The java.policy File in IBM Domino

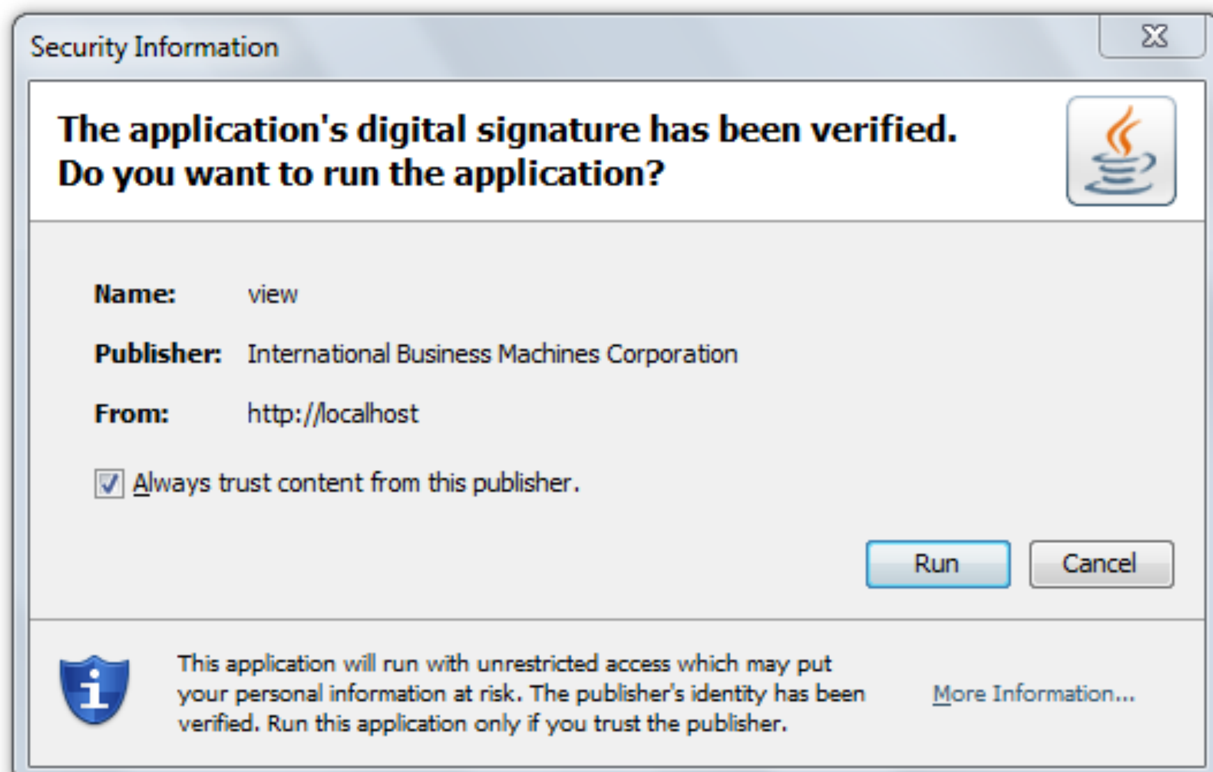
by Julian Robichaux, panagenda
originally published on socialbizug.org, September 2014

If you've been writing or supporting Java code in IBM Notes and Domino for long enough, you will eventually encounter the dreaded java.lang.SecurityException telling you that your code is not allowed to perform a certain action. While the solution to the problem often involves the java.policy file somehow (or, just not calling the offending code in the first place), it's not very obvious how that file works or exactly what you need to change. This article will attempt to clear up the murky waters.

A Very Brief History of Security in Java

In the very early days of Java, there was the security sandbox. The sandbox prevented remote Java code from doing malicious things to your computer, so that it would be "safe" to run Java from a browser — remember that a lot of the initial programming examples and use-cases had to do with applets. Local Java code was outside the sandbox and it had full access to your computer, but remote programs and applets were restricted from doing anything outside the sandbox. The security model was sort of binary: remote code in sandbox, local code outside of sandbox, done.

Because some of these non-sandbox things (like access to files and network sockets) was desirable, the next iteration of Java security had the notion of "signed" code. Remote code and applets that had a valid signature could be permitted to run outside the sandbox, if you allowed it to. That was helpful, but it was still an all-or-nothing proposition: either you were inside the sandbox or you were outside the sandbox. There was no in-between.



Soon after that, Java security was implemented in the way we know it now. There is a SecurityManager, and all code (local or remote) has to check with the SecurityManager before it is able to perform non-sandboxed types of actions. Permission to perform restricted actions is somewhat fine-grained, so you can allow specific types of access to code with specific signatures or from specific locations. This permission is established in the java.policy file.

[NOTE 1: you can launch a Java JVM instance with no SecurityManager at all, and completely bypass all the checks. However, in the context of IBM Domino there is always a SecurityManager and everything in this article will apply.]

[NOTE 2: it's obviously much more complicated than "there's a SecurityManager", because security is inherently complex. But for the purposes of our discussion that's the level we'll be working at.]

The java.security File

Our first stop in this journey is the java.security file, located in the jvm/lib/security folder. This is where the SecurityManager looks to find its basic configuration information. My best advice here is: do not edit this file!

There is some information here about cryptography providers, SSL settings, and the like (because the notion of security goes beyond just sandboxing code). But the important section for what we're talking about is:

```
policy.url.1=file:${java.home}/lib/security/java.policy
policy.url.2=file:${java.home}/lib/security/java.pol
policy.url.3=file:///${user.home}/.java.policy
```

This lists the location of the policy files that Java uses to build its list of permissions. The SecurityManager parses these files once, at JVM startup.

Note that I said "files" there, in the plural. There can be multiple policy files in effect, that are combined together by the SecurityManager. We will discuss the importance of this in a moment. For now, just note that the java.policy file is the default file, and it is pre-populated with permissions, and it is what we will be looking at next.

The java.policy File

Most of what we'll be discussing for the rest of this article is the java.policy file (and its siblings). In an IBM Domino installation, this file can be found in the jvm/lib/security directory. A few initial tidbits about this file:

- It must use UTF-8 encoding (although ASCII is fine if you're not using accented characters)
- Typos can potentially cause some or all the permissions in the file to be ignored
- Path separators for files and directories are "/" for all platforms, and optionally "\\" for Windows (but don't use a single "\")

First thing to understand about java.policy: it only GIVES permission to do things, it does not take permissions away.

With the Java SecurityManager running, there is a [http://docs.oracle.com/javase/7/docs/technotes/guides/security/spec/security-spec.doc3.html] default list of actions that all Java code does NOT have permission to do. So if you don't have a java.policy file at all, you will be completely sandboxed and you won't have any permissions outside the sandbox, unless you're running a custom security manager that gives you special default permissions.

The second thing to understand is that java.policy permissions can be granted based on one or more of the following:

- The certificate used to sign the code
- The name of the code signer
- Where the code is located (either a local or remote location, referred to as the codeBase)

I won't go into detail about the first two options here because it involves setting up a keystore, but that might be a nice option if you have a highly organized set of admins and developers. Two points of note if you decide to explore this option: you can specify only one keystore in the current default Java implementation, and if your keystore requires a password to open (which it should!) the option for specifying a password in the java.policy file might leave you wanting. For more information, see http://docs.oracle.com/javase/7/docs/technotes/guides/security/PolicyFiles.html#FileSyntax

So that leaves us with using either the codeBase to specify permissions, or granting permissions globally to all code. By default, the java.policy file in IBM Domino gives some minimal permissions to all Java code in a grant block like this:

```
grant {
   permission java.lang.RuntimePermission "stopThread";
   permission java.net.SocketPermission "localhost:1024-", "listen";
   permission java.util.PropertyPermission "java.version", “read";
   // several more PropertyPermissions below…
};
```

This gives threads the ability to stop themselves, the ability to listen on local network ports greater than 1024, and the ability to read many different Java properties.

The codeBase permissions look like this:

```
grant codeBase "file:${java.home}/lib/ext/*" {
   permission java.security.AllPermission;
};
```

In this case, it gives any class or JAR file in the jvm/lib/ext folder the ability to do anything (I bet you always wondered why the ext folder was so special). There are several other folders that are granted the AllPermission abilities too, including files in the Notes program, xsp, and osgi folders.

One interesting bit of syntax here: if you end a codeBase with "*", that means "all the files in this folder but NOT in subfolders". If you end it with "-", that will include all files in all subfolders.

The java.pol And .java.policy Files
Remember the policy.url entries in the java.security file? There were three of them. The first one referenced the java.policy file we talked about above, but the other two were:

${java.home}/lib/security/java.pol
${user.home}/.java.policy

On an IBM Domino server, ${java.home} is the jvm/ folder in your Domino program directory (so the java.pol file is in the same place as the java.policy file) and ${user.home} is your home directory on the local machine (for example: c:\users\julian for me).

If either or both of these files exist, they will be COMBINED with your java.policy file. Remember that the policy files can only grant permissions, not take them away. So the formula for determining how the permissions are combined is easy: you get all the permissions from all the entries in all the files that pertain to your code.

Should you use these files? Yes!

Which one should you use? It depends!

First, the "why should you use them" reason. When you upgrade to new versions of Domino, be it a major version, a minor version, or a fixpack, the java.policy file often (always?) gets overwritten. So if you edit the java.policy file, you need to go and re-edit it every time you update your server, which you will almost certainly forget to do. The java.pol file usually gets left alone.

In terms of which one to use, that's sort of a matter of personal preference. I would tend to lean towards the java.pol file because it's in the same place as the java.policy file, making it easier to stumble across if you've forgotten it's there. The ${user.home} folder might not be in an obvious or easily accessible place for whatever context your server is running under also.

HOWEVER, be warned that the java.pol file can potentially be overwritten. It doesn't usually happen, but it did happen recently to Mark Leusink when he installed 9.0.1 FP1 [http://linqed.eu/?p=482]. So it's possible that the ${user.home} folder is safer. Your call.

A Quick Example
Here's a quick example of where you might need to use this knowledge. Let's say you have an XPage with SSJS that looks like this:

```
try {
   var loggerName = "com.example.logger";
   var logger = java.util.logging.Logger.getLogger(loggerName);
   logger.setLevel(java.util.logging.Level.FINEST);
   logger.fine("This action ran just fine");
} catch(e) {
```

```
    _dump(e);
}
```

With default Java policies and permissions in place, you will see this message on the Domino server console when the code runs:

java.security.AccessControlException: Access denied (java.util.logging.LoggingPermission control)

Okay, it looks like we did something with logging that we weren't allowed to do. Luckily, the error message told us exactly which permission was missing: java.util.logging.LoggingPermission control .

If it's not clear by the error message which permission you need, you will need to look at a stack trace of the error and note the method call that is causing the Security Exception. Then you can cross-reference the method against this list and find the appropriate permission: http://docs.oracle.com/javase/7/docs/technotes/guides/security/permissions.html#PermsAndMethods

Now we can create a jvm/lib/security/java.pol file using a text editor, with the following content:

```
grant {
    permission java.security.SecurityPermission "createAccessControlContext";
    permission java.util.logging.LoggingPermission "control";
};
```

The "createAccessControlContext" bit is a new requirement (feature?) in Domino 8.5.3FP5 and 9.0.1. See this IBM Technote for details: http://www-01.ibm.com/support/docview.wss?uid=swg21669594 .

After you've created and saved the java.pol file, you must restart the Notes server for the changes to take effect. When you run the code again, you should see that the error is gone — if not, please check for typos.

Special XPages codeBase Option

In the XPages Portable Command Guide from IBM Press, the authors mention an interesting XPages-specific codeBase you can use to give permission to ONLY to the XPages in a specific database. Here's an example, modifying the permission we granted above:

```
grant codeBase "xspnsf://server:0/julian/teststuff.nsf/-" {
    permission java.security.SecurityPermission "createAccessControlContext";
    permission java.util.logging.LoggingPermission "control";
};
```

This will only grant the LoggingPermission to XPages in the julian/TestStuff.nsf database on the server, instead of granting it to all code in all databases.

A few notes on this:

- The server portion of the URL must be “server:0”. Not the name of your server, not port 80 or 443 or whatever. It should be exactly “server:0”.
- Everything must be lowercase.
- The URL must end with a “/-“. That will include the database and everything inside it.
- This is XPages only, not Java agents.

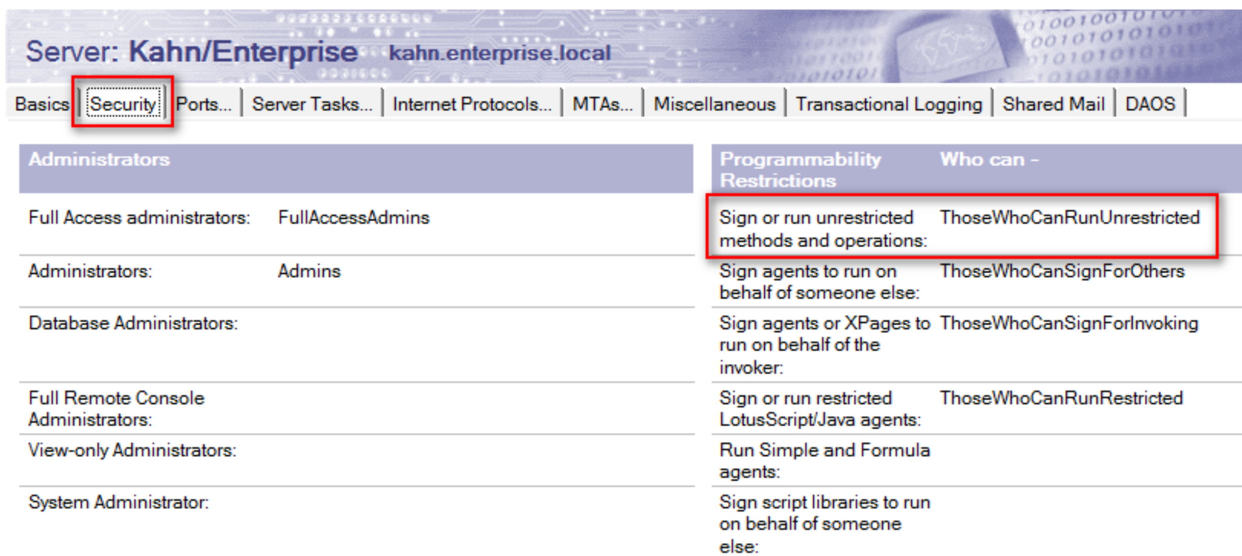
That’s a nice option anyway, if you want to grant dangerous permissions on a database-by-database basis.

But Wait, IBM Domino is Special…
There’s one more thing though. Let’s say you run some Java code from a Domino database that accesses the file system like this:

```
File tempFile = File.createTempFile("test", ".tmp");
System.out.println("temp file created: " + tempFile);
tempFile.delete();
```

If you have high enough access on the Domino server, this will work EVEN THOUGH YOU DON’T HAVE FILE PERMISSIONS IN THE JAVA POLICY FILES. What the heck?

What’s happening is that the Domino server has a custom SecurityManager implementation that also respects the “Restricted Methods” section in the Domino server document.



As far as Java policies are concerned, the relevant “restricted” methods are ones that access the file system and ones that access network sockets. If you are listed as having “Sign or run unrestricted methods and operations” in the server doc, you can access files and sockets even though the Java policy might not allow you to. Likewise, even if the Java policy gives you all the file and network access in the world, the Domino server can prevent you from accessing those things. Domino security trumps Java policy in that case.

So if you're troubleshooting Java code that's not able to access files or sockets, look to the Domino server doc rather than the java.policy file.

References and Links
For further reading, see the following links:

Java Security Architecture: http://docs.oracle.com/javase/7/docs/technotes/guides/security/spec/security-specTOC.fm.html
Policy File Syntax: http://docs.oracle.com/javase/7/docs/technotes/guides/security/PolicyFiles.html#FileSyntax
Permissions in the JDK: http://docs.oracle.com/javase/7/docs/technotes/guides/security/permissions.html#PermsAndMethods
IBM Technote on using a java.pol file: https://www-304.ibm.com/support/docview.wss?uid=swg21679242
IBM Technote about java.policy changes in 8.5.3 FP5: http://www-01.ibm.com/support/docview.wss?uid=swg21669594

Also, a special thank you to Declan Lynch who reminded me about the use of java.pol files when I was talking to someone about Java policies at the MWLUG conference a few weeks ago, which ultimately prompted me to write this up.