Java Security for Application Developers

Milan Potocnik
AMRES/UOB
Poznan, 22.06.2010

Overview

- Java Security Manager
- Java Authentication and Authorization Service
- Java Secure Socket Extension

Java Security Manager – Introduction

- Java Security Manager is part of the Java access control architecture.
- Protects access to sensitive resources or sensitive application code.
- The default security manager provided with JDK - **java.lang.SecurityManager**
- In early versions of Java it was used to limit Java Applets running in the browser.
- The Java SE JDK security architecture is policy-based, and allows for fine-grained access control by checking permissions.

Java Security Manager – Application Servers

- Java Security Manager is commonly used to protect Java Application Servers from malicious third party code.
- Imagine if someone who is authorized to publish JSPs on your site inadvertently included the following in their JSP:
  - <% System.exit(1); %>
- Security Manager can be started:
  - java -Djava.security.manager YourApp
  - System.setSecurityManager(new SecurityManager());

Java Security Manager – Checking Permissions


```
public class File implements Serializable, Comparable<File> {
/* -- Code goes here -- */

public boolean exists() {
  SecurityManager security = System.getSecurityManager();
  if (security != null) {
    security.checkRead(path);
  }
  return ((fs.getBooleanAttributes(this) & FileSystem.BA_EXISTS) != 0);
}
/* -- Code goes here -- */
}
```

- More generic way to check permissions:

`SecurityManager.checkPermission(myPermission)`

Java Security Manager – Policy

- Default policy implementation supports Principal-based grant entries.
- The policy can be specified within one or more policy configuration files.
  - `java -Djava.security.manager -Djava.security.policy=`*`someURL`* `SomeApp`
  - `java -Djava.security.manager -Djava.security.policy==someURL SomeApp`
- Code being executed is grouped in Protection Domains.

```
grant signedBy "signer_names", codeBase "URL",
       principal principal_class_name "principal_name",
       principal principal_class_name "principal_name",
       ... {

     permission permission_class_name "target_name", "action",
         signedBy "signer_names";
     permission permission_class_name "target_name", "action",
         signedBy "signer_names";
     ...
  };
```

Java Security Manager – Policy File Example

```
// ======== SYSTEM CODE PERMISSIONS ===========
// These permissions apply to java
grant codeBase "file:${java.home}/lib/-" {
    permission java.security.AllPermission;
};
// ======== APP SERVER CODE PERMISSIONS =======
// Grant all permissions to ${app.home}/lib/*
grant codeBase "file:${app.home}/lib/-" {
    permission java.security.AllPermission;
};

// ======== PLUGINS PERMISSIONS ===============
// These permissions are granted by default to all plugins.
grant {
    permission java.util.PropertyPermission "java.home", "read";
    permission java.util.PropertyPermission "os.name", "read";
};


// Additional specific permissions can be added here:
// Example:
grant codeBase "file:${app.home}/plugins/example-app/-" {
    permission java.net.SocketPermission "*:1024-65535",
   "connect,accept,resolve";
};
```

Java Security Manager – Dynamic Permissions

```
public class PluginClassLoader extends URLClassLoader {

private SecurityManager securityManager;
protected ArrayList<Permission> permissionList =
                                          new ArrayList<Permission>();

// code goes here
public void addPermission(String path) {
  if (path == null) {
    return;
  }
  if (securityManager != null) {
     Permission permission = null;
     if (!path.endsWith(File.separator)) {
        permission = new FilePermission(path , "read");
        addPermission(permission);
        path = path + File.separator;
     }
     permission = new FilePermission(path + "-“, "read");
     addPermission(permission);
   }
}

public void addPermission(Permission permission){
  if (securityManager != null && permission != null) {
    permissionList.add(permission);
  }
}
} //end of class
```

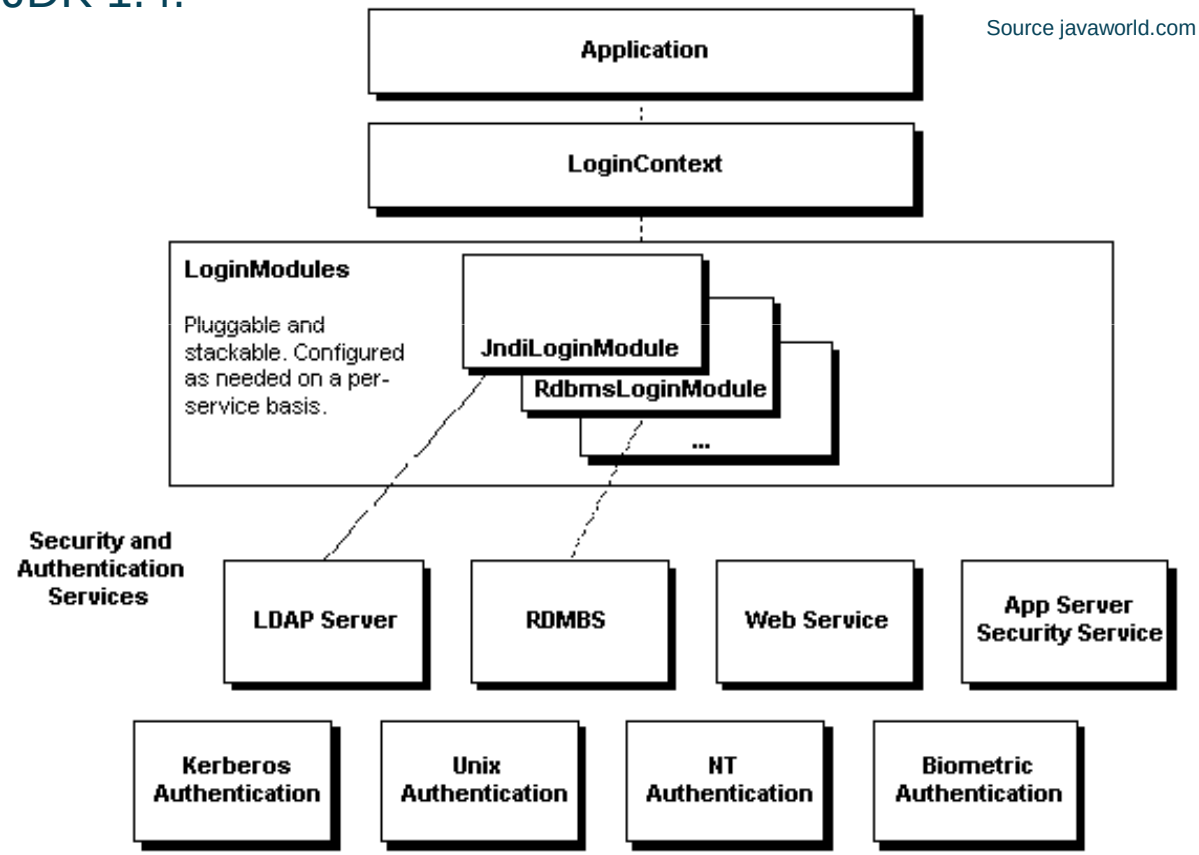
Java Security Manager – Dynamic Permissions (cont.)


```
// Code that loads the plugins

PluginClassLoader cl = new PluginClassLoader(pluginLibURLs);

/* add permissions to classloader */
cl.addPermission(pluginDir);

/* load the appropriate plugin class */
Class<?> c = Class.forName(pluginClass, true, cl);
```

JAAS – Introduction

- Java Authentication and Authorization Service is part of Java platform since JDK 1.4.

Source javaworld.com



JAAS – Introduction (cont.)

- The JAAS-related core classes and interfaces can be broken into three categories:
  - Common: Subject, Principal, Credential.
  - Authentication: LoginContext, LoginModule, CallbackHandler, Callback.
  - Authorization: Policy, AuthPermission, PrivateCredentialPermission.

JAAS – Authentication

- JAAS authentication is performed in a *pluggable* fashion.
- An implementation for a particular authentication technology to be used is determined at runtime. The implementation is specified in a login configuration file.
  - `System.setProperty("java.security.auth.login.config", "jaas.config");`
  - `java -Djava.security.auth.login.config==jaas.config`

```
// example jaas.policy file
example {
  org.jboss.security.ClientLoginModule required;
};
```

JAAS – Authentication (cont.)

- To perform the authentication, user needs to instantiate a **javax.security.auth.login.LoginContext**.
- LoginContext takes two arguments:
  - The name of an entry in the JAAS login configuration file, identifying the application and the ***javax.security.auth.spi.LoginModule*** implementation that will be used.
  - The ***javax.security.auth.callback.CallbackHandler*** instance.
- If **login()** was successful, **commit()** will set proper Subject's credentials. Calling l**ogout()** will clear these credentials.

```
UsernamePasswordHandler handler;
handler = new UsernamePasswordHandler("jack", "black");
LoginContext lc = new LoginContext("example", handler);
lc.login();

//perform a restricted operation that requires login

lc.logout();
```

JAAS – Authorization

- To be able to authorize, the user first needs to be authenticated.
- Authorization component ensures that the authenticated caller has the access control rights (permissions) required to do security-sensitive operations.
- JAAS authorization extends the existing Java security architecture that uses a security policy to specify what access rights are granted to executing code.
- Besides controlling where the code is coming from and who signed it, it is also possible to control who is running it.

```
grant codebase "file:./SampleAction.jar",
        Principal sample.principal.SamplePrincipal “jack" {

   permission java.util.PropertyPermission "java.home", "read";
   permission java.io.FilePermission "foo.txt", "read";
};
```

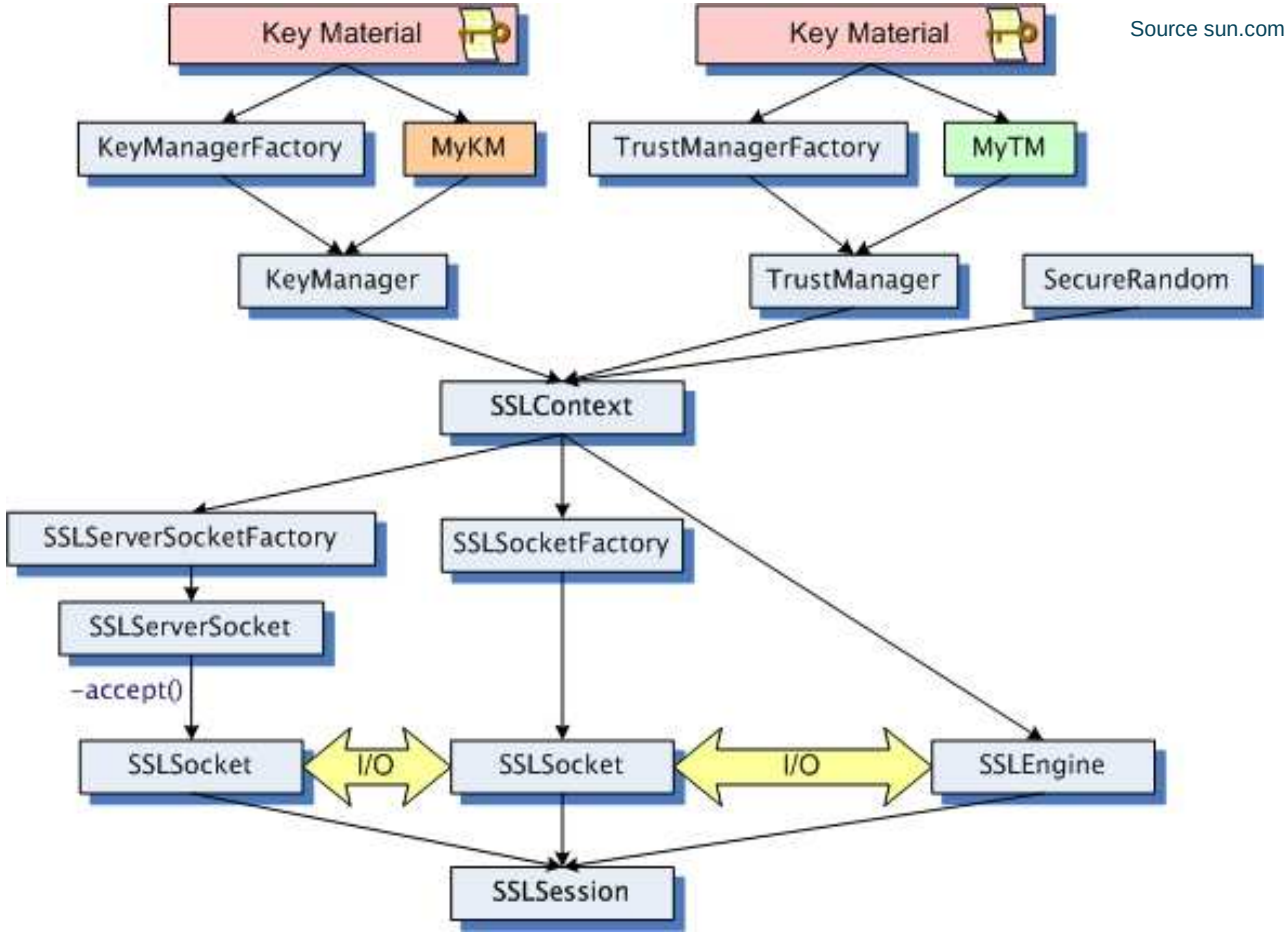
JAAS – Authorization (cont.)


```
UsernamePasswordHandler handler;
handler = new UsernamePasswordHandler("jack", "black");
LoginContext lc = new LoginContext("example", handler);
lc.login();

Subject mySubject = lc.getSubject();

// now try to execute some action as the authenticated Subject
Subject.doAsPrivileged(mySubject, new PrivilegedAction<String>() {
  public String run() {
    // do privileged code here
    String home = System.getProperty("java.home");
    return home;
  }
}, null);
lc.logout();
```

JSSE – Introduction

- Java Secure Socket Extension (JSSE) enables secure Internet communications.
- Framework and an implementation for a Java version of the SSL and TLS protocols.
- JSSE APIs were designed to allow other SSL/TLS protocol and Public Key Infrastructure (PKI) implementations to be plugged in seamlessly.

JSSE – API Overview



Source sun.com

JSSE – Features

- Secure (SSL) sockets and server sockets.
- A non-blocking engine for producing and consuming streams of SSL/TLS data (SSLEngine).
- Factories for creating sockets, server sockets, SSL sockets, and SSL server sockets.
- A class representing a secure socket context.
- Key and trust manager interfaces (including X.509-specific key and trust managers).
- A class for secure HTTP URL connections (HTTPS).

JSSE – Secure Sockets Layer

- A secure enhancement to the standard TCP/IP sockets protocol used for Internet communications:
  - Application Layer: HTTP, NNTP, Telnet, FTP, etc…
  - Secure Sockets Layer (SSL)
  - Transport Layer: TCP
  - Internet Layer IP
- Communication using SSL begins with an exchange of information between the client and the server (SSL handshake):
  - Negotiate the cipher suite.
  - Authenticate identity (optional).
  - Establish information security by agreeing on encryption mechanisms.

JSSE – TrustManager & KeyManager

- TrustManager
  - Determines whether the remote authentication credentials should be trusted.
  - System properties: `javax.net.ssl.trustStore`, `javax.net.ssl.trustStoreType`
- KeyManager
  - Determines which authentication credentials to send to the remote host.
  - System properties: `javax.net.ssl.keyStore`, `javax.net.ssl.keyStoreType`

JSSE – SSL Context Example

```
KeyStore ks = KeyStore.getInstance("pkcs12");
/* get user password and file input stream */
char[] password =
   System.getProperty("javax.net.ssl.keyStorePassword").toCharArray();
FileInputStream fis = new FileInputStream(keyStoreFile);
ks.load(fis, password);

KeyManagerFactory kmf = KeyManagerFactory.getInstance("SUNX509");
kmf.init(ks, password);
KeyManager[] km = kmf.getKeyManagers();

SSLContext sc = SSLContext.getInstance("SSL");
sc.init(km, new TrustManager[] { new CustomX509TrustManager() }, new
   java.security.SecureRandom());

SSLServerSocket serverSocket = (SSLServerSocket)
   sc.getServerSocketFactory().createServerSocket(port);
serverSocket.setNeedClientAuth(requireClientAuth);
```

JSSE – SSL Session Example


```
SSLSocket socket = (SSLSocket) serverSocket.accept();


X509Certificate[] certChain = (X509Certificate[])
   socket.getSession().getPeerCertificates();
X509Certificate clientCert = certs[CertUtil.findClientCert(certs)];
String userIdentity = clientCert.getSubjectX500Principal()).getX500();


logger.info("Authorization check for: " + userIdentity);
// perform some authz based on user's DN
// *******


// if authorization succeeded communicate with client
OutputStream out = socket.getOutputStream();
InputStream in = socket.getInputStream();
```