Going Beyond Standard Security for the Java™ Platform

Otto Moerbeek

Chief Architect, Tryllian

otto@tryllian.com

Tryllian

Outline

- Tryllian’s platform
- Standard Java security model
- Tryllian’s requirements
- Tryllian’s custom policy

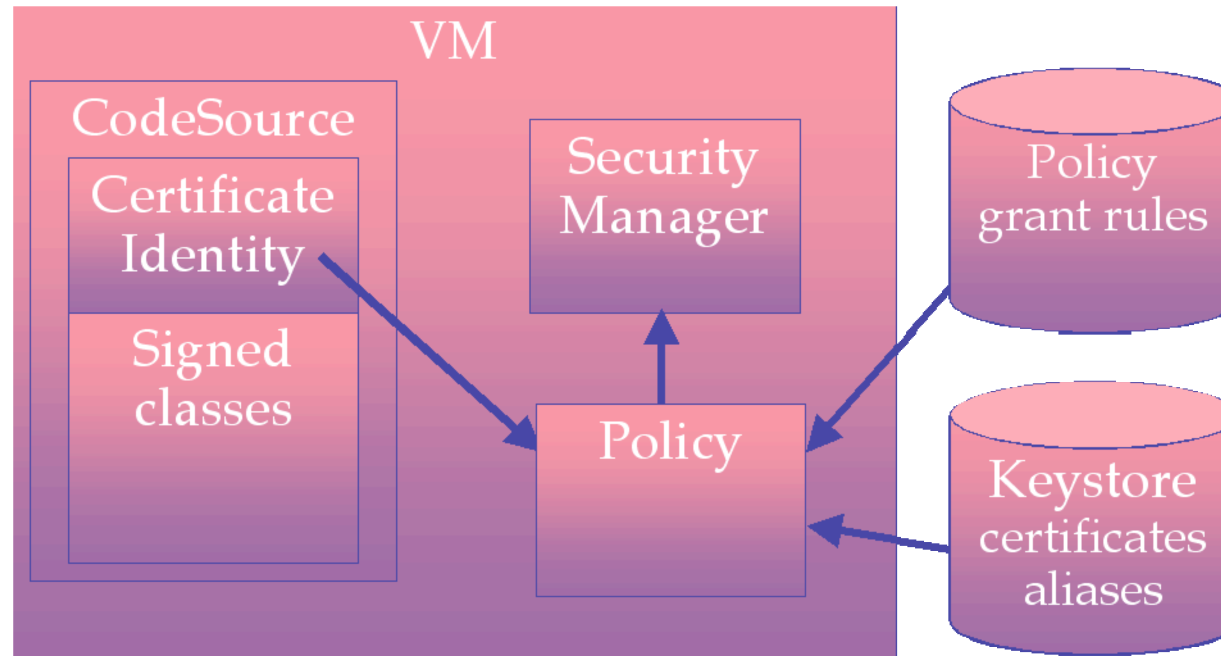
Tryllian

- Agent platform and development kit
- Runs on J2SE 1.3 VM
- Platform for distributed computing
- Targeted at open environments and enterprises

Standard Security Policy Implementation

- Classes are retrieved from server
- Network is not trusted
- Code is signed
- Policy defines the permissions
- Security Manager and class library enforce policy

Security Manager enforces Policy



Security manager grants permissions based on verified identity, and according to the policy.

Tryllian

Joe developer makes an applet

- Joe gets certificate from CA
- Joe packs code in jar file
- Joe signs jar file using his private key

Platform checks Joe's code

- Verify signature of code
  - Get Joe's certificate from jar file
  - Use Joe's public key in certificate to verify signature of classes
- Check Certificate
  - Check trust chain
  - Certificate of CA of Joe's certificate should be known
  - Lookup alias of Joe in key store
- Result: verified authenticity of Joe's code

Standard permission policy

- Based on location and certificates of code
- Defined by permission policy file:

```
grant codeBase URL signedBy alias {
   permission java.util.PropertyPermission "java.version", "read";
   permission java.util.PropertyPermission "java.vendor", "read";
   permission java.util.PropertyPermission "java.vendor.url", "read";
   permission java.util.PropertyPermission "java.class.version", "read";
   permission java.util.PropertyPermission "os.name", "read";
   … a lot of entries
};
```

Standard policy

- Policy files have lot of entries
- Policy entries needed for each code signer
- No way of grouping permissions
- Only possible to grant permissions to known parties

Imagine you are the Security Officer managing a big distributed environment…

Now take a look at mobile agents

- Agents are mobile components
- Agents are mobile code that carry state
- Agents travel from VM to VM
- Agents can originate from many places
- Tryllian agents run in corporate and open distributed environment

Extra security requirements…

Tryllian's Security Requirements

- Run agents from (unknown) third parties
- Platform should be protected from agents
- Agents should be protected form each other
- Resource protection: memory, CPU, files, network, …

Life of Security Officer should be made more easy.

Agents show standard Security Model shortcomings

- No way of grouping permissions into named roles
- Access can only be granted to known parties
- A class can be denied access to resources, only *after* a class has been loaded
- Some potential harmful functions may be called, e.g. thread creation

Tryllian's Custom Policy

- Roles

  Enable grouping of permissions

- Delegation

  Enable granting of roles to groups

Makes life of Security Officer more easy.

What is a role?

- Role defines *named* set of permissions
- Role mapping defines mapping from key store alias to role
- Result: easy way to define common set of permissions
- Model can easily be extended to include more expressive role definitions.

Role interfaces

```
public interface Role
{
    String getName();
    PermissionCollection
                  getPermissions();
}
public interface RoleMapping
{
    Role getRole(String alias);
}
```

Define custom policy

In `java.security.Policy` class:

```
PermissionCollection
  getPermissions(CodeSource codesource);
```

Standard policy:

class → certificate → alias → permissions

Custom policy using roles:

class → certificate → alias → role → permissions

Next: use delegation for granting permissions

- Use certificate chain for role assignment
- I may not know signer *A*, but I know and trust the certification authority *B* that issued *A's* certificate
- Assign permissions based on role of certification authority *B*

Enables assignment of permissions to unknown, but trusted parties.

DelegatingPolicy

```
// Assign permissions to list of certificates
public PermissionCollection getPermissions(
      Certificate[] certs) {
  Permissions perms = new Permissions();

  Split up certificate in chains
  For each chain {
       Verify chain
       String alias = First known alias of chain
       Role r = rolemapping.getRole(alias);
       Extend perms with r.getPermissions();
  }

  return perms;
}
```

Define custom class loader

- Class loader's **`getPermission()`** determines permissions associated with code
- The custom class loader **`DelegatingCL`** uses **`DelegatingPolicy`** to determine permissions

Result: system classes use standard policy, our classes use custom policy.

DelegatingCL uses this policy

```
public class DelegatingCL extends URLClassLoader {
  public DelegatingCL(URL[] urls, ClassLoader parent,
                              Policy policy) {
     super(urls, parent);
     this.policy = policy;
}
  /**
   * Return permissions based on the policy of this CL
   */
  protected PermissionCollection
       getPermissions(CodeSource cs)
  {
     return policy.getPermission(cs);
  }
}
```

Summary
Tryllian's custom policy

- Build on top of standard mechanism
- Assigns roles to groups of developers
- Roles and delegation make life of Security Officer more easy

But there are more issues…

- Preventing code entering the VM
- Controlling scarce resource usage

Preventing class loading

- We can assign permissions only *after* a class has been loaded
- We would like to *prevent* class loading and initialization

Solution:

- Define a new class load permission
- Check for that permission in `findClass()`

New: ClassLoadPermission

```
public class ClassLoadPermission extends
      java.security.BasicPermission
{
  public ClassLoadPermission(String name)
  {
    super(name);
  }

  public ClassLoadPermission(String name, String action)
  {
    super(name, action);
  }
}
```

DelegatingCL.findClass

```
protected Class findClass(final String name)
       throws ClassNotFoundException
{
  // Try to load the class using the URL class loader
  final Class loadedClass = super.findClass(name);

  // If the class is loaded, get its protection domain.
     ProtectionDomain pd = (ProtectionDomain)
        AccessController.doPrivileged(
        new PrivilegedAction() {
            public Object run() {
              return loadedClass.getProtectionDomain();
            }});
```

findClass continued

```
// If the protection domain is lacking
// ClassLoadPermission, do as if the class could not
// be found.
    if (!pd.implies(new ClassLoadPermission(name))) {
      throw new ClassNotFoundException(name);
    }
    return loadedClass;
}
```

Protecting Scarce Resources

- Protect physical resources like
  - CPU
  - Files system
  - Network I/O
  - Memory
- from
  - Evil code
  - Mistakes easily made…

Can do: prevent creation of threads

- Normally any code may create threads outside the *system* thread group
- We can prevent thread creation by checking for `modifyThreadGroupPermission` for any thread group
- Add the following method to security manager:

checkAccess

```
/** Check access to the ThreadGroup */
public void checkAccess(ThreadGroup g) {
   // First do the standard security check
   super.checkAccess(g);

   // Now do the more strict check: check if the
   // proper permission is there for any thread group
   checkPermission(
        new RuntimePermission("modifyThreadGroup"));

}
```

Thread control not really possible

- If a piece of code has a thread, it *owns* the thread
- There is no guaranteed way to stop a thread!
- Methods are there, but are of no use
  - `Thread.interrupt()`
  - `Thread.destroy()`
  - `Thread.stop()`
- Priority mechanism: also no guarantees
- Conclusion: no thread or CPU resource control possible :-(

Other types of resources

- Memory control permissions: nonexistent
- Same for file and network I/O
- Room for improvement!

Summary

- Custom policy enables roles and delegation
- We can prevent class loading and code execution
- Resource control is only partially possible

Future

- Cooperative multitasking with incentives
  - Stimulate good behavior, punish bad behavior.
  - If an agent really misbehaves, disallow access.
- Agent passport
  - Determine permissions based on identity and travel history: use JAAS.
- Resource control
  - Need VM support for this (please!)

Thank you!

- Sample code: http://www.tryllian.com/download/j1examples.tar.gz
- More info: http://www.tryllian.com
- Questions?
- Email: otto@tryllian.com