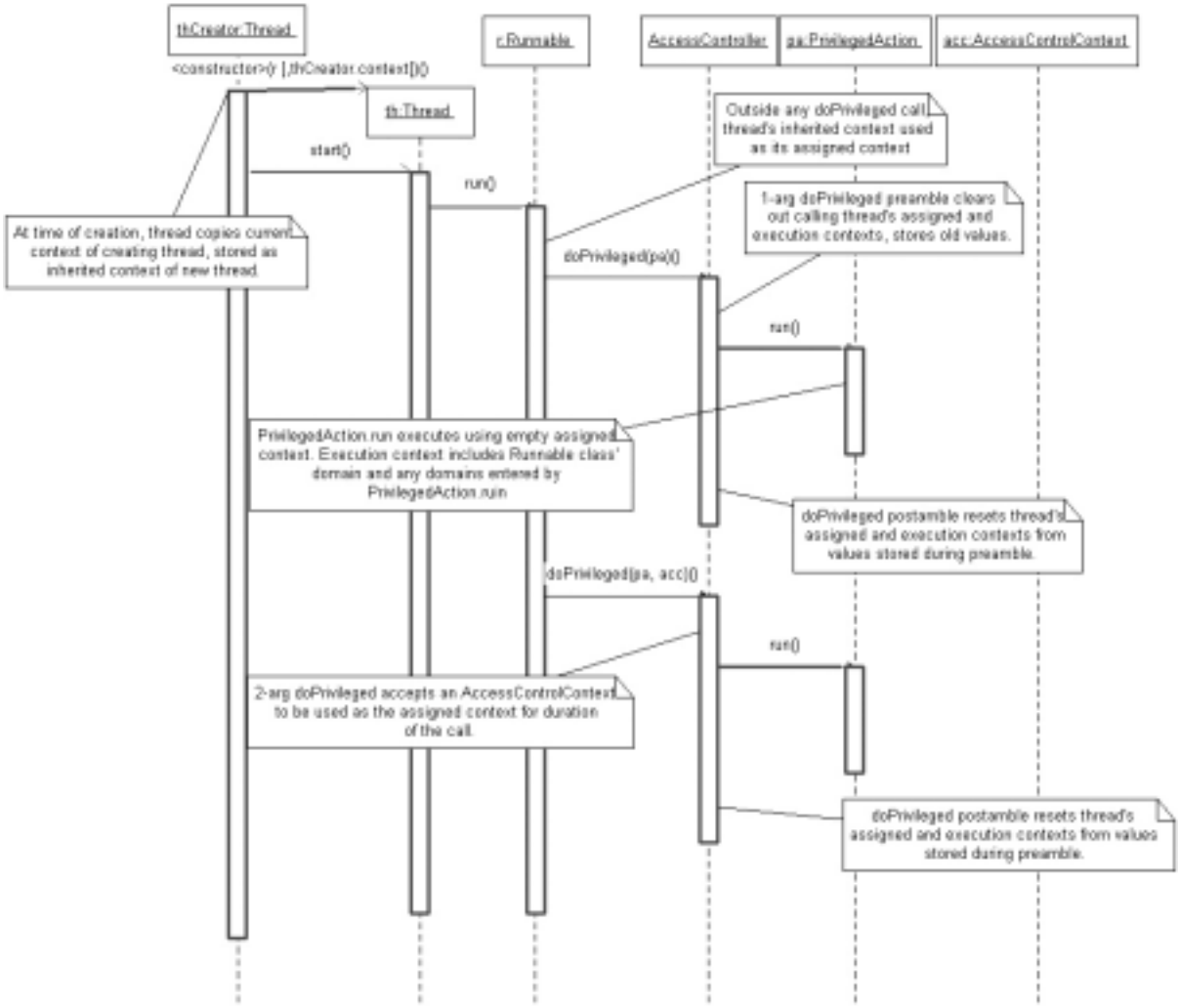
thCreator:Thread
r:Runnable
AccessController
pa:PrivilegedAction
acc:AccessControlContext
<constructor>() [,thCreator.context]()
th:Thread
start()
run()
Outside any doPrivileged call, thread's inherited context used as its assigned context
1-arg doPrivileged preamble clears out calling thread's assigned and execution contexts, stores old values.
At time of creation, thread copies current context of creating thread, stored as inherited context of new thread
doPrivileged(pa)()
run()
PrivilegedAction.run executes using empty assigned context. Execution context includes Runnable class' domain and any domains entered by PrivilegedAction.run
doPrivileged postamble resets thread's assigned and execution contexts from values stored during preamble.
doPrivileged(pa, acc)()
run()
2-arg doPrivileged accepts an AccessControlContext to be used as the assigned context for duration of the call.
doPrivileged postamble resets thread's assigned and execution contexts from values stored during preamble.

Figure 2

More AccessControlContext Internals

It is a good idea at this point to acquaint you with some AccessControlContext implementation details before continuing with Java 2 security concepts. An AccessControlContext object contains both contexts described above: both the execution and assigned contexts. The ProtectionDomains in the execution context are stored in the AccessControlContext's private *context* member. In addition, the current thread's assigned context is stored in the AccessControlContext object's *privilegedContext* member:

```
public final class AccessControlContext
{
  /**
   * execution domains
   */
  private ProtectionDomain[] context;

  /**
   * assigned domains are the "context" array of this nested
   * ACC object.
   **/
  private AccessControlContext privilegedContext;

  ...
}
```

Figure 3 illustrates the static structure of an AccessControlContext, and how it is used to represent a thread's current execution and assigned contexts.

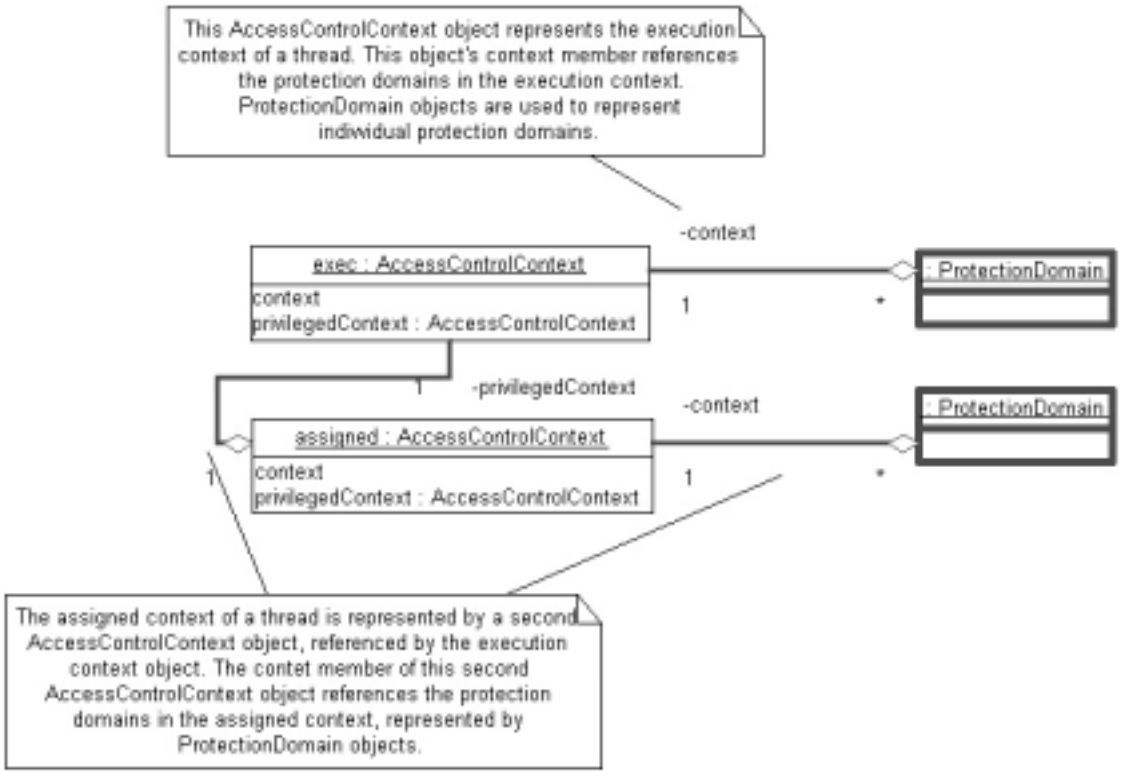

Figure 3

The private AccessController.getStackAccessControlContext method returns a new AccessControlContext object with the exact structure illustrated in Figure 3, representing the current thread's full security context. Thus an implementation of AccessController.checkPermission could look like the following. (This code will be revised in the next section. It is provided here to lay the foundation for the AccessControlContext object's structure. But several important features of this code are temporarily removed for clarity.):

```
public final class AccessControlContext
{
 ...


  public void checkPermission(Permission perm)
  {
    //...get raw execution and assigned contexts using
    //   getStackAccessControlContext
    AccessControlContext acc = getStackAccessControlContext();

    //...if there is no recorded assigned context, then use the
    //   current thread's inherited context as the assigned
context...
    if(acc.privilegedContext == null)
      acc.privilegedContext =

Thread.currentThread().getInheritedAccessControlContext();

    //...combine execution PDs and assigned PDs in to single
    //   array by simple copy...
    int execLen = (acc.context == null ? 0 : acc.context.length);
    int assignLen = (acc.privilegedContext.context == null ? 0 :
                     acc.privilegedContext.context.length);

    ProtectionDomain[] pds = new ProtectDomains[execLen +
assignLen];
    int index = 0;

    if(execLen > 0)
    {
      System.arraycopy(acc.context, 0, pds, index, execLen);
      index += execLen;
    }

    if(assignLen > 0)
      System.arraycopy(acc.privilegedContext.context, 0,
                       pds, index, assignLen);

    //...finally, check all protection domains to see if
    //   permission is available...
```

The getStackAccessControlContext method returns a "raw" context object. The context member in that object includes the ProtectionDomains associated with all classes the current thread is executing in since the most recent doPrivileged call--that is, the execution context. The nested privilegedContext reference points to another AccessControlContext captured when the most recent doPrivileged call happened. This nested AccessControlContext object is the physical encoding of the assigned context.

Combining Execution and Assigned Contexts

A ProtectionDomain's most important attribute is a set of java.security.Permissions. Taken together, the ProtectionDomains of a thread's assigned and execution contexts act as a cumulative "Permission filter". The set of Permissions associated with a Java thread at a point in time is defined as the set of all Permissions *common* to all ProtectionDomains in the thread's current combined access control context. That is, the intersection of the Permission sets of all ProtectionDomains in the thread's assigned and execution contexts.
Figure 4 illustrates these concepts: the two ProtectionDomain sets that comprise a thread's full access control context, and how the Permission sets in those ProtectionDomains are intersected, resulting in a "most common denominator" set of Permissions available to the thread.

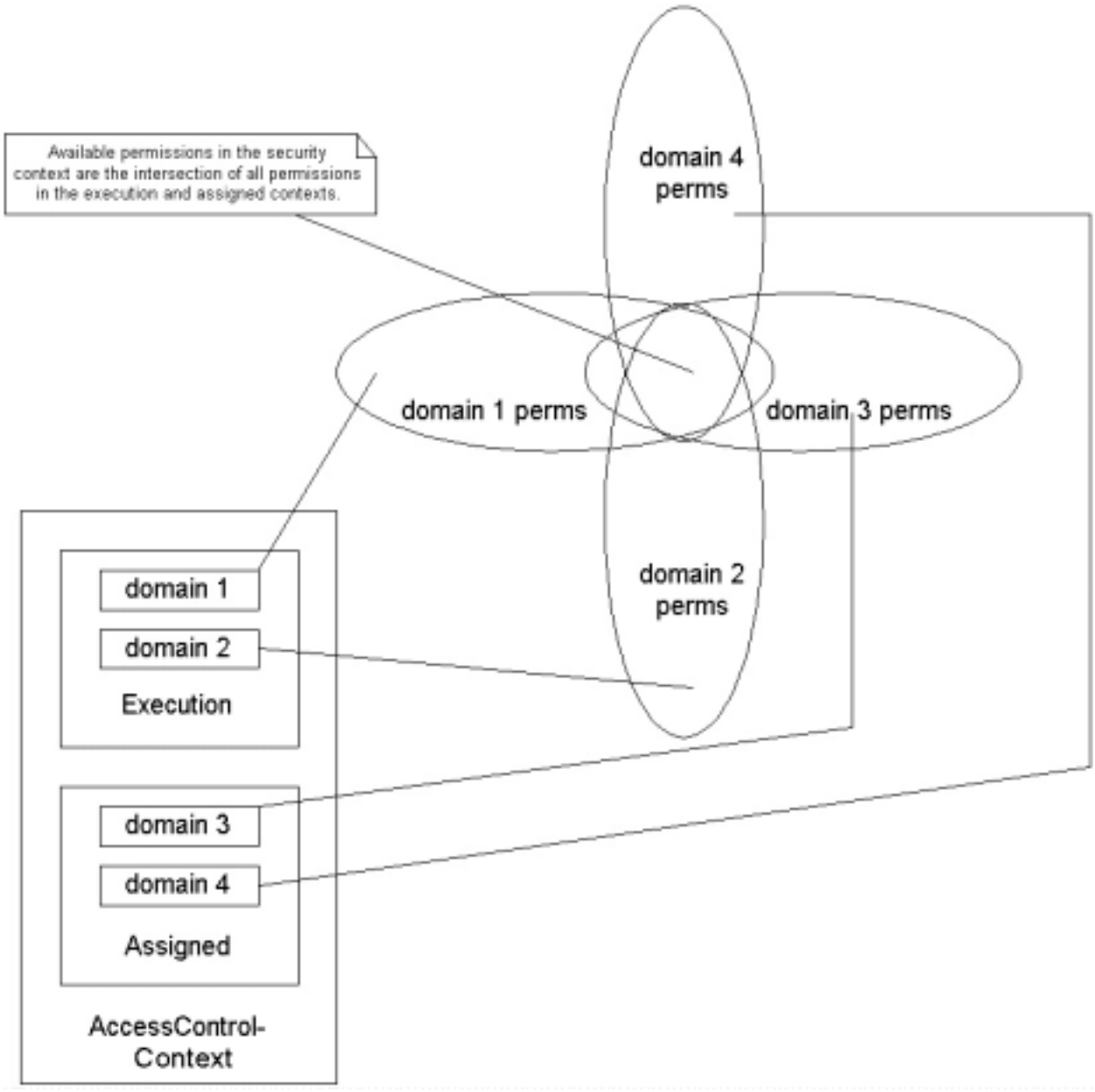

Figure 4

The execution context at a point in time obviously may contain redundant references to ProtectionDomains. For example, if a thread executes method **a** in class **Foo**, which in turn calls method **b** in class **Foo**, then the thread's execution context will include two references to the same ProtectionDomain during the execution of **Foo.b**. Similarly, an execution context may also contain references to ProtectionDomains that are also members of the current assigned context.

You would definitely want to "optimize" a thread's access control context by taking out redundant references to ProtectionDomains before actually trying to use the context for security checks. The checkPermission pseudo-code above is going to work a lot more efficiently if redundant ProtectionDomains are culled from the "pds" ProtectionDomain[]. That is, you would want the array to be as short as possible, so culling redundant references would be in order. The pseudo-code for the AccessController.checkPermission method given above leaves out any such optimization for clarity.

The AccessController.checkPermission method does perform an optimizing combination of assigned and execution contexts dynamically. The method takes a snapshot of a thread's execution and assigned contexts and combines them in such a way that redundancies are removed by using a DomainCombiner object. A combiner object implements the DomainCombiner interface and has just one job in life: to combine execution and assigned contexts in an optimizing manner. The DomainCombiner interface looks like this:

```
public interface DomainCombiner
{
  public ProtectionDomain[] combine(
      ProtectionDomain[] executionContext,
      ProtectionDomain[] assignedContext);
}
```

The most obvious implementation of this method simply combines the execution and assigned contexts, removing redundancies. Here is one possible implementation:

```
public class SimpleCombiner implements DomainCombiner
{
  public ProtectionDomain[] combine(
      ProtectionDomain[] executionContext,
      ProtectionDomain[] assignedContext)
  {
    int execLen   = (executionContext == null ? 0 :
                     executionContext.length);
    int assignLen = (assignedContext == null ? 0 :
                     assignedContext.length);

    //...if either context is null or empty, return the other
    //   one...
    if(execLen == 0)
      return assignedContext;
    if(assignedLen == 0)
      return execContext;

    //...collect all domains in to a Set object, which ensures
    //   uniqueness of objects...
    Set set = new HashSet();

    for(int ii=0 ; ii
```

Each AccessControlContext object has a reference to a DomainCombiner (which may be null). As it turns out, only AccessControlContexts representing an assigned context can have an associated combiner. AccessControlContext objects representing execution contexts (i.e., those "raw" AccessControlContext objects created by the AccessController.getStackAccessControlContext method) never have a combiner. Figure 5 illustrates the static structure of "raw" AccessControlContext objects returned by AccessController.getStackAccessControlContext, in which you can see the "combiner" member of the outer AccessControlContext object is always null, while the nested AccessControlContext (representing the thread's current assigned context) may have a non-null reference to a DomainCombiner.

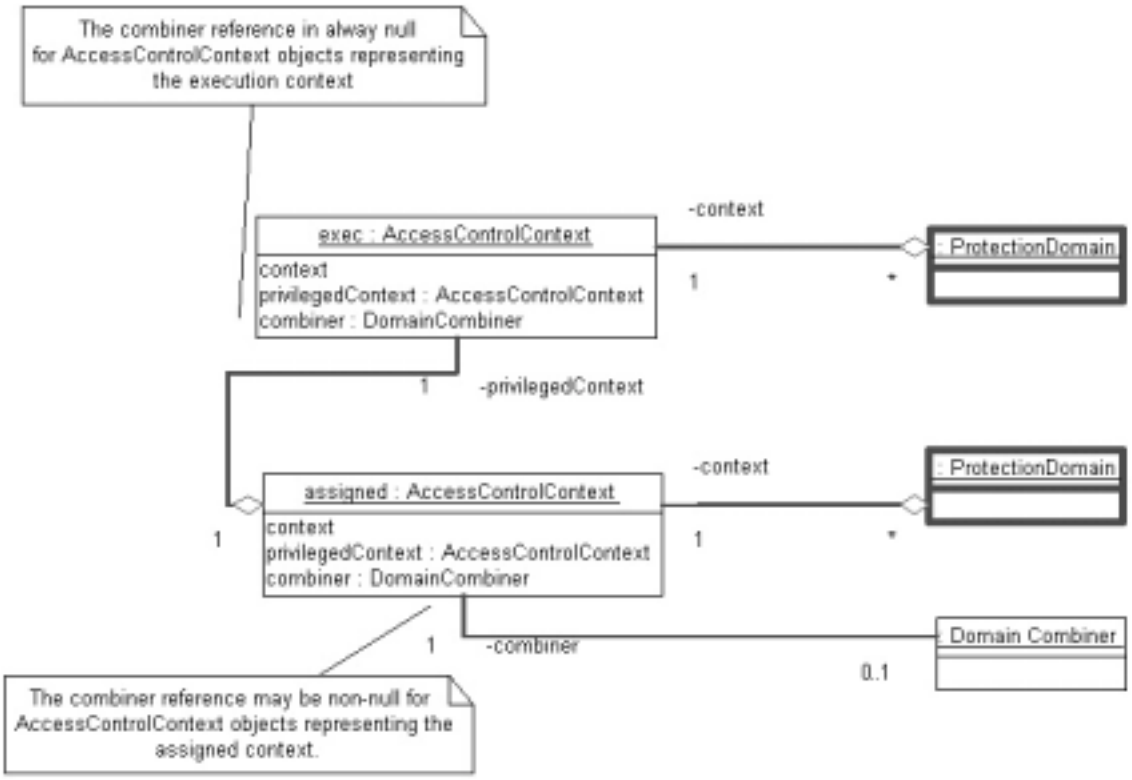

Figure 5

The AccessController.checkPermission and AccessController.getContext methods both start out the same: copying the "raw" AccessControlContext (with separate execution and assigned contexts) using getStackAccessControlContext, and then combining those contexts in an optimizing manner, like so (some details elided for clarity, and some of the code is reorganized also for clarity):

```
public class AccessController
{
  public void checkPermission(Permission perm)
  {
    AccessControlContext acc = getStackAccessControlContext();
    ProtectionDomain[] combined = null;

    //...if there is no assigned context, use thread's inherited
    //   context as the assigned context. (In fact, the actual
    //   test is a little more complicated, but this code hits
    //   the major points. See core source code if you need
    //   to see the bare naked truth.)...
    if(acc.privilegedContext == null)
      acc.privilegedContext = getInheritedAccessControlContext();

    //...if assigned context has a combiner, use it to
    //   combine the execution and assigned context
    //   ProtectionDomain[] arrays.
    if(acc.privlegedContext != null &&
       privilegedContext.combiner != null)
      combined =
acc.privilegedContext.combiner.combine(acc.context,
                    acc.privilegedContext.context);

    //...if there is no combiner, use simple algorithm for
    //   combining contexts...
    else
      combined = ...; // See SimpleCombiner above for algorithm

    //...check for presence of the permission in all
    //   ProtectionDomains in the combined array. Iteration
    //   left out for brevity...
    ...
  }

  ...
}
```

That is, if no DomainCombiner is present, then a block of code implementing the basic combiner algorithm illustrated above by my SimpleCombiner class is used.

Assigning Permissions to ProtectionDomains

I've been carefully avoiding one crucial part of the Java 2 security system: how permissions are assigned to protection domains. The assignment of permissions to protection domains is actually an open task, left up to Java ClassLoaders. The ClassLoader base class does not proscribe a mechanism for defining this association. ClassLoader subclass instances are left with the task of providing a ProtectionDomain instance as part of a class definition. Recall that ClassLoader subclasses define classes using one of these two overloaded, inherited defineClass methods:

```
public abstract class ClassLoader
{
  ...

  /**
   * Associates "default" protection domain to the new class.
   **/
  protected final Class defineClass(String name, byte[] bytecode,
                                    int offset, int length)
  {
    defineClass(name, bytecode, offset, len, null);
  }

  /**
   * Associates subclass-provided protection domain to the new
   * class.
   **/
  protected final Class defineClass(String name, byte[] bytecode,
                                    int offset, int length,
                                    ProtectionDomain domain)
  {
    ...
  }

  ...
}
```

So subclasses must either use the "default" protection domain, or define a new one. It's easy to define a new protection domain because the ProtectionDomain class has a simple public constructor:

```
public class ProtectionDomain
{
  public ProtectionDomain(CodeSource source,
                          PermissionCollection perms);
}
```

The java.security.Policy singleton class is provided to make a ClassLoader subclass' life easier. The java.security.Policy class is just a glorified lookup table, able to provide a set of Permissions for a given CodeSource:

```
public abstract class Policy
{
  /**
   * get singleton Policy object
   **/
  public static Policy getPolicy();

  /**
   * Get the set of permission associated with a CodeSource.
   **/
  public PermissionCollection getPermissions(CodeSource source);
}
```

The java.security.SecureClassLoader base class, the parent of URLClassLoader and ultimately a superclass of all ClassLoaders created by the Core Java classes, uses the singleton Policy object to assign ProtectionDomains to new classes. Here's an except from the SecureClassLoader class to illustrate the mechanism (details and exceptions removed for clarity):

```
public abstract class SecureClassLoader extends ClassLoader
{
 /**
 * Overloaded defineClass to be used by subclasses.
 **/

  public Class defineClass(String name, byte[] bytecode, int
offset,
                           int length, CodeSource source)
  {
    PermissionCollection perms =
        Policy.getPolicy().getPermissions(source);
    ProtectionDomain domain = new ProtectionDomain(source,
perms);

    //...invoke superclass defineClass to load the class,
associated
    //   with the given protection domain...
    return defineClass(name, bytecode, offset, length, domain);
  }

  ...
}
```

Figure 6 is a combination collaboration and class diagram illustrating how the Policy object is used to assign a protection domain to all classes defined by SecureClassLoader subclasses.

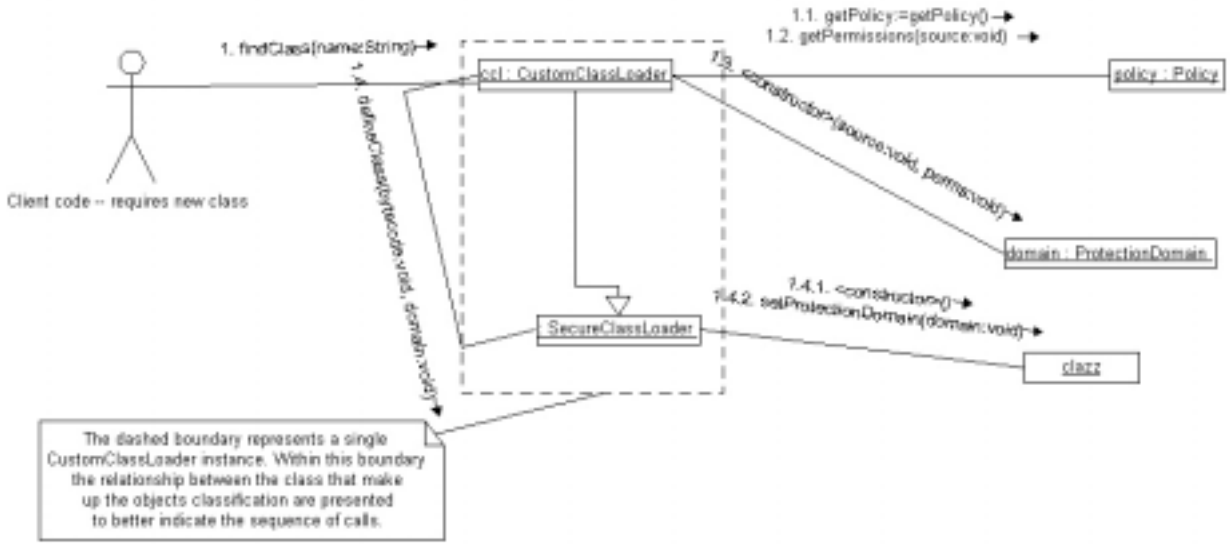

Figure 6

The Default Policy Implementation

You may already be familiar with the default Policy implementation in Java 2. This implementation is file-based, using the so-called "policy" files referred to in most Java 2 security tutorials. Policy files include grant blocks, where each block is distinguished by a codeBase and/or zero or more signedBy qualifiers. The grant contents list a set of Permissions. This default Policy implementation's getPermissions method finds all grants in active policy files matching the input CodeSource, returning a PermissionCollection containing Permissions in those matching grants.

For example, putting this grant in an active policy file will make the default Policy implementation's getPermissions method return a PermissionCollection containing the FilePermission for any CodeSource whose source URL matches "file:/java/classes/-" (that is, whose URL begins with "file:/java/classes/"):

```
grant codeBase "file:/java/classes/-"
{
  permission java.io.FilePermission "/tmp/-" "read, write,
delete";
};
```

You can easily override the default Policy implementation, providing an implementation of your own devising, using the "java.policy.provider" system property. For example, you may want a Policy that reads its grants from a database, or that uses an XML format, instead of Java's somewhat cumbersome grant-based "policy" files. Defining a custom Policy implementation is relatively simple, but it is also beyond the scope of this paper. (Read the java.security.Policy javadocs for more on designing custom Policy implementations.)

The Default ProtectionDomain

Classes loaded using the 4-argument overloaded ClassLoader.defineClass method are automatically assigned to the default (or null) ProtectionDomain. The first time a class is defined using the default domain, the ClassLoader base class consults the singleton Policy object to obtain the default ProtectionDomain, like so (some details and synchronization simplified for clarity):

```
public abstract class ClassLoader
{
  private ProtectionDomain defaultDomain = null;

  ...

  protected final Class defineClass(String name, byte[] bytecode,
                                    int offset, int length,
                                    ProtectionDomain domain)
  {
    ...

    if(domain == null)
      domain = getDefaultDomain();

    ...
  }

  private static synchronized ProtectionDomain getDefaultDomain()
  {
    if(defaultDomain == null)
    {
      CodeSource nullSource = new CodeSource(null, null);
      PermissionCollection perms =
```

```
          Policy.getPolicy().getPermissions(nullSource);

      defaultDomain = new ProtectionDomain(nullSource, perms);
    }

    return defaultDomain;
  }

  ...
}
```

Modifying and Recording Contexts at Runtime

Now I will show you how to take use hooks in the Java 2 security API to implement non-trivial security structures in your Java apps. The simplest hooks in to the Java 2 security API allow you to record and modify a thread's assigned context. With this API you can

- Suspend some of a thread's protection constraints temporarily to perform services the thread wouldn't normally be able to execute. For example, applet code generally cannot modify system properties. But a trusted class invoked from an applet might need to perform this operation. The trusted class can temporarily suspend the executing thread's more restricted protection domains in order to perform the operation on behalf of the sandboxed applet.
- Add constraints to a thread by adding protection domains. This is most necessary when executing third-party scripts or other non-Java code. For example, a script engine interpreting arbitrary script would want to restrict the interpreter thread so that untrusted script doesn't execute with the engine's own (more permissive) protection domain. This is not the same problem as adding protection domains to execute third-party Java code, such as applets within a browser or EJB within an EJB container. The Java applets or EJB are Java classes, which will have protection domains associated at load time. Dynamically interpreted scripts will not have associated Java classes, so you must physically add protection domains for the script source to an interpreting thread at runtime.
- Create logical threads of execution (not necessarily the same as a physical Java thread). With this feature you can force a Java thread to take on a pre-recorded protection domain to perform "out of band" operations. For example, class loaders use this feature during class loading to execute static initializers without incidental security restrictions that happen to be associated with the loading thread.

Temporarily Suspending Protection Domains

In general, when a third-party component is running inside a hosted environment, the component runs with tight security constraints. While there may be many operations the host environment wants to restrict the component from executing directly, the host still needs those operations to be available to the host's classes executing in the component's thread.

For example, an applet running inside a browser; the applet class is associated to a protection domain with very few permissions. The applet's init, start, stop and destroy methods thus have very few permissions available to them. This prevents the applet from opening local filesystem files, for example. However, you might want browser classes running in the applet thread to be able to open certain files on behalf of the applet. For example, the host browser may want to log TCP communications between the applet and its original server to a file. The logging code would be executed by the applet thread, so you would want to suspend the applet thread's association with the applet's protection domain temporarily while executing the logging code.

Another example is EJBs hosted inside an EJB container. EJBs usually run in a very restricted protection domain that prevents them from creating new threads. (Thread management is considered a container

responsibility--allowing hosted EJB objects to create threads could hamstring the constainer's ability to manage resources and preserve scalability.) However, several facilities require thread creation that must be made available to EJB objects. For example, if an EJB object posts a JMS message to a JMS queue or topic. In this case the JMS code invoked by the EJB object might need to create a background thread in order to asynchronously send the JMS message. The JMS code would need to temporarily suspend the association to the EJB object's protection domain in order to execute properly.

Figure 7 illustrates the security problem with hosted components. The thread in this illustration enters the hosted component's (applet's or EJB's) protection domain automatically when it is used to invoke any of the hosted component's methods. The hosted component then invokes host container code, basically requesting that the container perform some operation on behalf of the component. The host's code must suspend the thread's current list of protection domains, which includes the hosted component's more restricted domain, in order to complete the operation on behalf of the component.

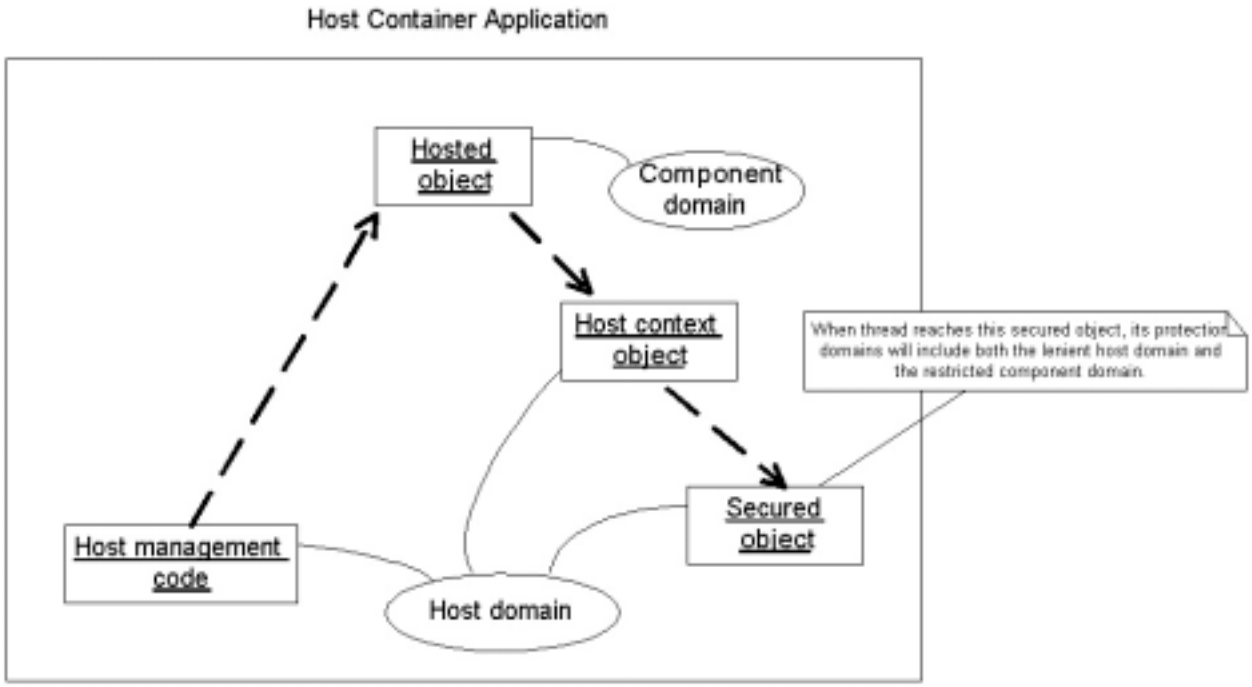

Figure 7

The single-argument AccessController.doPrivileged method is used to suspend the assigned and execution context of a Java thread for the duration of one PrivilegedAction.run call. This is the mechanism you use in host container code to temporarily suspend the association of a thread to a hosted component's protection domains.

In Java code the pattern has three aspects:

1. The hosted component's protection domain restricts the component from doing some operation directly. For example, a third-party EJB object is restricted from creating new threads because its protection domain does not include the java.lang.RuntimePermissions ("modifyThread" and "modifyThreadGroup") required to execute the Thread class constructor.
2. The host component has a reference to a host object whose own protection domain does include the required permission. For example, when an EJB object posts a JMS message it does so through a MessageQueue or Topic object received through a series of JNDI and object accessor calls. The queue or topic object may need to create new threads, even if the EJB object controlling it is not allowed to.
3. The host object method invoked by the hosted component wraps its privileged operation code with a AccessController.doPrivileged call. In the EJB example, the JMS MessageQueue or Topic that must create a new Java thread would do so in a PrivilegedAction.run method implementation executed through AccessController.doPrivileged.

The single-argument doPrivileged call suspends the calling thread's current assigned and execution contexts. For the duration of a PrivilegedAction.run call the thread uses an empty assigned context, and the thread's execution context includes just the invoking class' and PrivilegedAction class' protection domains.

The following code snippets from an EJB object class and a JMS MessageQueue class demonstrate the use of the single-argument doPrivileged call.

```
// Hosted EJB object code
public class MyEJBObject implements SessionObject
{
 ...   /**
 * This method must post a message to a JMS queue as
 * part of its execution.
 **/
 public void postsMessage()
 {
 //...obtain reference to MessageQueue through series
 //   of JNDI and JMS calls...
 MessageQueue mq = ...;   mq.postMessage(text);   ...
 }   ...
}  // Host MessageQueue code suspends call's protection
// domains in order to create new thread.
public class HostMessageQueue implements MessageQueue
{
 ...
   public void postMessage(TextMessage text)
  {
                 AccessController.doPrivileged(
                   new PrivilegedAction() {
                     public Object run() {
      //...within PrivilegedAction.run the thread's assigned
context
      //   includes only the protection domain of the invoking
      //   class: HostMessageQueue's domain in this example. The
      //   execution context includes just the protection
      //   domain of this nonymous inner class...
      Thread t = new PostingThread(this, text);
      t.start();
```

```
                    }
                  }
                });
  }

  ...
}
```

Adding Dynamically-defined Protection Domains to a Thread

The problem: How to add restrictive protection domains to a thread? My favorite example for when this might be necessary is in script interpreters. A scripting engine dynamically interprets scripts (JavaScript, custom script language, etc.) and executes whatever method calls are indicated by that script. The script is third-party provided code, but there is no Java class--just a text file. How to "inject" a protection domain into the executing thread so that scripts cannot, for example, overwrite important files or do other malicious actions?
The script interpreter is in effect a host environment, and the scripts are hosted components. We want to add a protection domain containing only those permissions we want to assign to the hosted script in order to "sandbox" the script. This is done using the two-argument overloaded AccessController.doPrivileged method:

```
public class AccessController
{
  public Object doPrivileged(PrivilegedAction action,
          AccessControlContext hostedComponentDomains);
}
```

Imagine a very simple script interpreter that reads a text file containing a series of Java class method calls in text form. The interpreter would read each line of the script and execute the named method calls through Java Reflection. Here is one sample script file (potentially dangerous because it truncates the Unix password file):

```
class: java.io.FileOutputStream; method: (java.lang.String);
    args: "/etc/passwd";
```

Assuming that the interpreter code runs with a very high level of privileges, a naïve interpreter implementation would blithely interpret this file, and end up truncating the Unix password file. Such a naïve implementation might look something like this (many implementation details left out for clarity):

```
public class NaiveInterpreter
{
  public static void main(String[] args)
  {
    BufferedReader input = new BufferedReader(
            new InputStreamReader(System.in));

    try
    {
      for(;;)
      {
        //...read class name, method name, method arg types
        //   and arg values...
        String line = input.readLine();
        String className = parseClassName(line);
        String methodName = parseMethodName(line);
        Class[] methodArgTypes = parseMethodArgTypes(line);
        Object[] methodArgs = parseMethodArgs(line);
```

```
        //...use reflection to find method or constructor to
use...
        Class clazz = Class.forName(className);

        Method method = null;
        Constructor ctor = null;
        if(methodName.equals(""))
          ctor = clazz.getConstructor(methodArgTypes);
        else
          method = clazz.getMethod(methodName, methodArgTypes);

        //...invoke method or constructor -- only static methods
        //   assumed...
        if(ctor != null)
          ctor.newInstance(methodArgs);
        else
          method.invoke(null, methodArgs);
      }
    } catch (Exception e)
    {
      e.printStackTrace();
    }
  }
}
```

A security-aware interpreter will wrap the actual method or constructor invocations with AccessController.doPrivileged calls, temporarily associating the thread with a more restrictive protection domain to ensure security policy is preserved. The interpreter will first obtain a reference to an AccessControlContext object containing only the protection domains we want the script code to run under--we can use the Policy singleton object to cache the default ProtectionDomain, for example:

```
public class SecureInterpreter
{
  static AccessControlContext scriptDomains;
  static
  {
    CodeSource nullSource = new CodeSource(null, null);
    PermissionCollection perms =
            Policy.getPolicy().getPermissions(nullSource);
    ProtectionDomain domain = new ProtectionDomain(nullSource,
perms);

    scriptDomains = new AccessControlContext(
            new ProtectionDomain[] {domain}); // 1-element array
  }

  ...
}
```

The SecureInterpreter will wrap all script method invocations using the 2-argument AccessController.doPrivileged. The AccessControlContext passed as the second argument will be used as the calling thread's assigned context for the duration of the PrivilegedAction.run method call:

```
public class SecureInterpreter
{
  ...

  public static void main(String[] args)
  {
    BufferedReader input = new BufferedReader(
    new InputStreamReader(System.in));

    try
    {
      for(;;)
      {
        //...read class name, method name, method arg types
        //   and arg values...
        ...

        //...use reflection to find method or constructor to
use...
        ...

        //...invoke method or constructor -- only static methods
        //   assumed...
        if(ctor != null)
               AccessController.doPrivileged(
                 new PrivilegedAction() {
                   public Object run() {
          ctor.newInstance(methodArgs);
                     return null;
                   }
                 }
               }, scriptDomains);  // use script domains
                                   // as assigned context
        else
               AccessController.doPrivileged(
                 new PrivilegedAction() {
                   public Object run() {
          method.invoke(null, methodArgs);
                     return null;
                   }
                 }
               }, scriptDomains);  // use script domains
                                   // as assigned context
      }
    } catch (Exception e)
    {
      e.printStackTrace();
    }
  }
}
```

Sidebar: A Better Scripting Engine Design Choice

Another interpreter implementation choice uses generative programming, which both avoids the requirement for this kind of context tweaking, and also enjoys vastly improved performance. Generative scripting engines will compile scripts into actual Java classes on the fly. The classes will probably run much faster than the interpreted equivalent. In addition, the generated Java classes will

automatically be associated with a protection domain, so any threads executing generated class code will automatically be restricted by the class' protection domain.

The best example of a generative system is J2EE's JSP engine. The JSP engine automatically compiles JSP files into equivalent Java classes. The generated classes run much faster than a text interpreter ever could, and you can also easily "sandbox" the generated classes so that JSP code can't do anything malicious/accidentally detrimental to the JSP container or associated resources.

Personally, I think generative script engines are an all-around better design choice than text interpreters, but they are also a lot harder to implement, requiring advanced use of class loaders as well as a third-party class generation tool or API. The design of generative script engines is beyond the scope of this paper, but definitely worth looking into if you are considering script features for your Java project.

Security for Logical Threads of Execution

The previous example demonstrates how to use the 2-argument AccessController.doPrivileged method to temporarily define a thread's assigned context in order to restrict that thread's security. The same method can be used to augment or to completely change a thread's access control context temporarily. This is most useful when implementing logical threads of execution.

A logical thread of execution is a sequence of operations that is executed in order, but is not actually executed using the same physical Java thread. Logical threads of execution are usually implemented by a pool of worker threads and queues, executing many asynchronous tasks in an efficient manor.

In the general case, an initial thread begins work on a particular task, and then suspends execution of that task--the remainer to be completed later. Usually the suspension is a domain requirement, for example, if the application must wait for a response from some other facility where a long turn-around time is anticipated. In such cases you can greatly reduce the number of physical threads as well as increase the system efficiency by pooling threads and reusing available threads to execute individual segments of a logical operation.

My favorite example from Java is the problem of class loading, specifically static intialization of loaded classes. A class loader is initially created by one thread in a VM. Once the loader is created, that creating thread may go to sleep, go do some other long-running task or simply die. There's really no reason to expect the creating thread to interact with the ClassLoader instance again. However, the *task* of the loader object--loading Java classes and initializing them--is not completed. In Java, classes are not usually loaded until they are required by a thread, that is, until a thread actually attempts to execute a method, create an instance or access a static field of a target class. The target class would only be loaded when the first such access occurs.

At class load time the VM must execute the class' static initializer. (The static initializer method is a hidden static method tucked away inside the class. This is the method that actually initializes static fields and performs other load-time activities of the class.) But which thread should be used to execute the static initializer? Certainly not the thread that actually created the class loader--that thread may be busy or dead when the class is actually loaded.

In fact, the requesting thread is used to load required classes and run their static initializers. In effect the ClassLoader object represents a logical task that is accomplished by multiple different worker threads over time.

There is an obvious security problem with this setup when taking into account Java 2 security: A requesting thread that is used to load a required class may not be able to complete the class' static initializer because of the security restriction imposed by the thread's access control context at load time. In fact, that thread's current restrictions may prevent it from even opening the class file. The security context of loading threads is completely unknown. The thread may be executing from a very

restricted protection domain, or maybe not--the security context of a thread when it requires a class to be loaded is completely non-deterministic.

One solution to this problem is to use the 1-argument AccessController.doPrivileged method to temporarily suspend the loading thread's access control context while the loader accesses the class file and runs the class' static initializer. That would negate the restrictions of the calling thread's security context and only subject the static initializer to run under the security constraints of the newly loaded class' own protection domain.

But if you think about it, that's not the safest way of doing things. That system might allow the loading thread to access class files that it really shouldn't have access to. What you really want is to impose not only the class' own protection domain on the loading thread, but also the protection domains of the thread that created the class loader at time of loader creation. Remember that the loading process is essentially a suspended operation, started initially by the thread that created the loader, and then suspended until later. The best solution allows the loading thread to "impersonate" the original thread that create the loader. This would complete the logical thread of execution fiction, since the loading thread will have all the same attributes as the creating thread.

The pattern for passing on the security attributes of a logical thread of execution between different physical threads has these three aspects:

1. The task to be completed by multiple, disparate threads is represented by a Java object that records as part of its state the status of the task. (This is as opposed to a task completed by a single physical thread of execution, which can store task status in local variables.)
2. An original thread begins the task by creating and interacting with the task object. Prior to initial suspension, the task object records the initial thread's AccessControlContext. This ACC is the security context to be passed on to other physical threads later as they continue to execute the task.
3. When the task is later continued by other threads, the task object wraps later execution steps (executed by other physical threads) in calls to the 2-argument AccessController.doPrivileged method, using the previously recorded AccessControlContext as the second argument--the assigned context to attach to the thread temporarily while it completes a portion of the task.

We can see all three aspects of this pattern by looking at the URLClassLoader source code:

```
public class URLClassLoader extends SecureClassLoader
{
  /**
   * Record of logical thread's security context
   **/
  private AccessControlContext acc;

  ...

  public URLClassLoader(URL[] urls)
  {
    ...

    //...record this thread's security context to be passed on
    //   to threads later when they are used to load classes...
    acc = AccessController.getContext();
  }
```

```
  /**
   * This is the method that is called when this loader must
   * load a class. Make sure to pass the recorded security
   * context on to the calling thread before calling
   * defineClass, so that this thread's context will be
   * suspended and logical thread's used during execution
   * of the new class' static initializer.
   *
   * Note: Several details elided for clarity. See actual source
   *       if you need to fully remove the veil.
   **/
  protected Class findClass(String name)
  {
    Class clazz = (Class)AccessController.doPrivileged(
                    new PrivilegedAction() {
                      public Object run() {
      URL u = ...; // Use URLs passed to constructor to
                   // find required resource file.
      CodeSource cs = ...; // URL and digital signatures
                           // used to sign the class file, if
any.
      byte[] bytecode = readURLContentToByteArray(u);

      //...call SecureClassLoader superclass defineClass...
      return defineClass(name, bytecode, 0, bytecode.length, cs);
                      }
                    }, acc); // run using recorded context,
                             // which suspends calling thread's
                             // for the loading process.

    return clazz;
  }

  ...
}
```

Summary

Java 2 security is built in to every Java 2 VM. Java 2 security automatically associates a set of protection domains with each running Java thread. In my opinion this is the best part of Java 2 security: the fact that you don't have to do anything. The VM already manages contexts and threads associations, so there's no code for you to write.

Classes "secure" themselves from malicious client code by adding guard blocks to methods or constructors. The guard block includes a single call to SecurityManager.checkPermission. checkPermission in turn ensures a required permission is available to all protection domains associated with the current thread. Thus third party code running within restricted protection domains cannot directly nor indirectly perform potentially dangerous operations.

A protection domain is defined by a CodeSource--an indicator characterizing a class' origin--and a collection of Permissions. The assignment of permissions to domains is generally done using the Policy singleton object, a convenience mechanism built in to the Java 2 core API.

The security context of a Java thread at a point in time is the optimized combination of the thread's assigned context and its execution context. The execution context includes all protection domains the thread has entered since entering the most recent AccessController.doPrivileged invocation. The assigned context is a static set of protection domains constructed by the doPrivileged method. The

single argument doPrivileged implies the assigned context is empty, and the 2-argument doPrivileged method allows you to enlist any arbitrary context as the assigned context for the duration of the doPrivileged call. The thread's inherited context is used as the assigned context if the thread is not involved in a doPrivileged method.

Using the 1-argument and 2-argument doPrivileged methods a hosting container can effectively sandbox interpreted non-Java scripts, create "privileged" operations allowing host classes to execute safely within hosted component threads, and create logical threads of execution. In all three cases host code dynamically manages thread assigned context to attach stricter, less strict, or completely unrestricted contexts to a Java thread.