CHAPTER 14

Security

Security plays an important role in distributed systems. All parts of a Jini djinn, which consists of clients, services, and lookup services, can be subjected to attack by hostile agents. You could trust everyone, but the large number of attacks that are made on all sorts of systems by both skilled and unskilled people doesn't make this a reasonable approach.

The Jini security model is based on the JDK 1.2 security system. This chapter deals with handling permissions granted to downloaded code and applies to Jini 1.1 and Jini 2.0. The advanced security for Jini 2.0 is considered in Chapter 22.

Getting Going with No Security

Security for Jini is based on the JDK 1.2 security model, which makes use of a `SecurityManager` to grant or deny access to resources. All potentially dangerous requests, such as opening a file, starting a process, or establishing a network connection, are passed to a `SecurityManager`. This manager will make decisions based on a security policy (which should have been established for that application) and either allow or deny the request.

A few of the examples given so far in this book may work fine without a security manager, but most will require an appropriate security manager in place, or RMI will be unable to download class files. The major requirement in most examples is for the RMI runtime to be able to download class files to instantiate proxy objects. You can install a suitable manager by including this statement in your code:

```
System.setSecurityManager(new RMISecurityManager());
```

This should be done before any network-related calls, and it is often done in the `main()` method or in a constructor for the application class.

The security manager will need to make use of a *security policy*. This is typically given in policy files, which are in default locations or are specified to the Java runtime. If `policy.all` is a policy file in the current directory, then invoking the runtime with the statement

```
java -Djava.security.policy="policy.all" ...
```

will load the contents of the policy file.

A totally permissive policy file can contain

```
grant {
    permission java.security.AllPermission "", "";
};
```

This will allow all permissions and should *never* be used outside of a test and development environment—and, moreover, one that is insulated from other potentially untrusted machines. (Stand-alone is good here!)

The big advantage of this permissive policy file is that it gets you going on the rest of Jini without worrying about security issues while you are grappling with other problems.

Why AllPermission Is Bad

Granting all permissions to everyone is a very trusting act in the potentially hostile world of the Internet. Not everyone is "mister nice guy." The client is vulnerable to attack because it is downloading code that satisfies a request for a service, and it then executes that code. Without security checks, the client can download code from a hostile service. While the code has to implement the requested interface, and maybe satisfy conditions on associated `Entry` objects, without security it is otherwise unconstrained as to what it does.

For example, a client asking for a simple file classifier could end up getting this hostile object:

```
package hostile;
import common.MIMEType;
import common.FileClassifier;
/**
 * HostileFileClassifier1.java
 */
public class HostileFileClassifier1 implements FileClassifier,
              java.io.Serializable {
    public MIMEType getMIMEType(String fileName) {
        if (java.io.File.pathSeparator.equals("/")) {
            // Unix - don't uncomment the next line!
            // Runtime.getRuntime().exec("/bin/rm -rf /");
        } else {
            // DOS - don't uncomment the next line!
            // Runtime.getRuntime().exec("format c: /u");
        }
        return null;
    }
    public HostileFileClassifier1() {
        // empty
    }

} // HostileFileClassifier1
```

This object would be exported from a hostile service to run completely in any client unfortunate enough to download it. When the client executes the `getMimeType()` method, the method is run in the client to attempt to trash the client's system. (Mind you, if the attacker was stupid enough to implement the service using RMI, which exports a proxy stub, then the method would run in the *service's* JVM and attempt to trash the attacker's system instead!)

It is not necessary to actually call a method on the downloaded object—the mere act of downloading can do damage if the object overrides the deserialization method:

```
package hostile;
import common.MIMEType;
import common.FileClassifier;
/**
 * HostileFileClassifier2.java
 */
public class HostileFileClassifier2 implements FileClassifier,
    java.io.Externalizable {
    public MIMEType getMIMEType(String fileName) {
        return null;
    }
    public void readExternal(java.io.ObjectInput in) {
        if (java.io.File.pathSeparator.equals("/")) {
            // Unix - don't uncomment the next line!
            // Runtime.getRuntime().exec("/bin/rm -rf /");
        } else {
            // DOS - don't uncomment the next line!
            // Runtime.getRuntime().exec("format c: /u");
        }
    }
    public void writeExternal(java.io.ObjectOutput out)
        throws java.io.IOException{
        out.writeObject(this);
    }
    public HostileFileClassifier2() {
        // empty
    }

} // HostileFileClassifier2
```

The previous two classes assume that clients will make requests for the implementation of a particular interface, and this means that the attacker would require some knowledge of the clients they are attacking (that they will ask for this interface). At the moment, there are no standard interfaces, so this may not be a feasible way of attacking many clients. As interfaces such as those for a printer become specified and widely used, however, attacks based on hostile implementations of services may become more common. Even without well-known interfaces, clients such as service browsers that attempt to find all possible services can be attacked, simply because they look up subclasses of `Object`.

Removing AllPermission

Setting the security access to `AllPermission` is easy and removes all possible security issues that may hinder development of a Jini application. But it leaves your system open, so that you must start using a more rigorous security policy at some stage—hopefully before others have

damaged your system. The problem with moving away from this open policy is that permissions are *additive* rather than *subtractive*. That is, you can't take permissions away from `AllPermission`; you have to start with an empty permission set and add to that.

Not giving enough permission can result in at least three situations when you try to access something:

- A security-related exception can be thrown. This is comparatively easy to deal with, because the exception will tell you what permission is being denied. You can then decide whether or not you should be granting this permission. Of course, this should be caught during testing, not when the application is deployed!
- A security-related exception can be thrown but caught by some library object, which attempts to handle it. This happens within the multicast lookup methods, which make multicast requests. If this permission is denied, it will be retried several times before giving up. This leads to a cumulative time delay before anything else can happen. The application may be able to continue, and will just suffer this time delay.
- A security-related exception can be thrown but caught by some library object and ignored. The application may be unable to continue in any rational way after this, and it may just appear to hang. This may happen if network access is requested but denied, and then a thread waits for messages that can never arrive. Or it may just get stuck in a loop . . .

The first two cases will occur if permissions are turned off for the service providers, such as `rmi.FileClassifierServer`. The third occurs for the client `client.TestFileClassifier`.

You can set the `java.security.debug` system property to print information about various types of access to the security mechanisms. This can be used with a slack security policy to find out exactly what permissions are being granted. Then, with the screws tightened, you can see where permission is being denied. An appropriate value for this property is `access`, as in

```
java -Djava.security.debug=access ...
```

For example, running `client.TestFileClassifier` with few permissions granted may result in a trace such as the following:

```
...
access: access allowed (java.util.PropertyPermission socksProxyHost read)
access: access allowed (java.net.SocketPermission 127.0.0.1:1174 accept,resolve)
access: access denied (java.net.SocketPermission 130.102.176.249:1024 accept,
    resolve)
access: access denied (java.net.SocketPermission 130.102.176.249:1025 accept,
    resolve)
access: access denied (java.net.SocketPermission 130.102.176.249:1027 accept,
    resolve)
...
```

The denied access is an attempt to make a socket accept or resolve a request on my laptop (IP address 130.102.176.249), probably for RMI-related sockets. Since the client just sits there indefinitely making this request on one random port after another, this permission needs to be opened up, because the client otherwise appears to just hang.

Jini with Protection

The safest way for a Jini client or service to be part of a Jini federation is through abstinence—that is, for it to refuse to take part. This doesn't get you very far in populating a federation, though. The JDK 1.2 security model allows a number of ways in which more permissive activity may take place:

- Grant permission only for certain activities, such as socket access at various levels on particular ports, or access to certain files for reading, writing, or execution.

```
grant {
    permission java.net.SocketPermission "224.0.1.85", "connect,accept";
    permission java.net.SocketPermission "*.edu.au:80", "connect";
}
```

- Grant access only to particular hosts, subdomains, or domains.

```
grant codebase "http://sunshade.dstc.edu.au/classes/" {
    permission java.security.AllPermission "", "";
}
```

- Require digital signatures to be attached to code.

```
grant signedBy "sysadmin" {
    permission java.security.AllPermission "", "";
}
```

For any particular security access, you will need to decide which of these options is appropriate. Your choice will depend on the overall security policy for your organization, and if your organization doesn't have such a policy that you can refer to, then you certainly shouldn't be exposing your systems to the Internet (or to anyone within the organization, either)!

Service Requirements

In order to partake in a Jini federation, a service must become sufficiently visible. The service needs to find a service locator before it can advertise its services. Once the service locator is found, it registers the service and then waits for calls to come in. Where is the security risk in this? Well, first, as a result of finding a service locator, the service gets a `ServiceRegistrar`. This runs in the server's JVM. If the lookup discovery service is hostile, then it can attack the server. Second, if the service has event listeners, then clients will be loading listener objects into the JVM, and these could attack the server, too.

Lookup locator discovery can be done by unicast to particular locations or by multicast. Sufficient permissions to do this must be granted. Unicast discovery does not need any particular permissions to be set. The discovery can be done without any policy file needed. For the multicast case, the service must have `DiscoveryPermission` for each group that it is trying to join. For all groups, the asterisk (*) wildcard can be used. So, to join all groups, the permission granted should be as follows:

```
permission net.jini.discovery.DiscoveryPermission "*";
```

For example, to join the `printers` and `toasters` groups, the permission would be this:

```
permission net.jini.discovery.DiscoveryPermission,
           "printers, toasters";
```

Once this permission is given, the service will make a multicast broadcast on 224.0.1.84. This particular address is used by Jini for broadcasts and should be used in your policy files. Socket permission for these requests and announcements must be given as follows:

```
permission java.net.SocketPermission "224.0.1.84", "connect,accept";
permission java.net.SocketPermission "224.0.1.85", "connect,accept";
```

The service may export a `UnicastRemoteObject`, which will need to communicate back to the server, and so the server will need to listen on a port for these remote object requests. The default constructor will assign a random port (above 1024) for this. If desired, this port may be specified by other constructors, which will require further socket permissions, such as the following:

```
permission java.net.SocketPermission "localhost:1024-", "connect,accept";
permission java.net.SocketPermission "*.dstc.edu.au:1024-", "connect,accept";
```

to accept connections on any port above 1024 from the `localhost` or any computer in the `dstc.edu.au` domain.

A number of parameters may be set by preferences, such as `net.jini.discovery.ttl`. It does no harm to allow the Jini system to look for these parameters, and this may be allowed by including code such as the following in the policy file:

```
permission java.util.PropertyPermission "net.jini.discovery.*", "read";
```

A fairly minimal policy file suitable for a service exporting an RMI object could then be as follows:

```
grant {
    permission net.jini.discovery.DiscoveryPermission "*";
    // multicast request address
    permission java.net.SocketPermission "224.0.1.85", "connect,accept";
    // multicast announcement address
    permission java.net.SocketPermission "224.0.1.84", "connect,accept";
    // RMI connections
    permission java.net.SocketPermission "*.canberra.edu.au:1024-",
        "connect,accept";
    permission java.net.SocketPermission "130.102.176.249:1024-",
        "connect,accept";
    permission java.net.SocketPermission "127.0.0.1:1024-",
        "connect,accept";
    // reading parameters
    // like net.jini.discovery.debug!
    permission java.util.PropertyPermission "net.jini.discovery.*", "read";
};
```

Client Requirements

The client is most at risk in the Jini environment. The service exports objects (and imports only relatively trusted service registrars); the lookup locator stores objects, but does not "bring them to life" or execute any of their methods; but the client brings an external object into its address space and runs it, giving it all of the permissions of a process running in an operating system. The object will run under the permissions of a particular user in a particular directory, with user accesses to the local file system and network. It could destroy files, make network connections to undesirable sites (or desirable sites, depending on your tastes!) and download images from them, start processes to send obnoxious mail to anyone in your address book, and generally make a mess of your electronic identity.

A client using multicast search to find service locators will need to grant discovery permission and multicast announcement permission, just like the service:

```
permission net.jini.discovery.DiscoveryPermission "*";
permission java.net.SocketPermission "224.0.1.84", "connect,accept";
permission java.net.SocketPermission "224.0.1.85", "connect,accept";
```

RMI connections on random ports may also be needed:

```
permission java.net.SocketPermission "*.dstc.edu.au:1024-", "connect,accept"
```

In addition, class definitions will probably need to be uploaded so that services can actually run in the client. This is the most serious risk area for the client, as the code contained in these class definitions will be run in the client, and any errors or malicious code will have their effect because of this. The client view of the different levels of trust is shown in Figure 14-1.

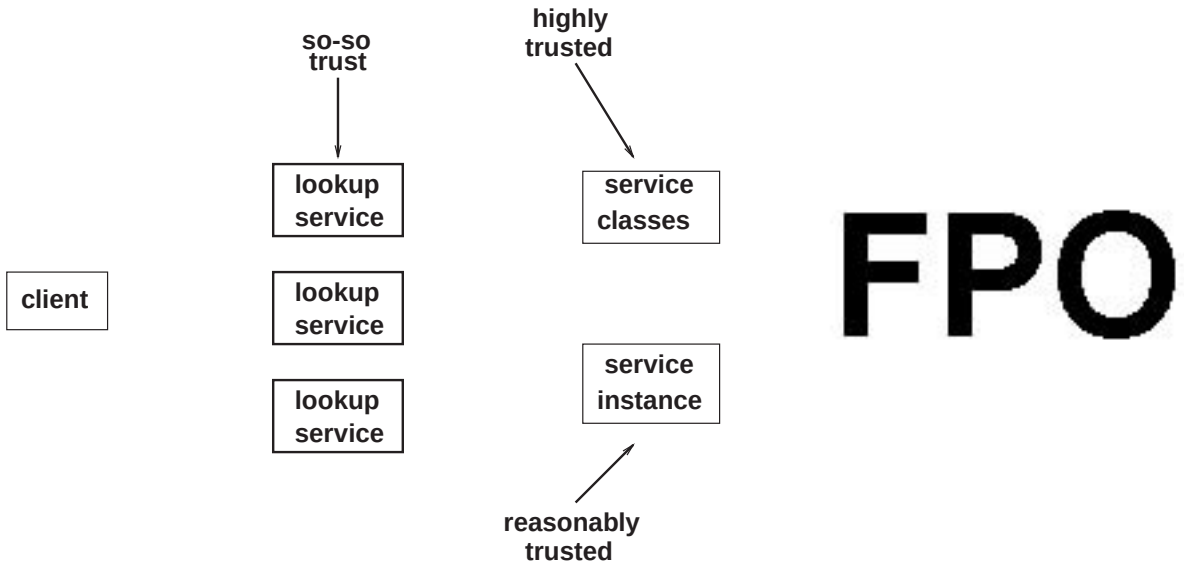

Figure 14-1. *Trust levels of the client*

The client is the most likely candidate to require signed trust certificates since it has the highest trust requirement of the components of the system.

Many services will just make use of whatever HTTP server is running on their system, and this will probably be on port 80. Permission to connect on this port can be granted with the following statements:

```
permission java.net.SocketPermission "127.0.0.1:80", "connect,accept";
permission java.net.SocketPermission "*.dstc.edu.au:80", "connect,accept";
```

However, while this will allow code to be downloaded on port 80, it may not block some malicious attempts. Any user can start an HTTP server on any port (Windows) or above 1024 (Unix). A service can then set its codebase to whatever port the HTTP server is using. Perhaps these other ports should be blocked, but unfortunately, RMI uses random ports, so these ports need to be open.

So, it is not probably possible to close all holes for hostile code to be downloaded to a client. What you need is a second stage defense: given that hostile code may reach you, set the JDK security so that hostile (or just buggy) code cannot perform harmful actions in the client.

A fairly minimal policy file suitable for a client could then be as follows:

```
grant {
    permission net.jini.discovery.DiscoveryPermission "*";
    // multicast request address
    permission java.net.SocketPermission "224.0.1.85", "connect,accept";
    // multicast announcement address
    permission java.net.SocketPermission "224.0.1.84", "connect,accept";
    // RMI connections
    // DANGER
    // HTTP connections - this is where external code may come in - careful!!!
    permission java.net.SocketPermission "127.0.0.1:1024-", "connect,accept";
    permission java.net.SocketPermission "*.canberra.edu.au:1024-",
        "connect,accept";
    permission java.net.SocketPermission "130.102.176.249:1024-", "connect,accept";
    // DANGER
    // HTTP connections - this is where external code may come in - careful!!!
    permission java.net.SocketPermission "127.0.0.1:80", "connect,accept";
    permission java.net.SocketPermission "*.dstc.edu.au:80", "connect,accept";
    // reading parameters
    // like net.jini.discovery.debug!
    permission java.util.PropertyPermission "net.jini.discovery.*", "read";
};
```

RMI Parameters

A service is specified by an interface. In many cases, an RMI proxy will be delivered to the client that implements this interface. Depending on the interface, the proxy can sometimes be used by the client to attack the service. The `FileClassifier` interface is safe, but in Chapter 15 we look at how a client can upload a new MIME type to a service, and this extended interface exposes a service to attack.

The relevant method from the `MutableFileClassifier` interface is as follows:

```
public void addType(String suffix, MIMEType type)
    throws java.rmi.RemoteException;
```

This method allows a client to pass an object of type `MIMEType` up to the service, where it will presumably try to add it to a list of existing MIME types. The `MIMEType` class is an ordinary class, not an interface. Nevertheless, it can be subclassed, and this subclass can perform the tricks discussed earlier to make an attack.

This particular attack can be avoided by ensuring that the parameters to any method call in an interface are all `final` classes. If the class `MIMEType` was defined by

```
public final class MIMEType {...}
```

then it would not be possible to subclass it. No attack could be made by a subclass, since no subclass could be made! There aren't enough Jini services defined yet to know whether making all parameters `final` is a good enough solution.

ServiceRegistrar

Services will transfer objects to be run within clients. This chapter has so far been concerned with the security policies that will allow this and the restrictions that may need to be in place. The major protection for clients at the moment is that there are no standardized service interfaces, so attackers do not yet know what hostile objects to write.

A lookup service, on the other hand, exports an object that implements `ServiceRegistrar`. It does not use the same mechanism as a service would to get its code into a client. Instead, the lookup service replies directly to unicast connections with a registrar object, or responds to multicast requests by establishing a unicast connection to the requester and again sending a registrar. The mechanism is different, but it is clearly documented in the Jini specifications, and it is quite easy to write an application that performs at least this much of the discovery protocols.

The end result of lookup discovery is that the lookup service will have downloaded registrar objects. The registrar objects run in both clients and services; they both need to find lookup services. The `ServiceRegistrar` interface is standardized by the Jini specification, so it is fairly easy to write a hostile lookup service that can attack both clients and services.

While it is unlikely that anyone will knowingly make a unicast connection to a hostile lookup service, someone might get tricked into doing so. There are already some quite unscrupulous web sites that will offer "free" services on production of a credit card (to the user's later

cost). There is every probability that such sites will try to entice Jini clients if they see a profit in doing so. Also, anyone with access to the network within broadcast range of clients and services (i.e., on your local network) can start lookup services that will be found by multicast discovery.

The only real counter to this attack is to require that all connections that can result in downloaded code should be covered by digital certificates, so that all downloaded code must be signed. This covers all possible ports, since an HTTP server can be started on any port on a Windows machine. The objects that are downloaded in the Sun implementation of the lookup service, `reggie`, are all in `reggie-dl.jar`. This is not signed by any certificates. If you are worried about an attack through this route, you should sign this file as well as the `.jar` files of any services you wish to use.

Being Paranoiac

Jini applications download and execute code from other sources, including the following:

- Both clients and services download `ServiceRegistrar` objects from lookup services. They then call methods such as `lookup()` and `register()`.
- A client will download services and execute whatever methods are defined in the interface.
- A remote listener will call the `notify()` method of foreign code.

In a safe environment where all code can be trusted, no safeguards need to be employed. However, most environments carry some kind of risk from hostile agents. An attack will consist of a hostile agent implementing one of the known interfaces (of `ServiceRegistrar`, of a well-known service such as the transaction manager, or of `RemoteEventListener`) with code that does not implement the implied "contract" of the interface but instead tries to perform malicious acts. These acts may not even be deliberately hostile: most programmers make at least some errors, and these errors may result in risky behavior.

There are all sorts of malicious acts that can be performed. Hostile code can simply terminate the application, but the code can also perform actions such as read sensitive files, alter sensitive files, forge messages to other applications, perform denial of service attacks such as filling the screen with useless windows, and so on.

It doesn't take much reading about security issues to instill a strong sense of paranoia and possible overreaction to security threats. If you can trust everyone on your local network (which you are already doing if you run a number of common network services such as NFS), then the techniques discussed in this section are possibly overkill. If you can't, then paranoia may be a good frame of mind to be in!

Protection Domains

The Java 1.2 security model is based on the traditional idea of *protection domains*. In Java, a protection domain is associated with classes based on their `CodeSource`, which consists of the URL from which the class file was loaded, plus a set of digital certificates used to sign the class files. For example, the class files for the `LookupLocator` class are in the file `jini-core.jar` (in the

lib directory of the Jini distribution). This class has a protection domain associated with the CodeSource for jin-core.jar. (All of the classes in jini-core.jar will belong to this same protection domain.)

Information about protection domains and code sources can be found by code such as this:

```
java.security.ProtectionDomain domain = registrar.
                                        getClass().getProtectionDomain();
java.security.CodeSource codeSource = domain.getCodeSource();
```

Information about the digital signatures attached to code can be found by code such as this:

```
Object [] signers = registrar.getClass().getSigners();
if (signers == null) {
    System.out.println("No signers");
} else {
    System.out.println("Signers");
    for (int m = 0; m < signers.length; m++)
        System.out.println(signers[m].toString());
}
```

By default, no class files or .jar files have digital signatures attached. Digital signatures can be created using keytool (part of the standard Java distribution). These signatures are stored in a *keystore*. From there, they can be used to sign classes and .jar files using jarsigner, exported to other keystores, and generally spread around. Certificates don't mean anything unless you believe that they really do guarantee that they refer to the "real" person, and certificate authorities, such as VeriSign, provide validation techniques for this.

This description has been horribly brief and is mainly intended as a reminder for those who already understand this stuff. If you want to experiment, you can do as I did and just create certificates as needed, using keytool, although there was no independent authority to verify them. A good explanation of this topic is given by Bill Venners at http://www.artima.com/insidejvm/ed2/security.html.

Signing Standard Files

None of the Java files in the standard distribution is signed. None of the files in the Jini distribution is signed either. For most of these it probably won't matter, since they are local files.

However, all of the Jini jar files ending in -dl.jar are downloaded to clients and services across the network and are Sun implementations of "well-known" interfaces. For example, the ServiceRegistrar object that you get from the discovery process has its class files defined in reggie-dl.jar, as a com.sun.jini.reggie.RegistrarImpl_Stub object. Hostile code implementing the ServiceRegistrar interface can be written quite easily. If there is the possibility that hostile versions of lookup services (or other Sun-supplied services) may be set running on your network, then you should only accept implementations of ServiceRegistrar if they are signed by an authority you trust.

Signing Other Services

Interfaces to services such as printers will eventually be decided and will become "well known." There should be no need to sign these interface files for security reasons, but an authority may wish to sign them for, say, copyright reasons. Any implementations of these interfaces are a different matter. Just like the previous cases, these implementation class files will come to client machines from other machines on the local or even remote networks. These are the files that can have malicious implementations. If this is a possibility, you should only accept implementations of the interfaces if they are signed by an authority you trust.

Permissions

Permissions are granted to protection domains based on their codesource. In the Sun implementation, this is done in the policy files, by `grant` blocks:

```
grant codeBase "url" signedBy "signer" {
    ...
}
```

When code executes, it belongs to the protection domains of all classes on the call stack above it. So, for example, when the `ServiceRegistration` object in the `complete.FileClassifierServer` is executing the `register()` method, the following classes are on the call stack:

- The `com.sun.jini.reggie.RegistrarImpl_Stub` class from `reggie-dl.jar`
- The `complete.FileClassifierServer` class, from the call `discovered()`
- Core Jini classes that have called the `discovered()` method
- Classes from the Java system core that are running the application

The permissions for executing code are generally the intersection of all the permissions of the protection domains it is running in. Classes in the Java system core grant all permissions, but if you restrict the permissions granted to your own application code to core Jini classes, or to code that comes across the network, you restrict what an executing method can do. For example, if multicast request permission is not granted to the Jini core classes, then discovery cannot take place. This permission needs to be granted to the application code and also to the Jini core classes.

It may not be immediately apparent what protection domains are active at any point. For example, in the earlier call of

```
registrar.getClass().getProtectionDomain()
```

I fell into the assumption that the `reggie-dl.jar` domain was active because the method was called on the `registrar` object. But it wasn't. While the call `getClass()` is made on the `registrar`, this completes and returns a `Class` object so that the call is made on this object, which by then is just running in the three domains: the system, the application, and the core Jini classes domains.

There are two exceptions to the intersection rule. The first is that the RMI security manager grants `SocketPermission` to connect back to the codebase host for remote classes. The second is

that methods may call the `AccessController.doPrivileged()` method. This essentially prunes the class call stack, discarding all classes below this one for the duration of the call, and it is done to allow permissions based on this class's methods, even though the permissions may not be granted by classes earlier in the call chain. This allows some methods to continue to work even though the application has not granted the permission, and it means that the application does not have to generally grant permissions required only by a small subset of code.

For example, the `Socket` class needs access to file permissions in order to allow methods such as `getOutputStream()` to function. By using `doPrivileged()`, the class can limit the "security breakout" to particular methods in a controlled manner. If you are running with security access debugging turned on, this explains how a large number of accesses are granted, even though the application has not given many of the permissions.

Putting It All Together

Adding all these bits of information together leads to security policy files that restrict possible attacks:

1. Grant permissions to application code based on the codesource. If you suspect these classes might get tampered with, sign them as well.
2. Grant permission to Jini core classes based on the codesource. These may be signed if need be.
3. Grant permission to downloaded code only if it is signed by an authority you trust. Even then, grant only the minimum permission that is needed to perform the service's task.
4. Don't grant any other permissions to other code.

A file based on this security policy might look like this:

```
keystore "file:/home/jan/.keystore", "JKS";
// Permissions for downloaded classes
grant signedBy "Jan" {
    permission java.net.SocketPermission "137.92.11.117:1024-",
        "connect,accept,resolve";
};
// Permissions for the Jini classes
grant codeBase "file:/home/jan/tmpdir/jini1_1/lib/-" signedBy "Jini" {
    // The Jini classes shouldn't require more than these
    permission java.util.PropertyPermission "net.jini.discovery.*", "read";
    permission net.jini.discovery.DiscoveryPermission "*";
    // multicast request address
    permission java.net.SocketPermission "224.0.1.85", "connect,accept";
    // multicast announcement address
    permission java.net.SocketPermission "224.0.1.84", "connect,accept";
    // RMI and HTTP
    permission java.net.SocketPermission "127.0.0.1:1024-", "connect,accept";
    permission java.net.SocketPermission "*.canberra.edu.au:1024-",
        "connect,accept";
```

```
    permission java.net.SocketPermission "137.92.11.*:1024-",
        "connect,accept,resolve";
    permission java.net.SocketPermission "130.102.176.*:1024-",
        "connect,accept,resolve";
    permission java.net.SocketPermission "130.102.176.249:1024-",
        "connect,accept,resolve";
    // permission java.net.SocketPermission "137.92.11.117:1024-",
        "connect,accept,resolve";
    // debugging
    permission java.lang.RuntimePermission "getProtectionDomain";
};
// Permissions for the application classes
grant codeBase "file:/home/jan/projects/jini/doc/-" {
    permission java.util.PropertyPermission "net.jini.discovery.*", "read";
    permission net.jini.discovery.DiscoveryPermission "*";
    // multicast request address
    permission java.net.SocketPermission "224.0.1.85", "connect,accept";
    // multicast announcement address
    permission java.net.SocketPermission "224.0.1.84", "connect,accept";
    // RMI and HTTP
    permission java.net.SocketPermission "127.0.0.1:1024-", "connect,accept";
    permission java.net.SocketPermission "*.canberra.edu.au:1024-",
        "connect,accept";
    permission java.net.SocketPermission "137.92.11.*:1024-",
        "connect,accept,resolve";
    permission java.net.SocketPermission "130.102.176.*:1024-",
        "connect,accept,resolve";
    permission java.net.SocketPermission "130.102.176.249:1024-",
        "connect,accept,resolve";
    // permission java.net.SocketPermission "137.92.11.117:1024-",
        "connect,accept,resolve";
    // debugging
    permission java.lang.RuntimePermission "getProtectionDomain";
    // Add in any file, etc., permissions needed by the application classes
};
```

Summary

You have to pay attention to security when running in a distributed environment, and Jini enforces security by using the JDK 1.2 security model. This chapter considered the range of security mechanisms possible, from turning security off through to paranoiac mode. It should be noted that this chapter did not cover issues such as encryption or nonrepudiation; these features are discussed in Chapter 22.